Altova UModel 2025 Professional Edition

User & Reference Manual

Altova UModel 2025 Professional Edition
User & Reference Manual

All rights reserved. No parts of this work may be reproduced in any form or by any means - graphic, electronic, or mechanical, including photocopying, recording, taping, or information storage and retrieval systems - without the written permission of the publisher.

Products that are referred to in this document may be either trademarks and/or registered trademarks of the respective owners. The publisher and the author make no claim to these trademarks.

While every precaution has been taken in the preparation of this document, the publisher and the author assume no responsibility for errors or omissions, or for damages resulting from the use of information contained in this document or from the use of programs and source code that may accompany it. In no event shall the publisher and the author be liable for any loss of profit or any other commercial damage caused or alleged to have been caused directly or indirectly by this document.

Published: 2025

© 2019-2025 Altova GmbH

Table of Contents

1 Introduction 12

1.1 Support Notes 13
1.2 Database Support 16

2 UModel Tutorial 18

2.1 Getting Started 19
2.2 Use Cases 22
2.3 Class Diagrams 31
2.3.1 Creating Derived Classes 40
2.4 Object Diagrams 46
2.5 Component Diagrams 53
2.6 Deployment Diagrams 59
2.7 Forward Engineering (from Model to Code) 64
2.8 Reverse Engineering (from Code to Model) 73

3 UModel Graphical User Interface 81

3.1 Model Tree Window 83
3.2 Diagram Tree Window 87
3.3 Favorites Window 88
3.4 Properties Window 89
3.5 Styles Window 90
3.6 Hierarchy Window 91
3.7 Overview Window 93
3.8 Documentation Window 94
3.9 Layer Window 95
3.10 Messages Window 96
3.11 Diagram Window 98
3.12 Diagram Pane 99

4 UModel Command Line Interface 101

4.1 Creating, Loading, and Saving Projects in Batch Mode ... 106

5 How to Model... 108

5.1 Elements ... 109

5.1.1 Creating Elements ... 109

5.1.2 Inserting Elements from the Model into a Diagram ... 110

5.1.3 Renaming, Moving, and Copying Elements ... 112

5.1.4 Deleting Elements ... 113

5.1.5 Converting Elements ... 114

5.1.6 Finding and Replacing Text ... 114

5.1.7 Checking Where and If Elements Are Used ... 116

5.1.8 Constraining Elements ... 117

5.1.9 Hyperlinking Elements ... 118

5.1.10 Documenting Elements ... 121

5.1.11 Changing the Style of Elements ... 122

5.2 Diagrams ... 124

5.2.1 Creating Diagrams ... 124

5.2.2 Generating Diagrams ... 125

5.2.3 Opening Diagrams ... 127

5.2.4 Deleting Diagrams ... 128

5.2.5 Changing the Style of Diagrams ... 128

5.2.6 Aligning and Resizing Modeling Elements ... 130

5.2.7 Adding Layers to Diagrams ... 132

5.2.8 Type Autocompletion in Classes ... 134

5.2.9 Zooming into/out of Diagrams ... 135

5.3 Relationships ... 136

5.3.1 Creating Relationships ... 136

5.3.2 Changing the Style of Lines and Relationships ... 137

5.3.3 Viewing Element Relationships ... 139

5.3.4 Associations ... 139

5.3.5 Collection Associations ... 142

5.3.6 Containment ... 145
5.4 Stereotypes and Tagged Values ... 146
5.4.1 Tagged Values ... 147
5.4.2 Applying Stereotypes ... 148
5.4.3 Showing or Hiding Tagged Values ... 150

6 Projects and Code Engineering 153

6.1 Managing UModel Projects ... 154
6.1.1 Creating, Opening, and Saving Projects ... 154
6.1.2 Opening Projects from a URL ... 155
6.1.3 Moving Projects to a New Directory ... 159
6.1.4 Applying UModel Profiles ... 160
6.1.5 Splitting UModel Projects ... 161
6.1.6 Including Subprojects ... 164
6.1.7 Sharing Packages and Diagrams ... 166
6.1.8 Tips for Enhancing Performance ... 169
6.2 Generating Program Code ... 170
6.2.1 Setting a Package as Namespace Root ... 170
6.2.2 Adding a Code Engineering Component ... 171
6.2.3 Checking Project Syntax ... 173
6.2.4 Code Generation Options ... 175
6.2.5 Example: Generate C# Code ... 177
6.2.6 Example: Generate Java Code ... 182
6.2.7 SPL Templates ... 190
6.3 Importing Source Code ... 192
6.3.1 Code Import Options ... 194
6.3.2 Example: Import a C# Project ... 196
6.4 Importing Java, C# and VB.NET Binaries ... 204
6.4.1 Adding Custom Java Runtimes ... 205
6.4.2 Import Binary Options ... 205
6.4.3 Example: Import .NET Assemblies ... 209
6.4.4 Example: Import Java .class Files ... 211
6.5 Synchronizing the Model and Source Code ... 217
6.5.1 Synchronization Tips ... 218

6.5.2 Refactoring Code and Synchronization ... 220
6.5.3 Code Synchronization Settings ... 221
6.6 UModel Element Mappings ... 224
6.6.1 C# Mappings ... 224
6.6.2 VB.NET Mappings ... 244
6.6.3 Java Mappings ... 258
6.6.4 XML Schema Mappings ... 264
6.6.5 Database Mappings ... 273
6.7 Merging UModel Projects ... 276
6.7.1 3-Way Project Merge ... 276
6.7.2 Example: Manual 3-Way Project Merge ... 278
6.8 UML Templates ... 281
6.8.1 Template Signatures ... 282
6.8.2 Template Binding ... 283
6.8.3 Template Usage in Operations and Properties ... 283

7 Generating UML Documentation 285

7.1 Documentation Generation Options ... 289
7.2 Customizing Output with StyleVision ... 294

8 UML Diagrams 296

8.1 Behavioral Diagrams ... 297
8.1.1 Activity Diagram ... 297
8.1.2 State Machine Diagram ... 314
8.1.3 Protocol State Machine ... 337
8.1.4 Use Case Diagram ... 342
8.1.5 Communication Diagram ... 342
8.1.6 Interaction Overview Diagram ... 346
8.1.7 Sequence Diagram ... 351
8.1.8 Timing Diagram ... 378
8.2 Structural Diagrams ... 387
8.2.1 Class Diagram ... 387
8.2.2 Composite Structure Diagram ... 401

8.2.3 Component Diagram ... 404
8.2.4 Deployment Diagram ... 404
8.2.5 Object Diagram ... 405
8.2.6 Package Diagram ... 405
8.2.7 Profile Diagram ... 411
8.3 Additional Diagrams ... 424
8.3.1 XML Schema Diagrams ... 424
8.3.2 Business Process Modeling Notation 1.0 / 2.0 ... 441
8.3.3 SysML Diagrams ... 468

9 UModel and Databases 486

9.1 Modeling Databases in UModel ... 487
9.1.1 Importing SQL Databases into UModel ... 488
9.1.2 Designing Database Objects ... 495
9.1.3 Configuring Round-Trip Engineering for Databases ... 500
9.1.4 Example: Update a Database from the Model ... 501
9.2 Connect to a Data Source ... 507
9.2.1 Start Database Connection Wizard ... 508
9.2.2 Database Drivers Overview ... 510
9.2.3 ADO Connection ... 514
9.2.4 ADO.NET Connection ... 519
9.2.5 JDBC Connection ... 525
9.2.6 ODBC Connection ... 528
9.2.7 SQLite Connection ... 531
9.2.8 Native Connections ... 532
9.2.9 Database Connection Examples ... 533

10 XMI - XML Metadata Interchange 568

11 UModel Plug-in for Visual Studio 570

11.1 Installing the UModel Plug-in for Visual Studio ... 572
11.2 Adding UModel Support to Visual Studio Projects ... 573
11.3 Loading/Unloading UModel Projects ... 577

11.4 Synchronizing the Model and Code ... 578

12 UModel Plug-in for Eclipse 581

12.1 Installing the UModel Plug-in for Eclipse ... 584
12.2 The UModel Perspective ... 586
12.3 Adding UModel Support to Eclipse Projects ... 589
12.4 Importing Existing UModel Projects ... 591
12.5 Loading/Unloading UModel Projects ... 593
12.6 How Automatic Synchronization Works ... 594
12.7 Example: Setting up Automatic Synchronization ... 595

13 Source Control 607

13.1 Setting Up Source Control ... 609
13.2 Supported Source Control Systems ... 610
13.3 Source Control Commands ... 612
13.3.1 Open from Source Control ... 612
13.3.2 Enable Source Control ... 615
13.3.3 Get Latest Version ... 616
13.3.4 Get ... 616
13.3.5 Get Folder(s) ... 617
13.3.6 Check Out ... 618
13.3.7 Check In ... 620
13.3.8 Undo Check Out ... 620
13.3.9 Add to Source Control ... 622
13.3.10 Remove from Source Control ... 624
13.3.11 Share from Source Control ... 625
13.3.12 Show History ... 626
13.3.13 Show Differences ... 628
13.3.14 Show Properties ... 629
13.3.15 Refresh Status ... 630
13.3.16 Source Control Manager ... 630
13.3.17 Change Source Control ... 630
13.4 Source Control with Git ... 632

13.4.1 Enabling Git Source Control with GIT SCC Plug-in ... 633
13.4.2 Adding a Project to Git Source Control ... 633
13.4.3 Cloning a Project from Git Source Control ... 635

14 UModel Diagram icons 637

14.1 Activity Diagram ... 638
14.2 Class Diagram ... 640
14.3 Communication diagram ... 641
14.4 Composite Structure Diagram ... 642
14.5 Component Diagram ... 643
14.6 Deployment Diagram ... 644
14.7 Interaction Overview diagram ... 645
14.8 Object Diagram ... 646
14.9 Package diagram ... 647
14.10 Profile Diagram ... 648
14.11 Protocol State Machine ... 649
14.12 Sequence Diagram ... 650
14.13 State Machine Diagram ... 651
14.14 Timing Diagram ... 652
14.15 Use Case diagram ... 653
14.16 XML Schema diagram ... 654
14.17 Business Process Modeling Notation ... 655
14.18 Business Process Modeling Notation 2.0 ... 657
14.19 Database Modeling ... 658

15 Menu Reference 659

15.1 File ... 660
15.2 Edit ... 662
15.3 Project ... 664
15.4 Layout ... 667
15.5 View ... 668
15.6 Tools ... 669
15.6.1 Spelling ... 669

15.6.2 Spelling Options ... 673
15.6.3 Scripting Editor ... 675
15.6.4 Macros ... 675
15.6.5 User-defined Tools ... 675
15.6.6 Customize ... 675
15.6.7 Restore Toolbars and Windows ... 685
15.6.8 Options ... 685
15.7 Window ... 698
15.8 Help ... 700

16 UModel Programmer's Reference 705

16.1 Scripting Editor ... 707
16.1.1 Creating a Scripting Project ... 708
16.1.2 Built-in Commands ... 720
16.1.3 Enabling Scripts and Macros ... 730
16.2 UModel IDE Plug-Ins ... 733
16.2.1 How to Create a UModel IDE Plug-In ... 733
16.2.2 Deployment of UModel IDE Plug-Ins ... 742
16.2.3 Configuration XML ... 744
16.2.4 Plug-Ins as ActiveX Controls ... 747
16.2.5 IUModelPlugIn Interface ... 748
16.3 The UModel API ... 752
16.3.1 Accessing the API ... 752
16.3.2 Object Model ... 753
16.3.3 How to ... 759
16.3.4 C# API Examples ... 771
16.3.5 Java API Example ... 797
16.3.6 JScript Examples ... 799
16.4 UModel API Reference ... 814
16.4.1 UModel Plug-Ins ... 814
16.4.2 UModel API Interfaces ... 816
16.4.3 UMLData Interfaces ... 903

17 SPL Reference 1274

17.1 Basic SPL structure 1275
17.2 Variables 1276
17.3 Operators 1285
17.4 Conditions 1286
17.5 Collections and foreach 1287
17.6 Subroutines 1289
17.6.1 Subroutine declaration 1289
17.6.2 Subroutine invocation 1290

18 License Information 1291

18.1 Electronic Software Distribution 1292
18.2 Software Activation and License Metering 1293
18.3 Altova End-User License Agreement 1295
18.4 Packaging License Files with UModel Installer 1296

Index 1297

1 Introduction

Altova UModel 2025 Professional Edition is a UML modeling application with a rich visual interface and superior usability features to help level the UML learning curve. UModel includes many high-end functions to empower users with the most practical aspects of the UML 2.5 specification. UModel is a 32/64-bit Windows application that runs on Windows 10, Windows 11, and Windows Server 2016 or newer. 64-bit support is available for the Enterprise and Professional editions. For an overview of UModel capabilities, see Support Notes 13.

UML®, OMG™, Object Management Group™, and Unified Modeling Language™ are either registered trademarks or trademarks of Object Management Group, Inc. in the United States and/or other countries.

Last updated: 17 March 2025

Altova website: UML tool

© 2019-2025 Altova GmbH

1.1 Support Notes

UModel is a 32/64-bit Windows application that runs on the following operating systems:

- Windows Server 2016 or newer
- Windows 10, Windows 11

64-bit support is available for the Enterprise and Professional editions.

UML diagrams

UModel supports all fourteen diagrams of the UML 2.5.1 specification, and additional specialized diagram types.

Structural	Behavioral	Additional
Class Diagrams	Activity Diagram	XML Schema Diagrams
Component Diagram	Communication Diagram	BPMN (Business Process Modeling Notation) 1.0 / 2.0 Diagrams (*UModel Enterprise and Professional editions*)
Composite Structure Diagram	Interaction Overview Diagram	SysML 1.2, 1.3, 1.4, 1.5, 1.6 Diagrams (*UModel Enterprise and Professional editions*)
Deployment Diagram	Sequence Diagram	Database Diagrams (*UModel Enterprise and Professional editions*)
Object Diagram	State Diagrams (State Machine and Protocol State Machine)	
Package Diagram	Timing Diagram	
Profile Diagram	Use Case Diagram	

UModel has been designed to allow complete flexibility during the modeling process:

- UModel diagrams can be created in any order, and at any time; there is no need to follow a prescribed sequence during modeling.
- The syntax coloring in diagrams is customizable. For example, you can customize modeling elements and their properties (font, color, borders, etc.) in a hierarchical fashion at the project, node/line, element family and element level, see Changing the Style of Elements 122.
- The unlimited levels of Undo/Redo track not only content changes, but also all style changes made to any model element.
- Modeling elements support hyperlinks, see Hyperlinking Elements 118.
- You can create multiple layers in the same UML diagram, see Adding Layers to Diagrams 132.

Code engineering and import of binaries

UModel supports code generation and reverse engineering of program code written in the following languages:

Language	Code engineering	Import of binaries
C#	1.2, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 7.1, 7.2, 7.3, 8.0, 9.0[1], 10	Same language versions as for code engineering[2]
C++ (*UModel Enterprise Edition*)	C++98, C++11 and C++14, C++17, C++20 Only partial support for C++20: modules are not supported.	Not applicable
Java	1.4, 5.0 (1.5), 6 (1.6), 7 (1.7), 8 (1.8), 9 (1.9), 10, 11, 12, 13, 14, 15, 16, 17, 18, 19	Same language versions as for code engineering[3]
Visual Basic .NET	7.1 or newer	Same language versions as for code engineering
XML Schemas[4]	1.0	Not applicable
Databases[5] (*UModel Enterprise and Professional editions*)	For more information about supported databases, see Database Support 16.	Not applicable

Table footnotes:

1. If you import binary files compiled from C# 9.0 code, note that any *records* will be imported as *classes*. This limitation is due to the fact that records are marked as classes in the assembly, which makes it impossible to distinguish them from classes.
2. C# code engineering and import of binaries include support for .NET Framework, .NET Core, .NET 5, and .NET 6. Note that .NET Framework, .NET Core, .NET 5 or .NET 6 must be installed, as applicable. Binaries of other .NET implementations which are not mentioned are likely to be imported as well. See also Importing Java, C# and VB.NET Binaries 204.
3. It is also possible to import binaries targeting Java Virtual Machines other than Oracle JDK, such as OpenJDK, SapMachine, Liberica JDK, and others, see Adding Custom Java Runtimes 205.
4. In the case of XML Schemas, code engineering means that you can import a schema (or multiple schemas from a directory) into UModel, view or modify the model, and write the changes back to the schema file. When you synchronize data from the model to a schema file, the schema file is always overwritten by the model. See also XML Schema Diagrams 424.
5. In the case of databases, code engineering means that you can (i) model a database in UModel with the option to update the database through a script generated from the model, or (ii) import an existing database structure into a model, make changes to it, and then deploy a script generated from the model to the database. Some database object types are not supported for modeling. For details, see UModel and Databases 486.

General notes:

© 2019-2025 Altova GmbH

- You can synchronize the code and model at the project, package, or even class level. UModel does not require that pseudo-code, or comments in the generated code be present, in order to accomplish round-trip engineering.
- A single project can support Java, C#, or VB.NET code simultaneously.
- UModel supports the use of UML templates and their mapping to or from Java, C# and Visual Basic generics.
- While importing source code, you can optionally generate Class 399 and Package 408 diagrams. Once the source code is imported into the model, you can also generate Sequence 366 diagrams.
- You can generate program code from Sequence diagrams 372 and from State Machine diagrams 326
- UModel projects can be split up into multiple sub-projects allowing several developers to simultaneously edit different parts of a single project. You can then reintegrate the changes back into a common model. You can also merge UModel projects, as a 2-way or as a 3-way merge, see Merging UModel Projects 276.
- Code generation in UModel is based on Spy Programming Language (SPL) templates and is customizable.

UML documentation generation

You can generate documentation from UModel projects in HTML, RTF, Microsoft Word 2000 or later formats. Various options are available that let you configure the level of detail of generated documentation, the look and feel, and other preferences. Generating documentation in PDF format and deep customization of document generation templates is possible with Altova StyleVision (https://www.altova.com/stylevision). For more information, see Generating UML Documentation 285.

IDE Integration

UModel is optionally available as a plug-in to the following integrated development environments:

- Visual Studio 2012/2013/2015/2017/2019/2022, see UModel Plug-in for Visual Studio 570
- Eclipse 2024-12 (4.34), 2024-09 (4.33), 2024-06 (4.32), 2024-03 (4.31), see UModel Plug-in for Eclipse 581

UModel provides a COM-based API 752 and also allows integration of custom IDE Plug-Ins 733 (DLL libraries) into its graphical user interface. The Scripting Editor 707 allows for development of custom VBScript or JScript scripts and macros to automate various tasks.

Microsoft Office integration

By virtue of its database modeling support, UModel can import Access databases into a model, and generate SQL scripts for Access databases. For more information, see UModel and Databases 486.

Interoperability

UModel also provides support for importing or exporting projects to or from XML Metadata Interchange (XMI) format, see XMI - XML Metadata Interchange 568.

1.2 Database Support

The table below lists all the supported databases. If your Altova application is a 64-bit version, ensure that you have access to the 64-bit database drivers needed for the specific database you are connecting to.

Database	Notes
Firebird 2.x, 3.x, 4.x	
IBM DB2 8.x, 9.x, 10.x, 11.x, 12.x	
IBM Db2 for i 6.x, 7.4, 7.5	Logical files are supported and shown as views.
IBM Informix 11.70 and later	
MariaDB 10 and later	MariaDB supports native connections. No separate drivers are required.
Microsoft Access 2003 and later	You can connect to an Access 2019 database from Altova products (i) only if the corresponding version of Microsoft Access Runtime is installed and (ii) only if the database does not use the "Large Number" data type.
Microsoft Azure SQL Database	SQL Server 2016 codebase
Microsoft SQL Server 2005 and later Microsoft SQL Server on Linux	
MySQL 5 and later	MySQL 5.7 and later supports native connections. No separate drivers are required.
Oracle 9i and later	
PostgreSQL 8 and later	PostgreSQL connections are supported both as native connections and driver-based connections through interfaces (drivers) such as ODBC or JDBC. Native connections do not require any drivers.
Progress OpenEdge 11.6	
SQLite 3.x	SQLite connections are supported as native, direct connections to the SQLite database file. No separate drivers are required.

© 2019-2025 Altova GmbH

Database	Notes
Sybase ASE 15, 16	
Teradata 16	

2 UModel Tutorial

This tutorial shows you how to create various UML diagrams with UModel, while acquainting you with the graphical user interface. You will also learn how to generate code from a UML model (forward engineering) as well as how to import existing code into a UML model (reverse engineering). With respect to code engineering, you will also learn how to perform full round-trip engineering (either model->code->model or code->model->code). This tutorial assumes basic knowledge of the UML.

The tutorial is organized into sections as shown below. In the initial sections of this tutorial you will be working with a sample project pre-installed with UModel. If you would like to quickly create a new modelling project from scratch with UModel, you can skip directly to Forward Engineering (from Model to Code) 64.

- Getting Started 19
- Use Cases 22
- Class Diagrams 31
- Creating Derived Classes 40
- Object Diagrams 46
- Component Diagrams 53
- Deployment Diagrams 59
- Forward Engineering (from Model to Code) 64
- Reverse Engineering (from Code to Model) 73

This tutorial makes use of the following sample UModel project files available in the directory **C:\Users\<username>\Documents\Altova\UModel2025\UModelExamples\Tutorial**:

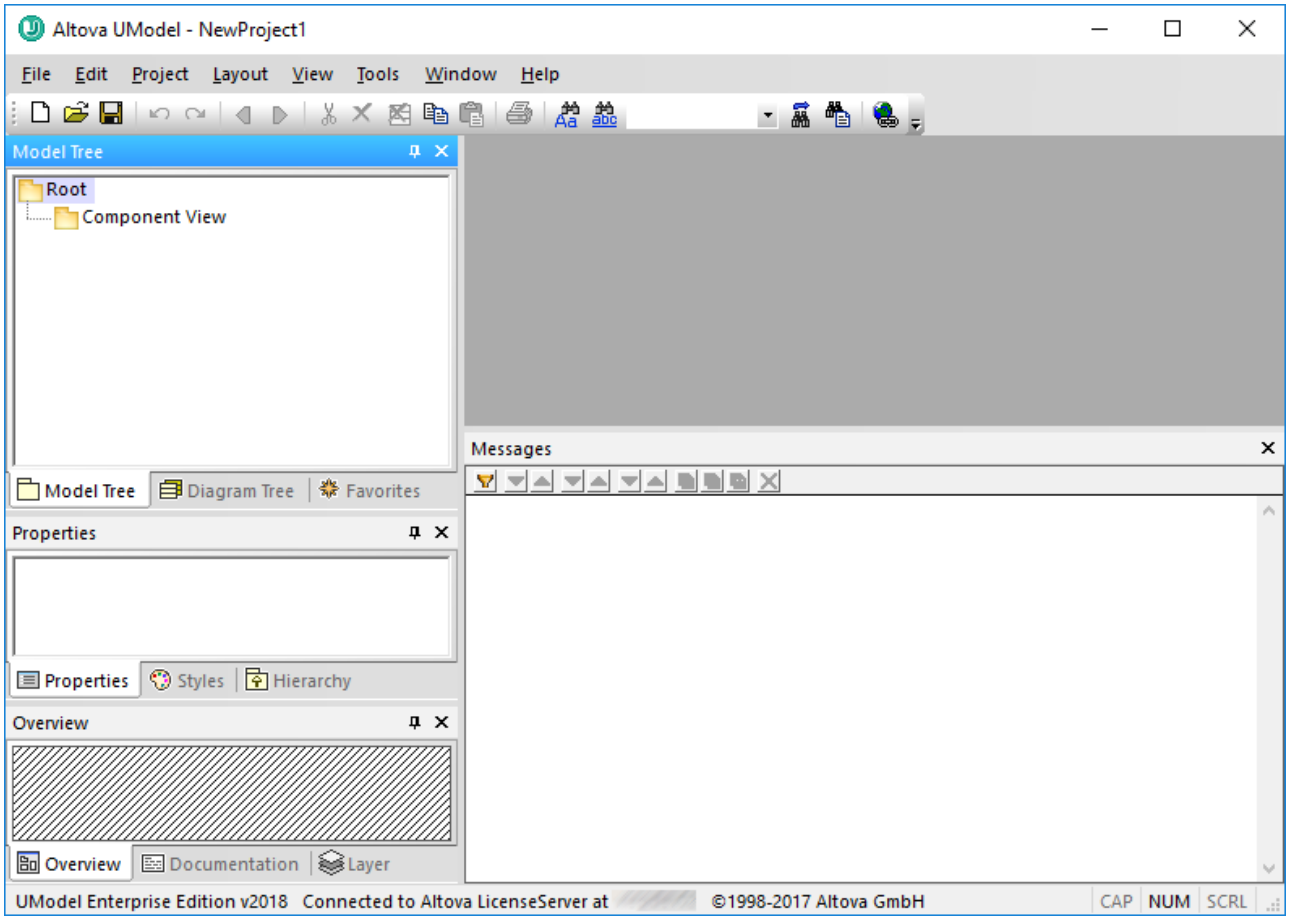
BankView-start.ump	This is the UModel project file that constitutes the initial state of the tutorial sample. Several model diagrams as well as classes, objects, and other model elements exist in this project. By working through the tutorial, you will be adding new elements or diagrams, or editing existing ones, using UModel. Note: This project is deliberately incomplete, so validation errors and warnings will be shown if you check the project syntax using the **Project | Check Project Syntax** menu command. The tutorial shows you how to resolve these issues.
BankView-finish.ump	This is the UModel project file that constitutes final state of the tutorial sample.

Note: All UModel example files are initially available in the directory **C:\ProgramData\Altova\UModel2025**. When any user starts the application for the first time, the example files are copied to **C:\Users\<username>\Documents\Altova\UModel2025\UModelExamples**. Therefore, do not move, edit, or delete the example files in the initial directory.

© 2019-2025 Altova GmbH

2.1 Getting Started

When you start UModel for the first time after installation, it opens a default empty project "NewProject1". On subsequent runs, UModel will open the last project that was loaded. To create, open, and save UModel projects (.ump files), use the standard Windows commands available in the **File** menu or in the toolbar.



UModel Graphical User Interface

Note the major parts of the user interface: multiple helper windows on the left hand side and the main diagram window to the right. Two default packages are visible in the Model Tree window, "Root" and "Component View". These two packages cannot be deleted or renamed in a project.

The helper windows in the upper-left area are as follows:

- The **Model Tree** window contains and displays all modeling elements of your UModel project. Elements can be directly manipulated in this window using the standard editing keys as well as drag and drop.
- The **Diagram Tree** window allows your quick access to the modeling diagrams of you project wherever they may be in the project structure. Diagrams are grouped according to their diagram type.
- The **Favorites** window is a user-definable repository of modeling elements. Any type of modeling element can be placed in this window using the "Add to Favorites" command of the context menu.

The helper windows in the middle-left area are as follows:

- The **Properties** window displays the properties of the currently selected element in the **Model Tree** window or in the **Diagram** window. Element properties can defined or updated in this window.
- The **Styles** window displays attributes of diagrams, or elements that are displayed in the Diagram view. These style attributes fall into two general groups: Formatting and display settings.
- The **Hierarchy** window displays all relations of the currently selected modeling item, in two different views. The modeling element can be selected in a modeling diagram, the Model Tree, or in the **Favorites** window.

The helper windows in the lower-left area are as follows:

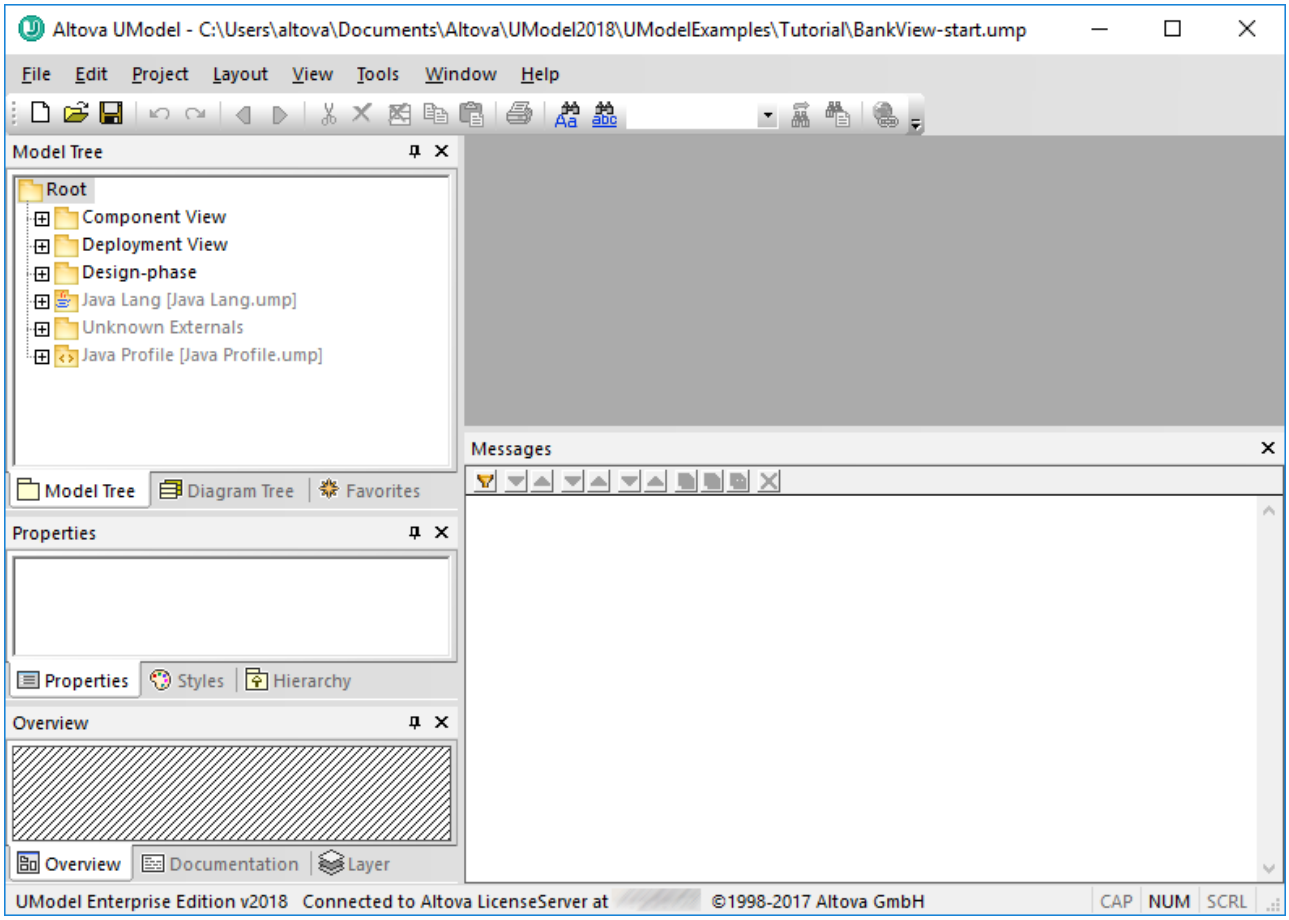
- The **Overview** window which displays an outline view of the currently active diagram.
- The **Documentation** window which allows you to document your classes on a per-class basis.
- The **Layer** window allows you to define multiple layers for any UModel diagram. Single, as well as multiple, layers can be shown, locked and hidden. Layers allow you to make logical groupings of modeling elements on a diagram.

In this tutorial, you will be working mostly within the **Model Tree** and **Diagram Tree** windows, as well as the main diagram window. For further information about the graphical user interface elements, see UModel User Interface 81.

To open the tutorial project:

1. Select the menu option **File | Open** and navigate to the …**UModelExamples\Tutorial** folder of UModel. Note that you can also open a *.ump file through a URL, please see Switch to URL 660 for more information.
2. Open the **BankView-start.ump** project file. The project file is now loaded into UModel. Several predefined packages are now visible under the Root package. Note that the main window is empty at the moment.

© 2019-2025 Altova GmbH



BankView-start.ump project

2.2 Use Cases

This tutorial section shows you how to create a Use Case diagram, while acquainting you with the basics of the UModel graphical user interface. Specifically, it illustrates the following tasks:

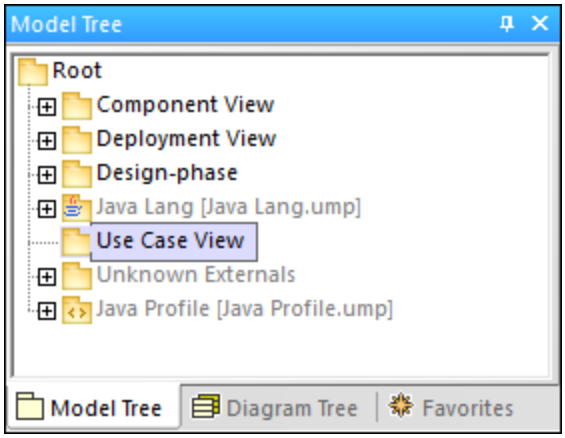
- Add a new package to the project
- Add a new use case diagram to the project
- Add use case elements to the diagram, and define the dependencies amongst them
- Align and adjust the size of elements in the diagram
- Change the style of all diagrams in a UModel project.

To proceed, run UModel and open the **BankView-start.ump** project (see also Opening the Tutorial Project 19).

Adding a new package to a project

As you already know from UML, a package is a container for organizing classes and other UML elements, including use cases. Let's begin by creating a package that will store a new use case diagram. Note that UModel does not require that a specific diagram must reside in a specific package; however, you might want to organize diagrams into packages for better organization and consistency.

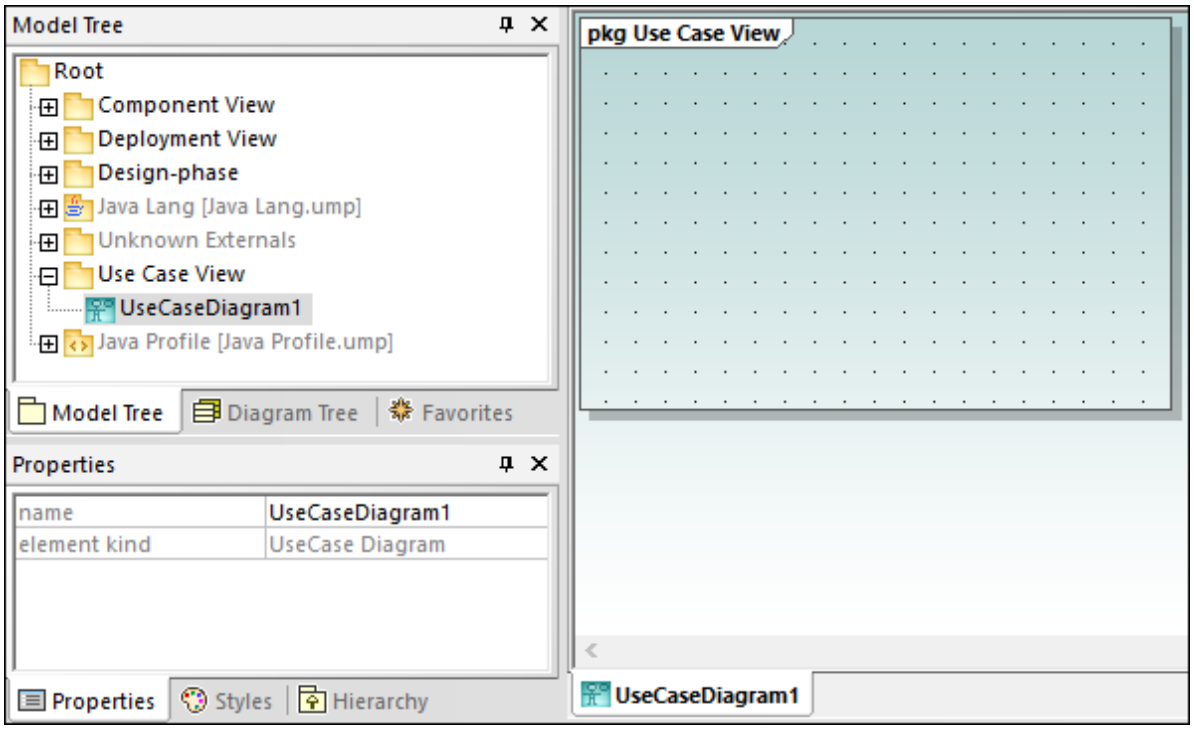
1. Right-click the **Root** package in the Model Tree window, and select **New Element | Package**.
2. Enter the name of the new package (in this example, "Use Case View"), and press **Enter**.



Adding a Use Case diagram to a package

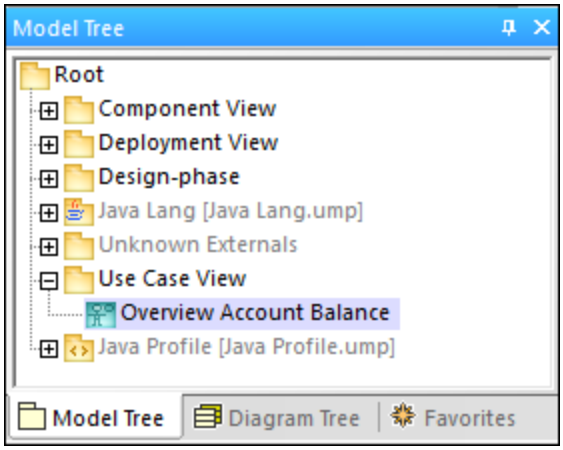
1. Right-click the previously created "Use Case View" package.
2. Select **New Diagram | UseCase Diagram**.

© 2019-2025 Altova GmbH



A Use Case diagram has now been added to the package in the **Model Tree** window, and a new **Diagram** window has been created as well. A default name has been provided automatically.

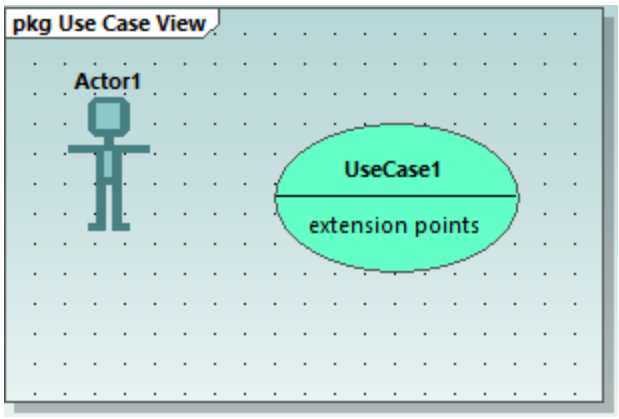
3. Double-click the diagram name in the **Model Tree** window, change it to "Overview Account Balance", and press **Enter** to confirm.



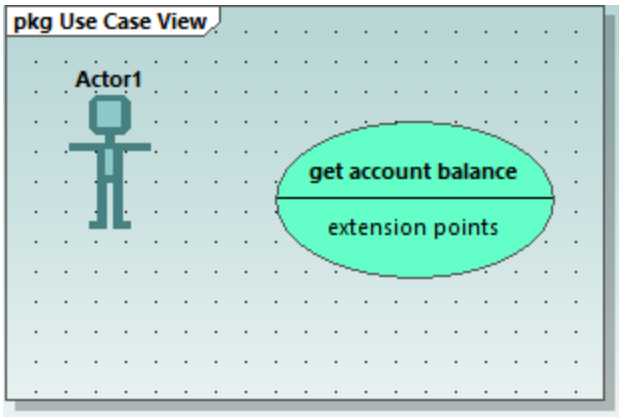
Adding Use Case elements to the Use Case diagram

1. Right-click in the newly created diagram and select **New | Actor**. The actor element is inserted at the click position.

2. Click the **Use Case** toolbar button and then click inside the diagram window to insert the element. A "UseCase1" element is inserted. Note that the element, and its name, are currently selected, and that its properties are visible in the **Properties** window.



3. Change the title to "get account balance", press **Enter** to confirm. Double-click the title if it is deselected. Note that the use case is automatically resized to adjust to the text length.



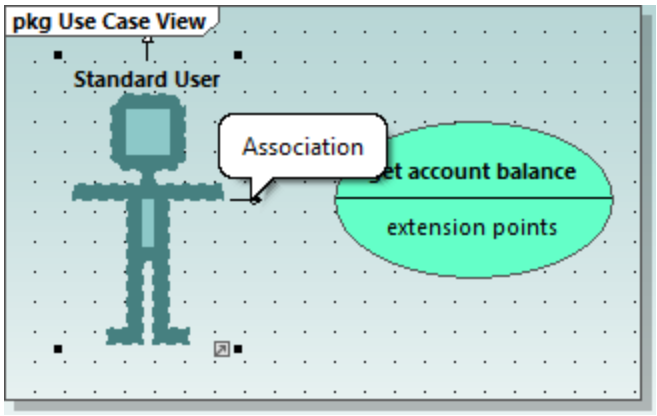
Note: To create a multi-line use case name, press **Enter** while holding the **Ctrl** key pressed.

Manipulating UModel elements: handles and compartments

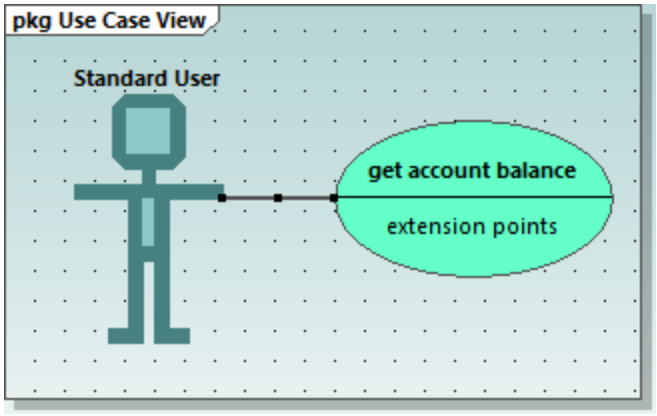
When selected, model elements in a diagram display various connection handles and other items used to manipulate them. Handles can be used to create relationships between elements, or show or hide certain compartments from the element, as shown below.

1. Double-click the "Actor1" text of the Actor element, change the name to "Standard User" and press **Enter** to confirm.
2. Place the mouse cursor over the handle to the right of the actor. A tooltip containing "Association" appears.

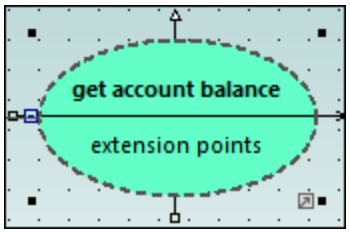
© 2019-2025 Altova GmbH



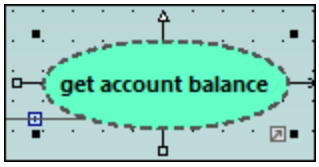
3. Click the handle, drag the Association line to the right, and drop it on the "get account balance" use case. An association has now been created between the actor and the use case. The association properties are also visible in the Properties window. The new association has been added to Model Tree under the Relations item of the Use Case View package.



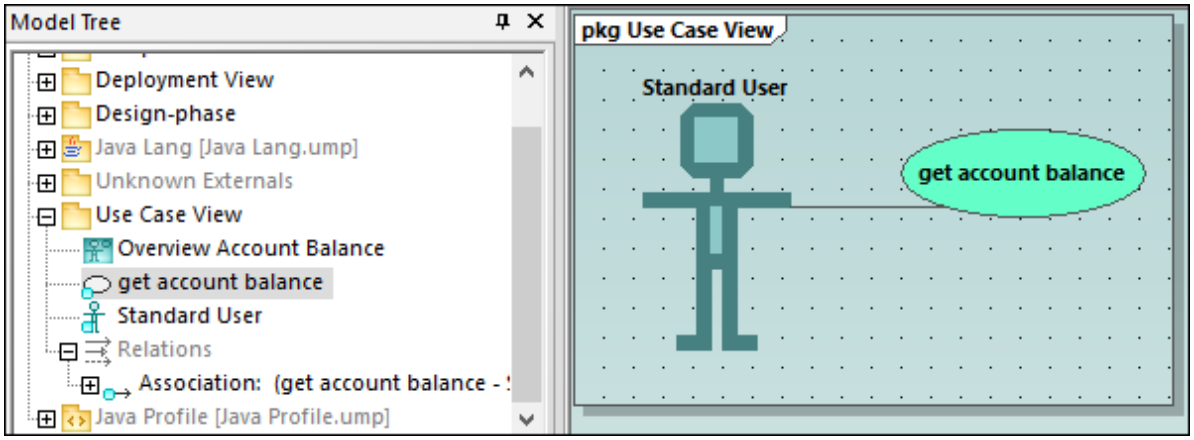
4. Click the use case and drag it to the right to reposition it. The association properties are visible on the association object.
5. Click the use case to select it, then click the **collapse icon** on the left edge of the ellipse.



The "extension points" compartment is now hidden.



A blue dot next to an element in the Model Tree window signifies that the element is visible in the current diagram. For example, in the image below, three elements are currently visible in the diagram and thus have a blue dot in the Model Tree:

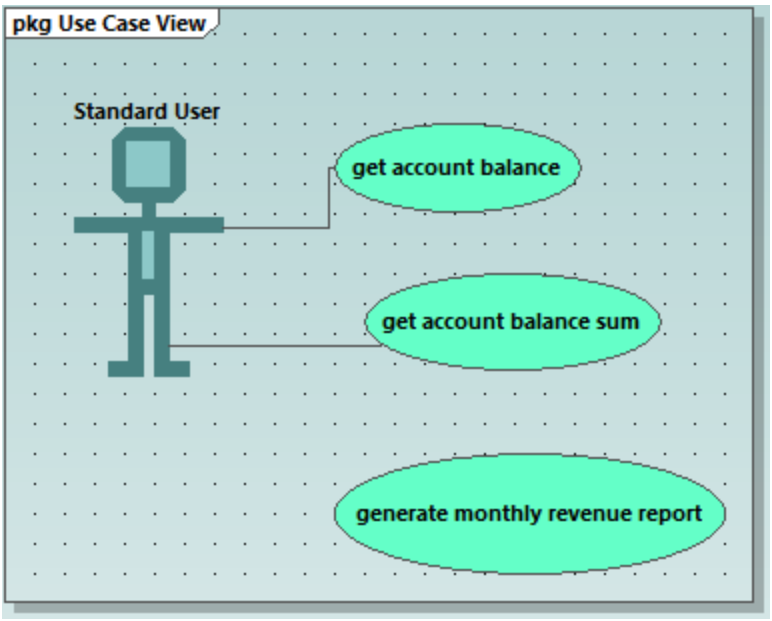


Resizing the actor adjusts the text field, which can also be multi-line. To insert a line break into the text, press **Enter** while holding the **Ctrl** key pressed.

To finish up the Use Case diagram:

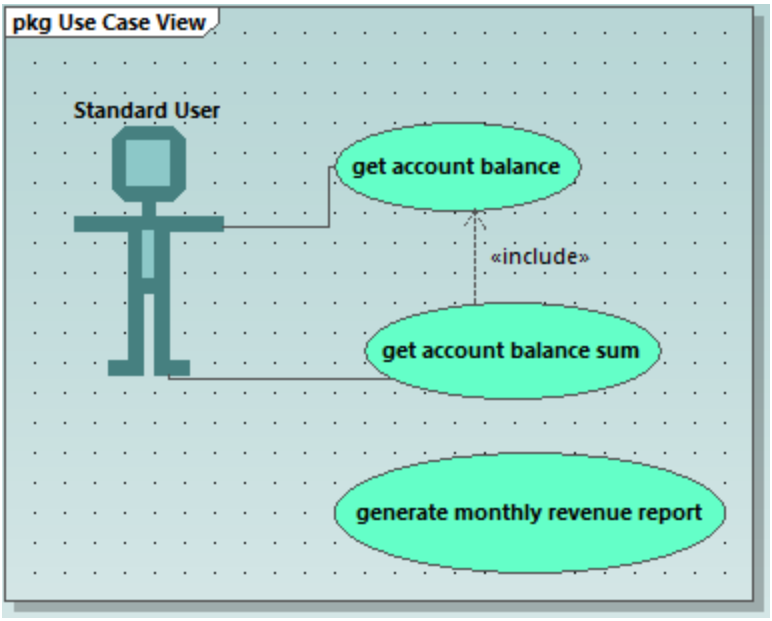
1. Click the **Use Case** toolbar button and simultaneously hold down the **Ctrl** key.
2. Click at two different vertical positions in the diagram to add two more use cases, then release the **Ctrl** key.
3. Name the first use case "get account balance sum" and the second, "generate monthly revenue report".
4. Click the collapse icon of each use case to hide the extensions compartment.
5. Click the actor and use the association handle to create an association between "Standard User" and "get account balance sum".

© 2019-2025 Altova GmbH



To create an "Include" dependency between use cases (creating a subcase):

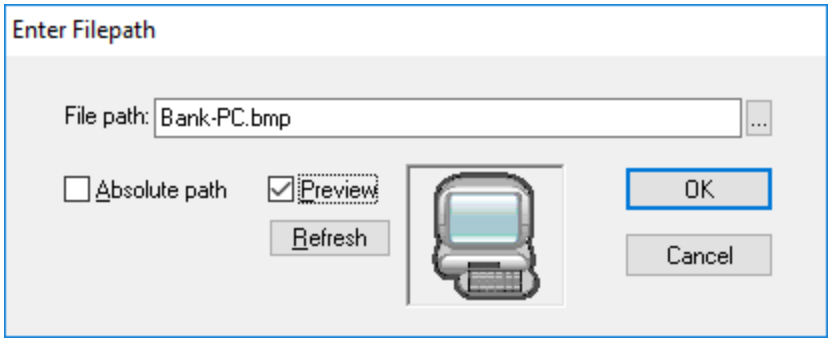
- Click the **Include** handle of the "get account balance sum" use case, at the bottom of the ellipse, and drop the dependency on "get account balance". An "include" dependency is created, and the include stereotype is displayed on the dotted arrow.



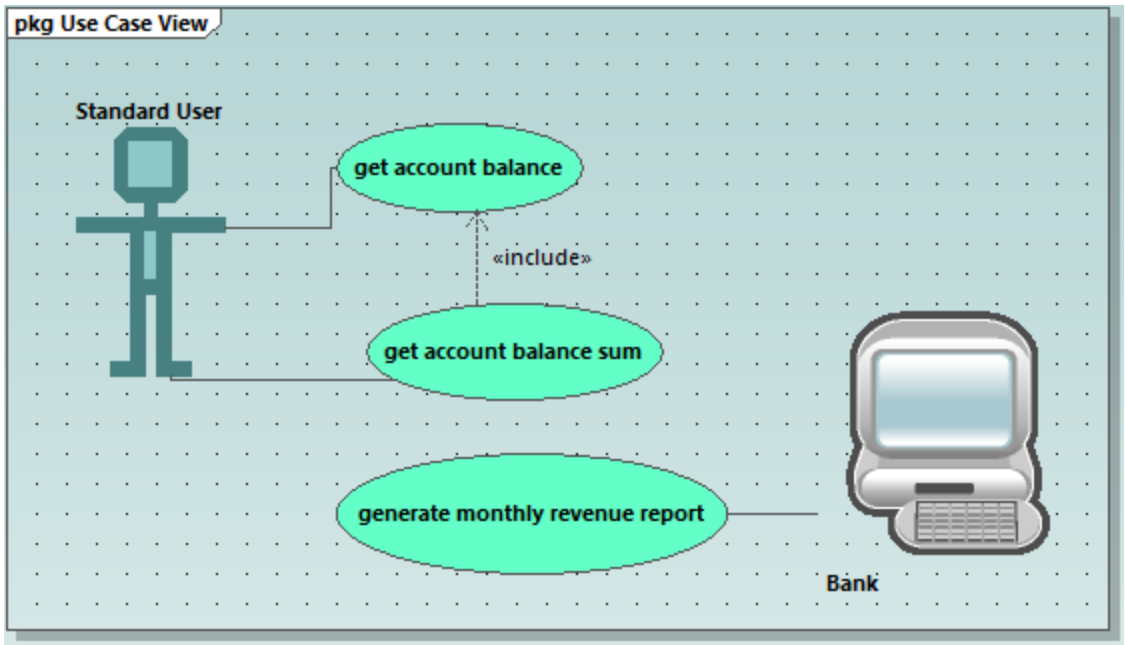
Inserting user-defined (customized) actors

The actor in the "generate monthly revenue report" use case is not a person, but an automated batch job run by a bank computer. The instructions below show to add a new actor to the diagram, and also use a custom image for it.

1. Click the **Actor** toolbar button to insert an actor in the diagram.
2. Rename the actor to "Bank".
3. In the **Properties** window, click **Browse** next to "icon file name" entry, and browse for the **Bank-PC.bmp** file available in the same folder as the project.
4. Clear the **Absolute Path** check box to make the path relative. Select **Preview** to display a preview of the selected file in the dialog box.



5. Click OK to confirm the settings and insert the new actor. Move the new "Bank" actor to the right of the lowest use case.
6. Click the **Association** toolbar button and drag from the "Bank" actor to the "generate monthly revenue report" use case. This is an alternative method of creating an association.



Note: The background color used to make the bitmap transparent has the RGB values 82.82.82.

© 2019-2025 Altova GmbH

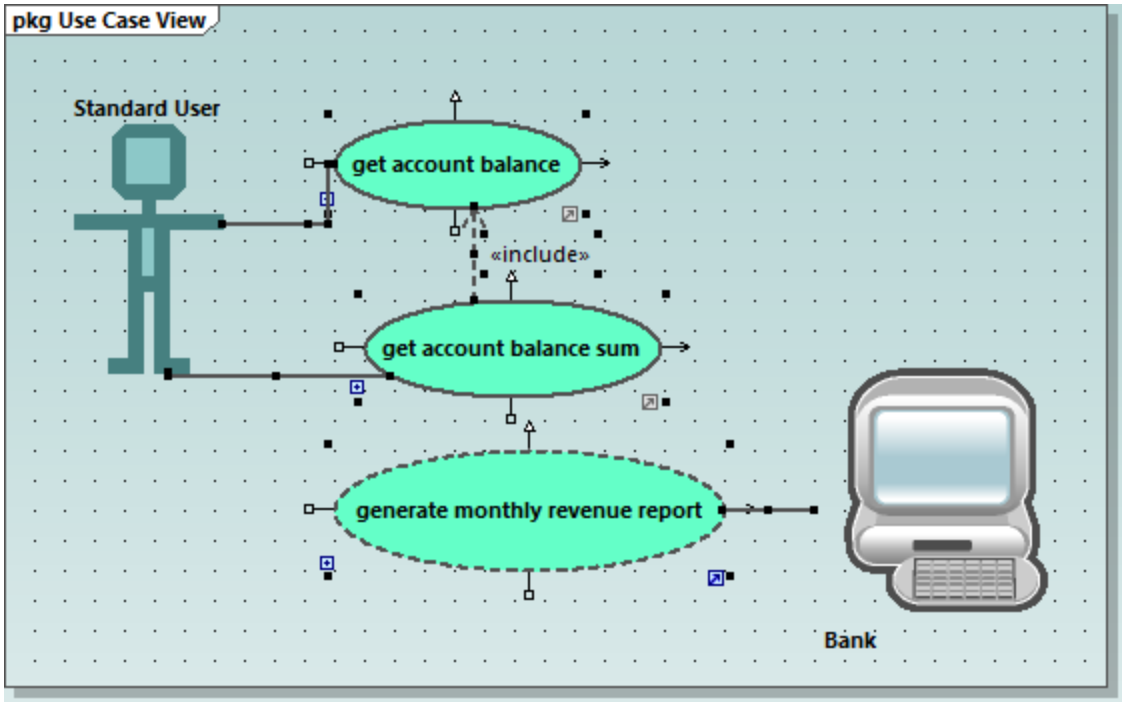
Aligning and adjusting the size of diagram elements

When dragging components in a diagram, guide lines appear allowing you to align an element to any other element in the diagram. You can enable or disable this option as follows:

1. On the **Tools** menu, click **Options**.
2. Click the **View** tab.
3. In the **Alignment** group, select the **Enable snap lines** check box.

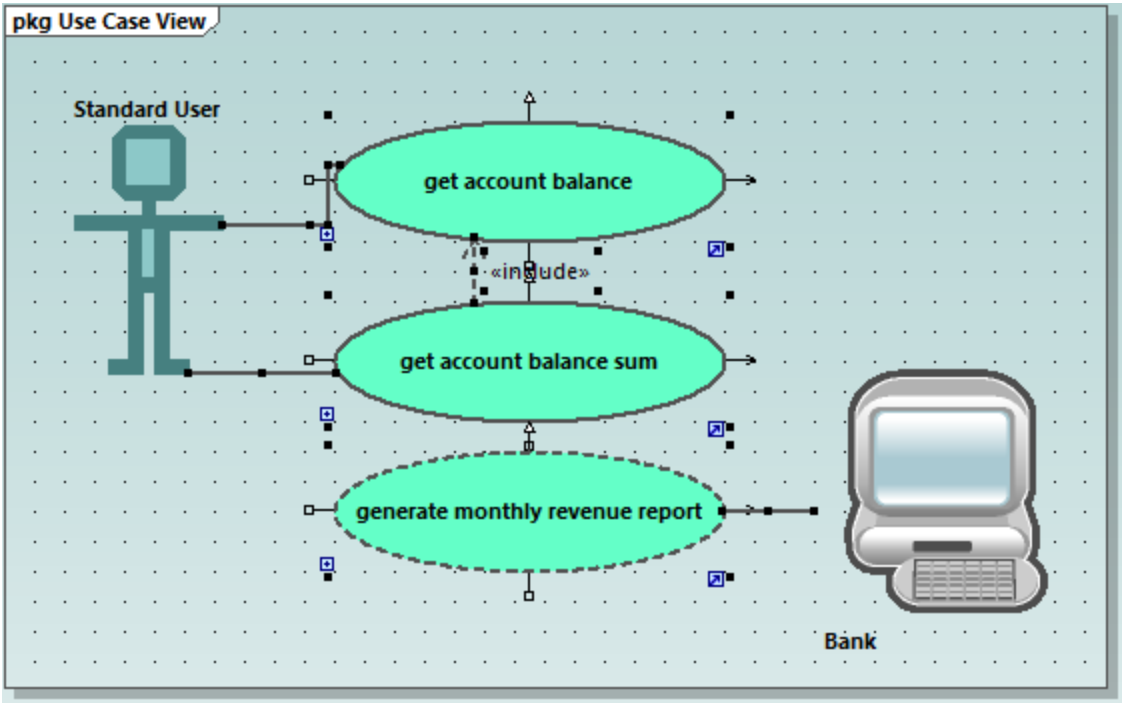
You can also align and adjust the size of multiple elements, as follows:

1. Create a selection marquee by dragging on the diagram background, making sure that you encompass all three use cases starting from the top. Alternatively, to select multiple elements, click elements while holding the **Ctrl** key pressed. Note that the last use case to be marked, is shown in a dashed outline in the diagram, as well as in the Overview window.



   All use cases are selected, with the lowest being the basis for the following adjustments.

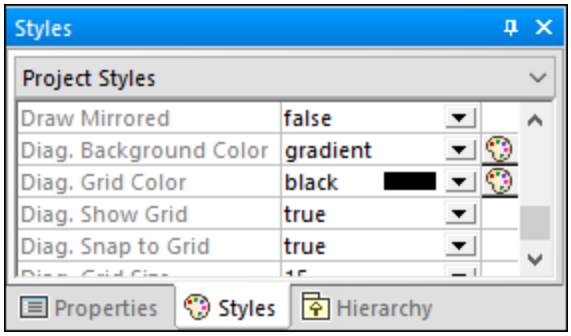
2. Click the **Make same size** toolbar button.
3. To line up all the ovals, click the **Center Horizontally** toolbar button.



Change the style of diagrams in a project

By default, all diagrams of the tutorial project have a gradient background color, and a background grid is also visible. The appearance of diagrams in a project is configurable. For example, to change the background color of all diagrams, do the following:

1. In the **Properties** window, click **Styles**.
2. Under **Project Styles**, identify the setting **Diag. Background Color**.



3. Change the value from "gradient" to a color of your choice.

To enable or disable the diagram background grid:

- Change the setting **Diag. Show Grid** from "true" to "false". (Alternatively, if a diagram is currently open, click the **Show Grid** toolbar button.)

© 2019-2025 Altova GmbH

2.3 Class Diagrams

This tutorial section illustrates the following tasks:

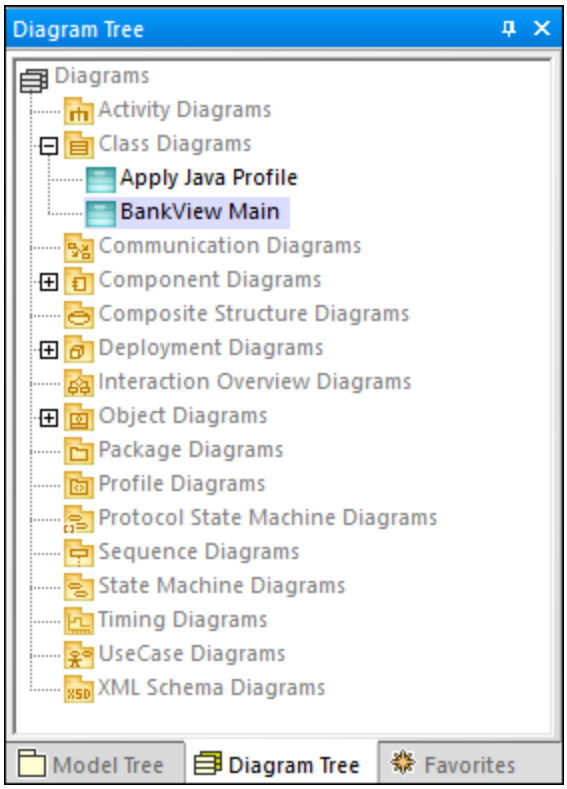
- Add an abstract class to an existing class diagram
- Add class properties and operations, and define parameters as well as their direction and type
- Add a return type to an operation
- Change icons to UML conformant symbols
- Delete and hide class properties and operations
- Create a composite association between two classes.

To proceed, run UModel and open the **BankView-start.ump** project (see also Opening the Tutorial Project 19).

Adding an abstract class

The diagram to which the abstract class will be added is called "BankView Main" and can be opened as follows:

1. In the **Diagram Tree** window, expand the "Class Diagrams" package to display all class diagrams contained in the project.

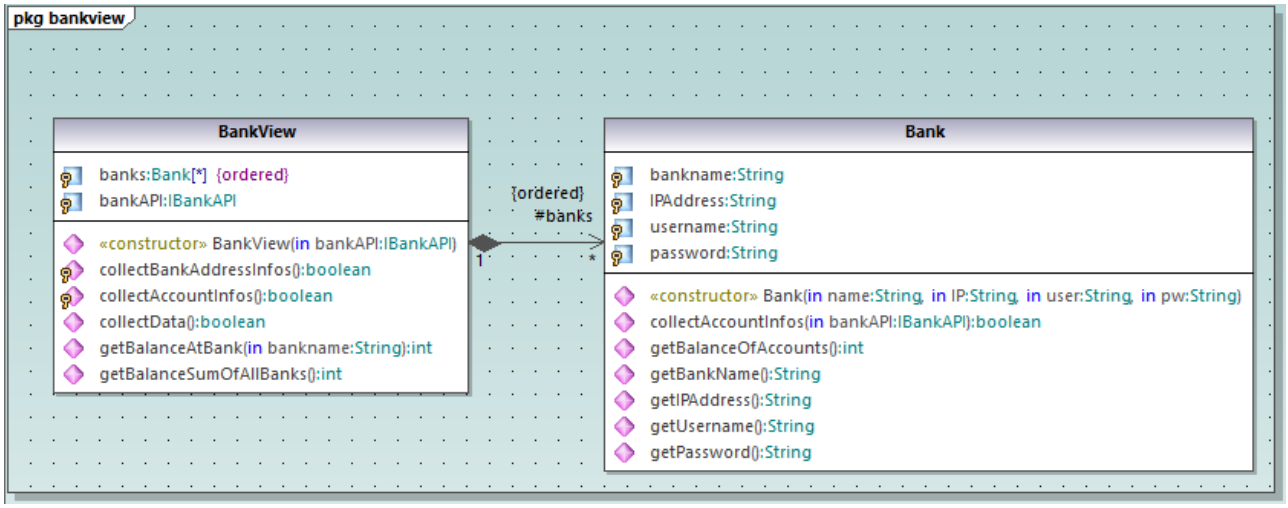


2. Do one of the following:

- Double-click the "BankView Main" diagram icon.
- Right-click the diagram, and select **Open diagram** from the context menu.

Note: It is also possible to open the diagram from the **Model Tree** window. First, locate the diagram under the package "Root | Design-phase | BankView | com | altova | bankview", and then use either of the methods above to open it.

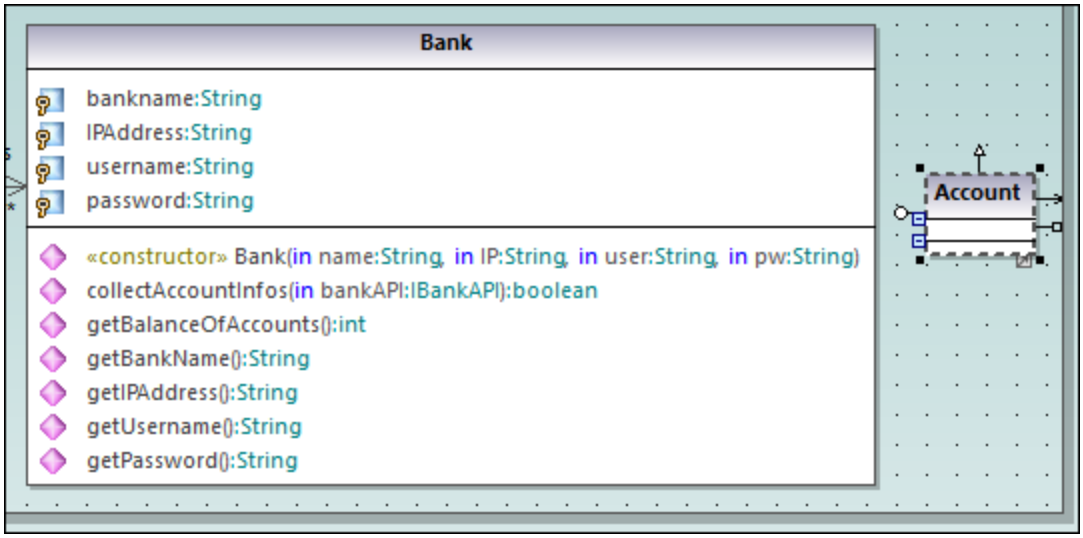
Two concrete classes with a composite association between them are visible in the class diagram.



"Bank View Main" diagram

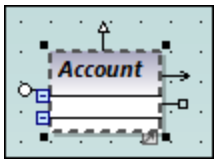
The new abstract class can be added as follows:

1. Click the **Class** toolbar button, and then click to the right of the `Bank` class to insert the new class.
2. Double-click the name of the new class and change it to `Account`.

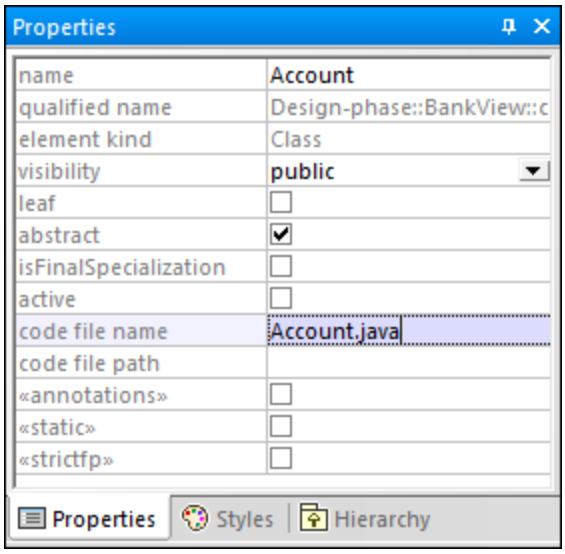


3. In the **Properties** window, select the **abstract** check box to make the class abstract. The class title is now displayed in italic, which is the identifying characteristic of abstract classes.

© 2019-2025 Altova GmbH

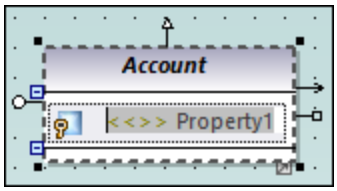


4. In the **code file name** text box, enter "Account.java" to define the Java class.

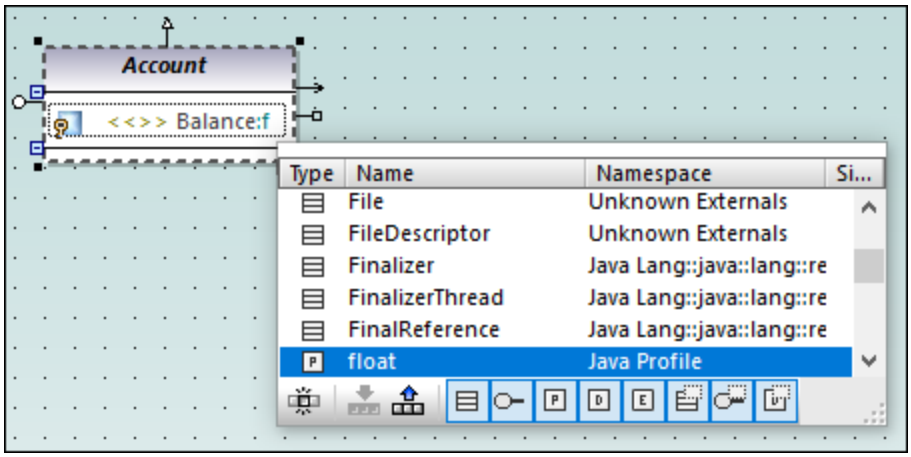


Adding properties to a class

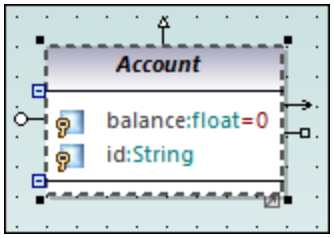
1. Right-click the "Account" class and select **New | Property**, or press **F7**. A default property `Property1` is inserted with stereotype identifiers `<< >>`.



2. Change the property name to `balance`, and then enter a colon (:) character. A drop-down list containing all valid types is displayed.
3. Type "f", and press **Enter** to insert the return type "float". Note that drop-down lists are case sensitive.

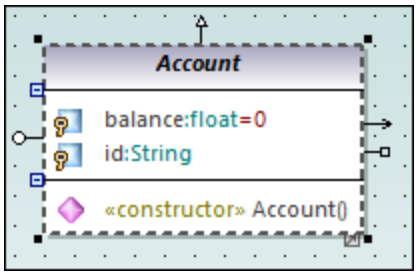


4. Continue on the same line by appending "=0" to define the default value.
5. Using the same method as above, create a new property `id` of type `String`.



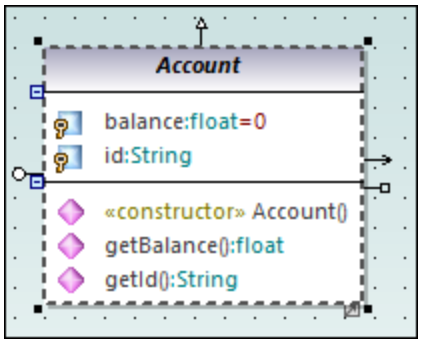
Adding operations to a class

1. Right-click the `Account` class and select **New | Operation**, or press **F8**.
2. Enter "Account()" as operation name. Notice that the stereotype has changed to <<`constructor`>>, since the operation name is the same as the class name.



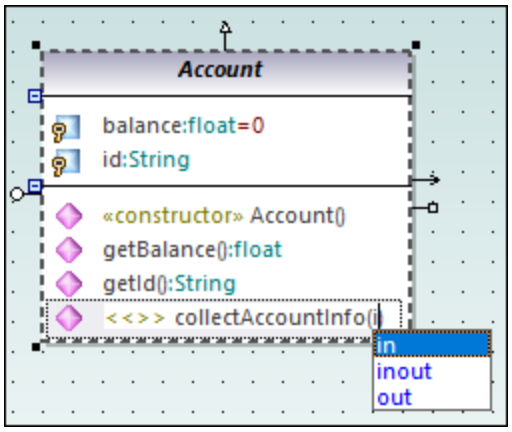
3. Using the same method as above, add two more operations, namely, `getBalance():float` and `getId():String`.

© 2019-2025 Altova GmbH

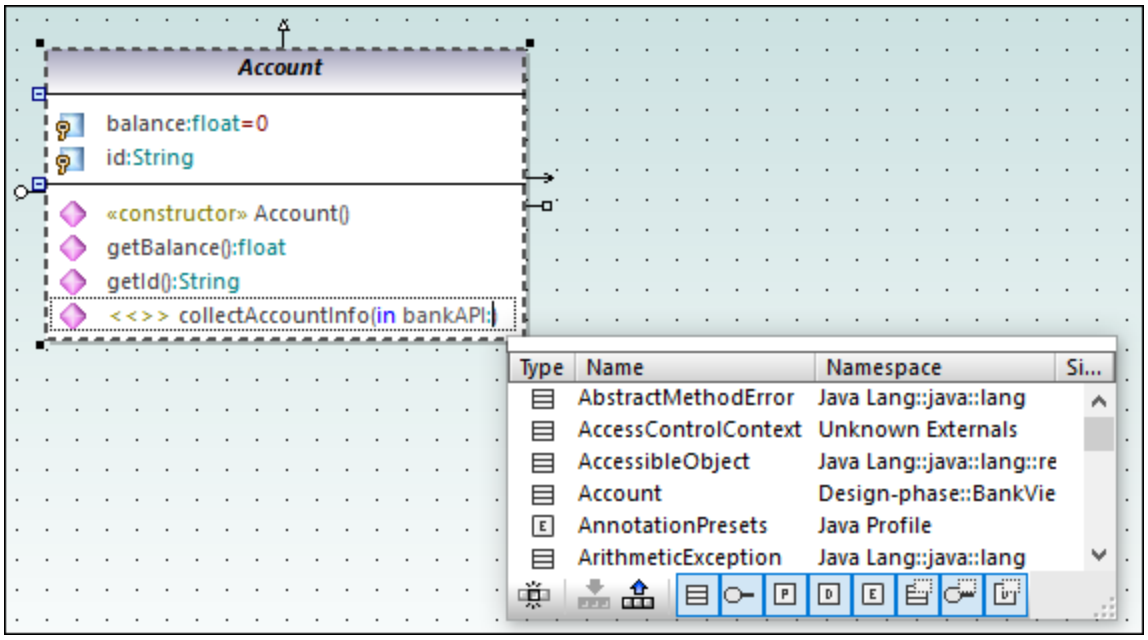


Let's now add a new operation which takes a parameter. We will also specify the parameter direction and type.

1. Press **F8** to create another operation, `collectAccountInfo()`.
2. Place the mouse cursor within the brackets and start typing "i". A drop-down list opens, allowing you to select the parameter direction: `in`, `inout`, or `out`.



3. Select "in" from the drop-down list, enter a space, and continue editing on the same line.
4. Enter "bankAPI" as parameter name and then a colon (:). A drop-down list opens, allowing you to select the parameter type.

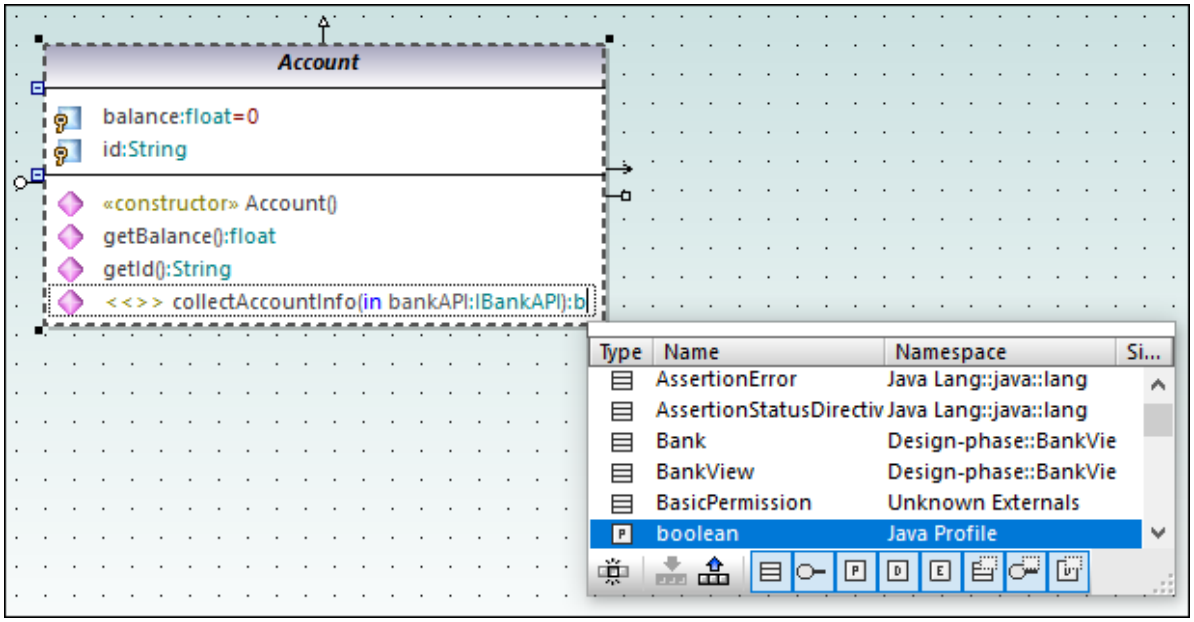


5. Select **IBankAPI** from the drop-down list.

Adding a return type to an operation

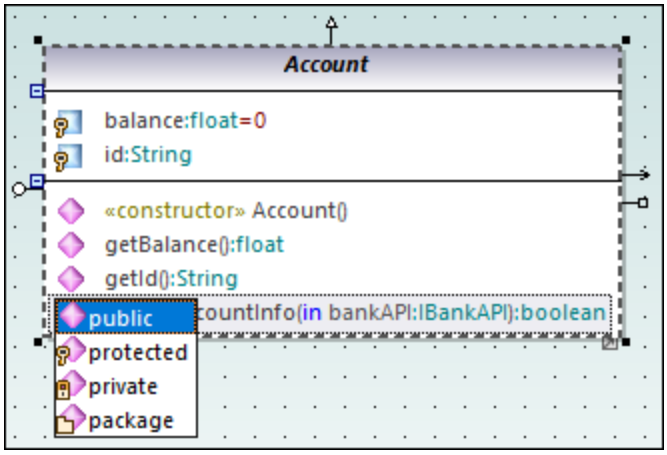
So far, the operation parameter has been added, but it does not have a return type yet. To add a return type:

1. Place the mouse cursor after the close parenthesis character ")" and enter a colon (:). A drop-down list opens, allowing you to select a return type.
2. Press the "b" key and select `boolean` as data type.



 © 2019-2025 Altova GmbH

To specify an operation's visibility (for example, "private", "protected", "public"), click the icon preceding the operation name, and select the required value, for example:



The visibility "package" is applicable for Java. In C#, use "package" to specify visibility as "internal". For information about how UModel elements map to constructs in each language, see UModel Element Mappings 224.

Changing icons to UML conformant symbols

The visibility icons can be changed to UML conformant symbols if necessary, as follows:

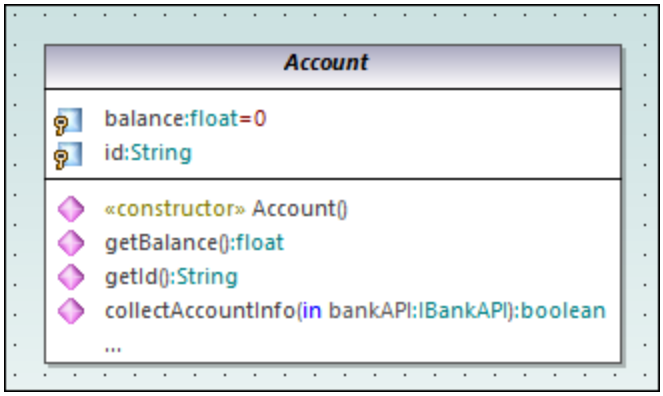
1. In the **Styles** window, select **Project Styles** from the top drop-down list.
2. Scroll down to the **Show Visibility** setting, and select **UML Style**.

Deleting and hiding class properties and operations from a Class diagram

Press **F8** to add a dummy operation `Operation1` to the `Account` class.

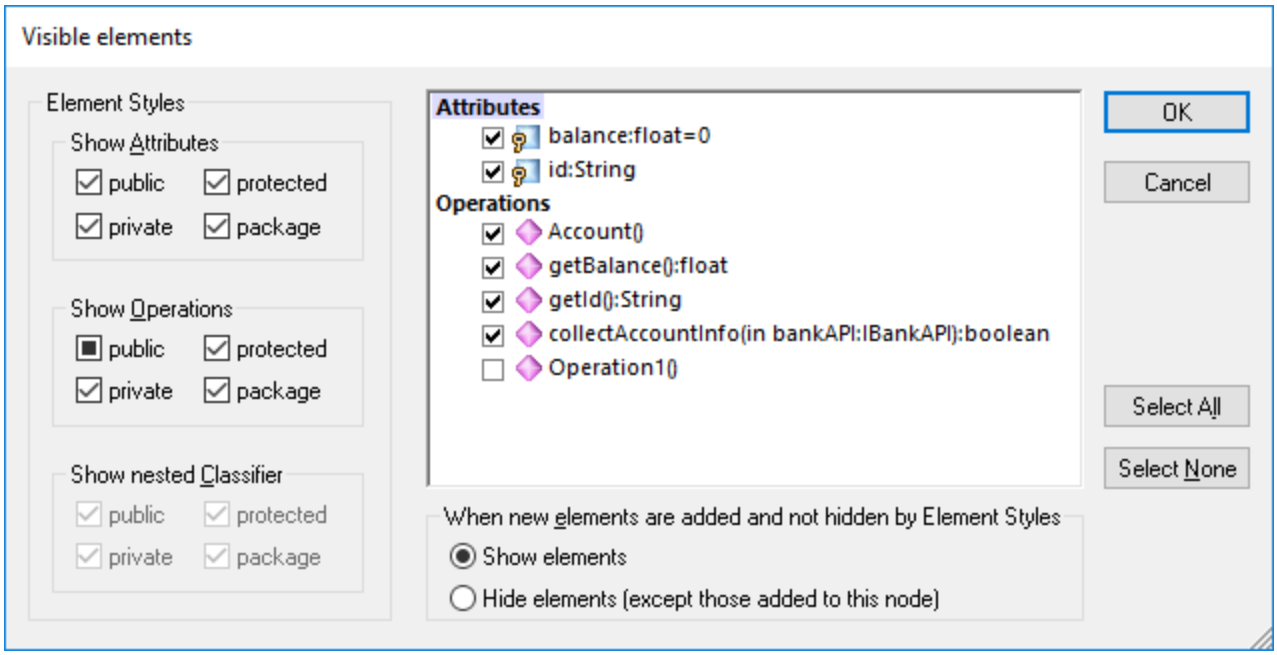
To delete the dummy operation, select it and then press **Delete**. (Alternatively, right-click it and select **Delete** from the context menu). A message box appears asking if you want to delete the element from the project. Click **Yes** to delete `Operation1` from the class diagram as well as from the project.

To delete the operation from the class in the diagram, but not from the project, press the **Ctrl+Delete**. This hides the operation from the diagram, although it continues to exist in the project. Classes with hidden members are displayed with an ellipsis (...) character, as shown below:



A class with hidden operations

To unhide the operation, double-click the ellipsis at the bottom of the class. A dialog box appears where you can choose the elements that should be visible on the diagram, for example:



"Visible elements" dialog box

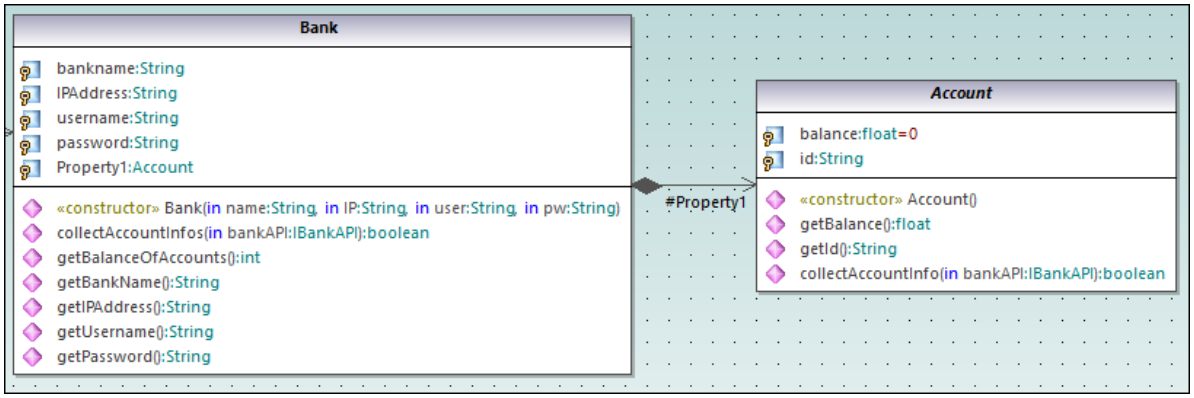
It is possible to configure UModel not to display a message box when you attempt to delete an object from the diagram, as follows:

1. On the **Tools** menu, click **Options**.
2. Click the **Editing** tab.
3. Under **Ask before deleting from project**, clear the **in diagrams** check box.

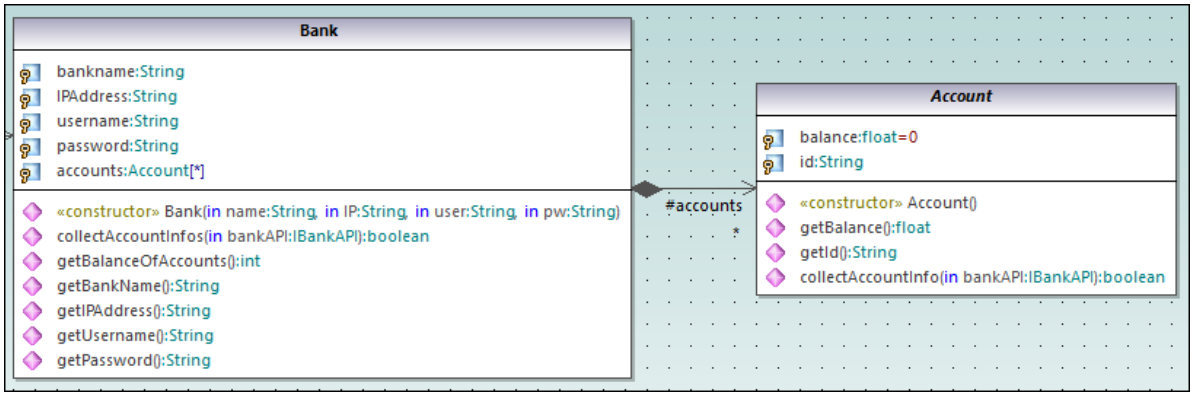
© 2019-2025 Altova GmbH

Creating a composition association between the Bank and Account classes

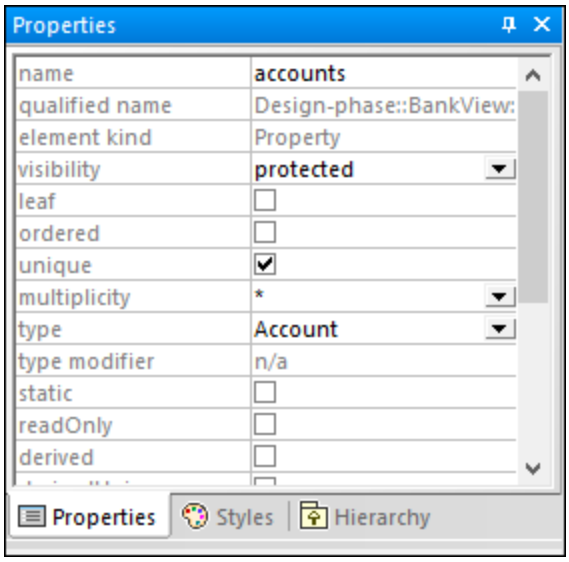
1. Click the **Composition** toolbar button, and then drag from the `Bank` class to the `Account` class. The class is highlighted when the association can be made. A new property (`Property1:Account`) is created in the `Bank` class, and a composite association arrow joins the two classes.



2. Double click the new `Property1` property in the `Bank` class and change it to "accounts", being sure not to delete the `Account` type definition (displayed in teal/green).
3. Press the **End** keyboard key to place the text cursor at the end of the line.
4. Enter the open square bracket character ([ ) and select asterisk (*) from the dropdown list. This defines the *multiplicity*, namely, the fact that a bank can have many accounts.



Notice that the multiplicity range previously added to the diagram is also visible in the **Properties** window:



2.3.1 Creating Derived Classes

This tutorial section illustrates the following tasks:

- Add a new class diagram to the project
- Add existing classes to a diagram
- Add a new class to a diagram
- Create derived classes from an abstract class, using generalizations.

Note: It is assumed you have already followed the previous tutorial section, Class Diagrams 31, to create the abstract class `Account`.

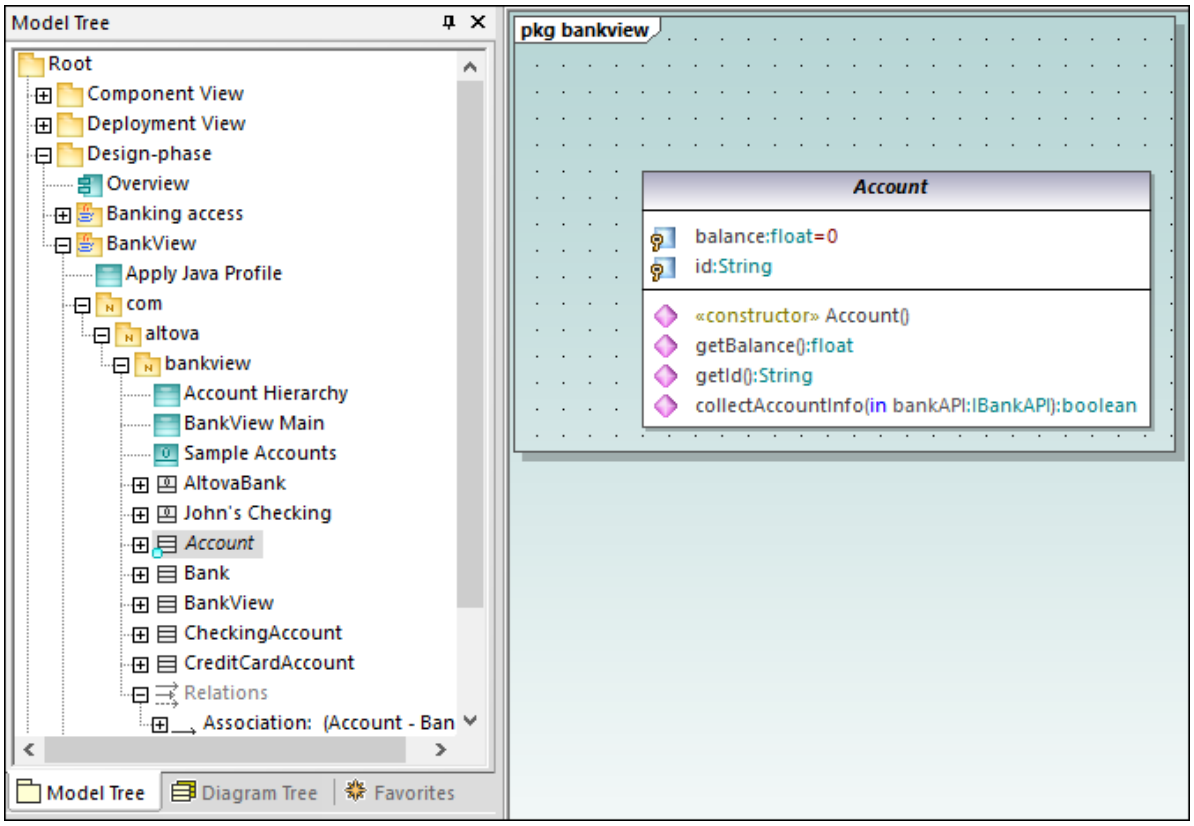
Creating a new Class Diagram

1. In the **Model Tree** window, right-click the `bankview` package (under **Root | Design-phase | BankView | com | altova**), and select **New Diagram | Class Diagram**.
2. Double-click the new "ClassDiagram1" entry, rename it to "Account Hierarchy", and press **Enter** to confirm. The new "Account Hierarchy" diagram is now visible in the working area.

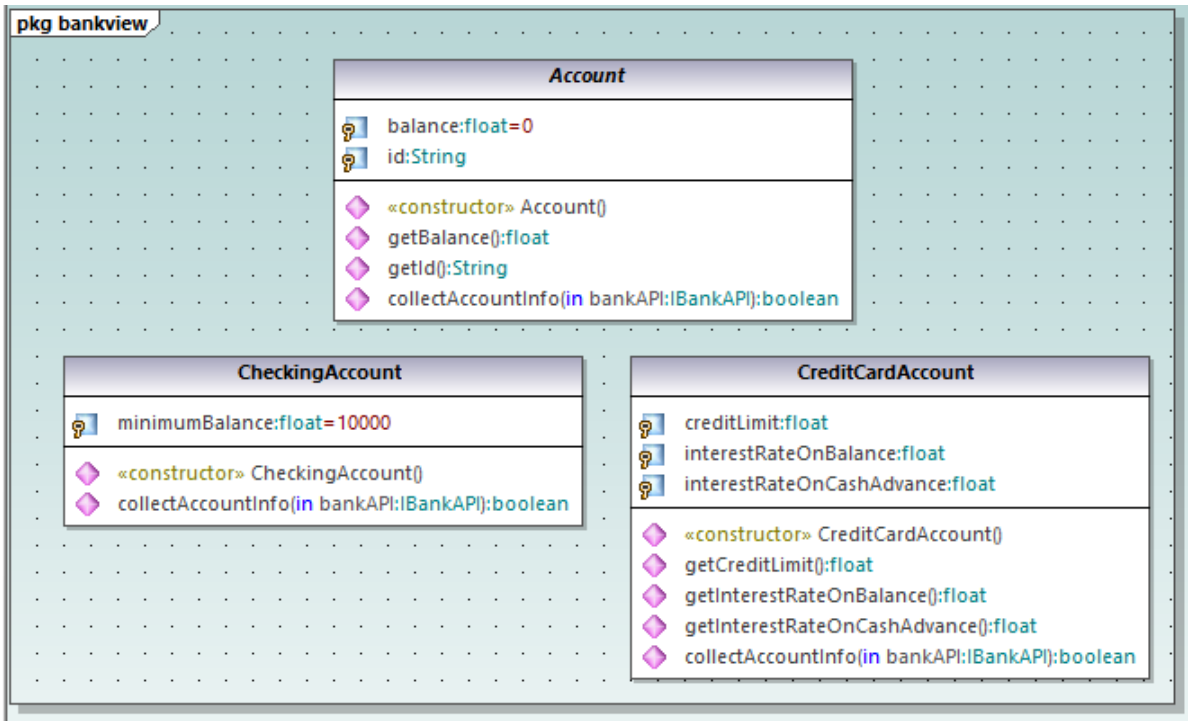
Adding existing classes to a diagram

1. In the **Model Tree** window, click the `Account` class in the `bankview` package (under **com | altova | bankview**), and drag it into the diagram.

© 2019-2025 Altova GmbH



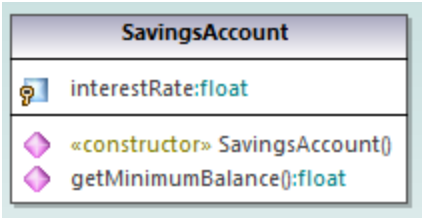
2. Click the `CheckingAccount` class (of the same package) and drag it into the diagram. Place the class below and to the left of the `Account` class.
3. Use the same method to insert the `CreditCardAccount` class. Place it to the right of the `CheckingAccount` class.



Adding a new class

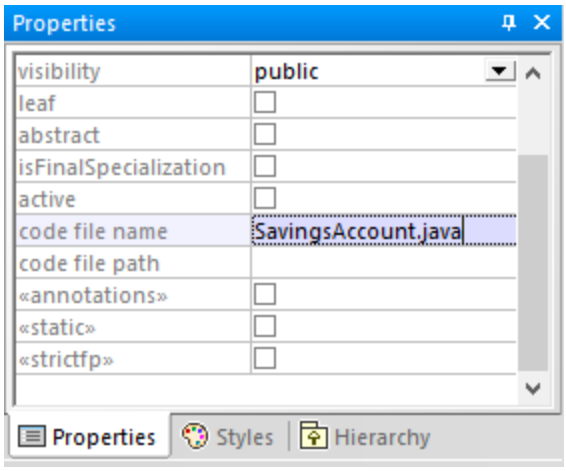
The third derived class, `SavingsAccount`, will be added manually to the diagram.

1. Right-click the diagram and select **New | Class**. A new class is automatically added to the correct package (`bankview`) which contains the current class diagram "Account Hierarchy".
2. Double-click the class name and change it to `SavingsAccount`.
3. Create the class structure as illustrated below. To add properties and operations, use the methods illustrated in the previous tutorial section, Class Diagrams 31.



3. In the **Properties** window, in the "code file name" text box, enter "SavingsAccount.java" to define the Java code class.

© 2019-2025 Altova GmbH



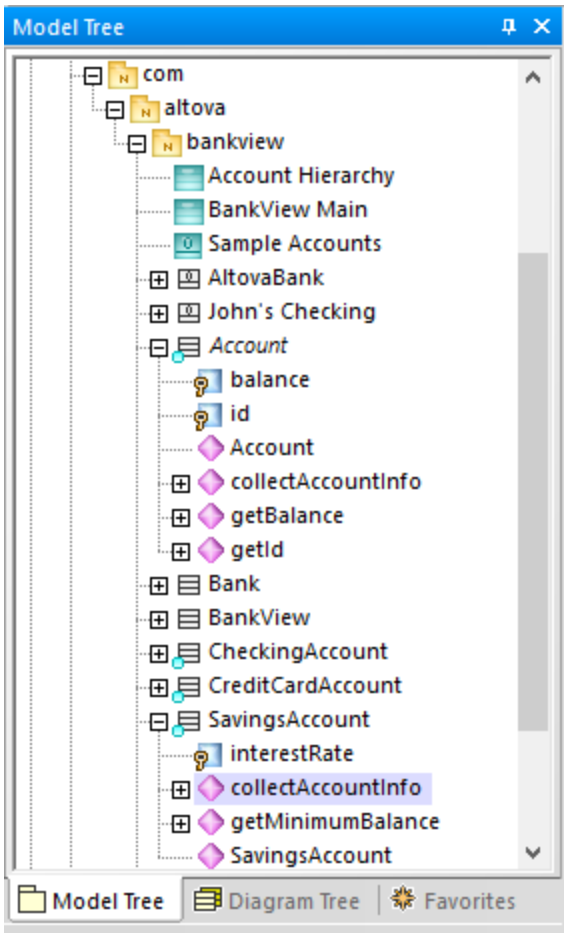
Properties and operations can be directly copied or moved from one class to another:

- Within a class in the current diagram
- Between different classes of the same diagram
- In the **Model Tree** window
- Between different UML diagrams, by dropping the copied data onto a different diagram.

This can be achieved using drag and drop, as well as the standard **Copy/Paste** keyboard shortcuts (**Ctrl + C**, **Ctrl + V**), see also Renaming, Moving, and Copying Elements 112. For the scope of this example, you can quickly copy the `collectAccountInfo()` operation from the `Account` class to the new `SavingsAccount` class, as follows:

1. In the **Model Tree** window, expand the `Account` class.
2. Right-click the `collectAccountInfo` operation and select **Copy**.
3. Right-click the `SavingsAccount` class and select **Paste**.

The operation is copied into the `SavingsAccount` class, which is automatically expanded to display the new operation.

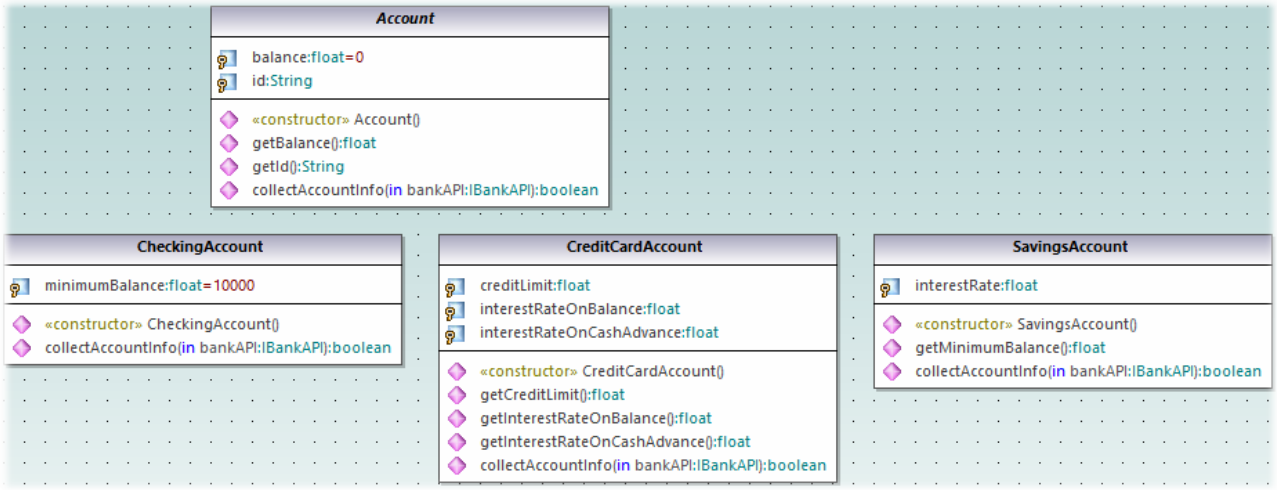


The new operation is now also visible in the `SavingsAccount` class in the class diagram.

Creating derived classes using generalization/specialization

At this point, the class diagram contains the abstract class, `Account`, as well as three specific classes.

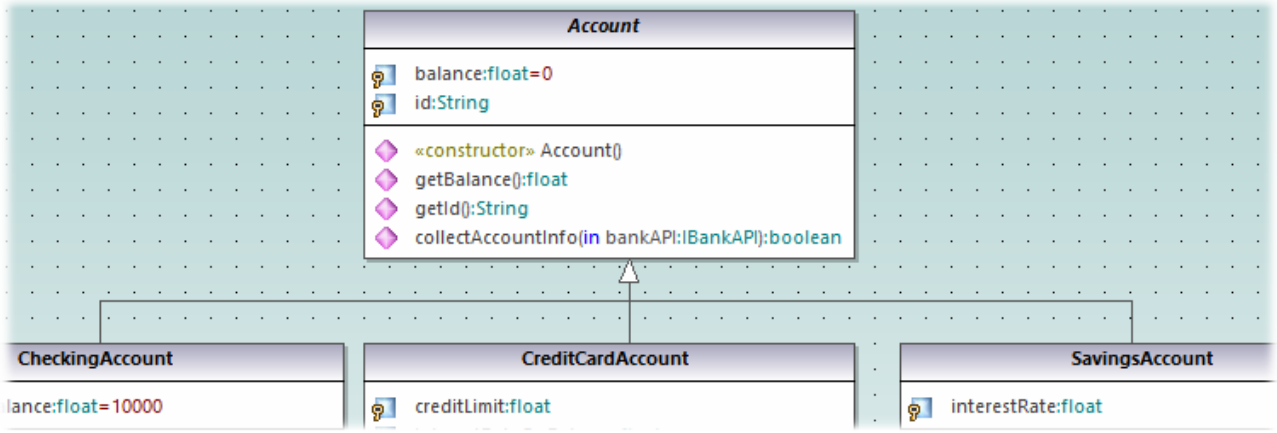
© 2019-2025 Altova GmbH



We will now create a generalization/specialization relationship between `Account` and the specific classes (that is, create three derived concrete classes).

1. Click the **Generalization** toolbar button and hold down the **Ctrl** key.
2. Drag from `CreditCardAccount` class and drop on the `Account` class.
3. Drag from the `CheckingAccount` class and drop on the *arrowhead* of the previously created generalization.
4. Drag from the `SavingsAccount` class and drop on the *arrowhead* of the previously created generalization: release the **Ctrl** key at this point.

Generalization arrows are created between the three subclasses and the `Account` superclass.



2.4 Object Diagrams

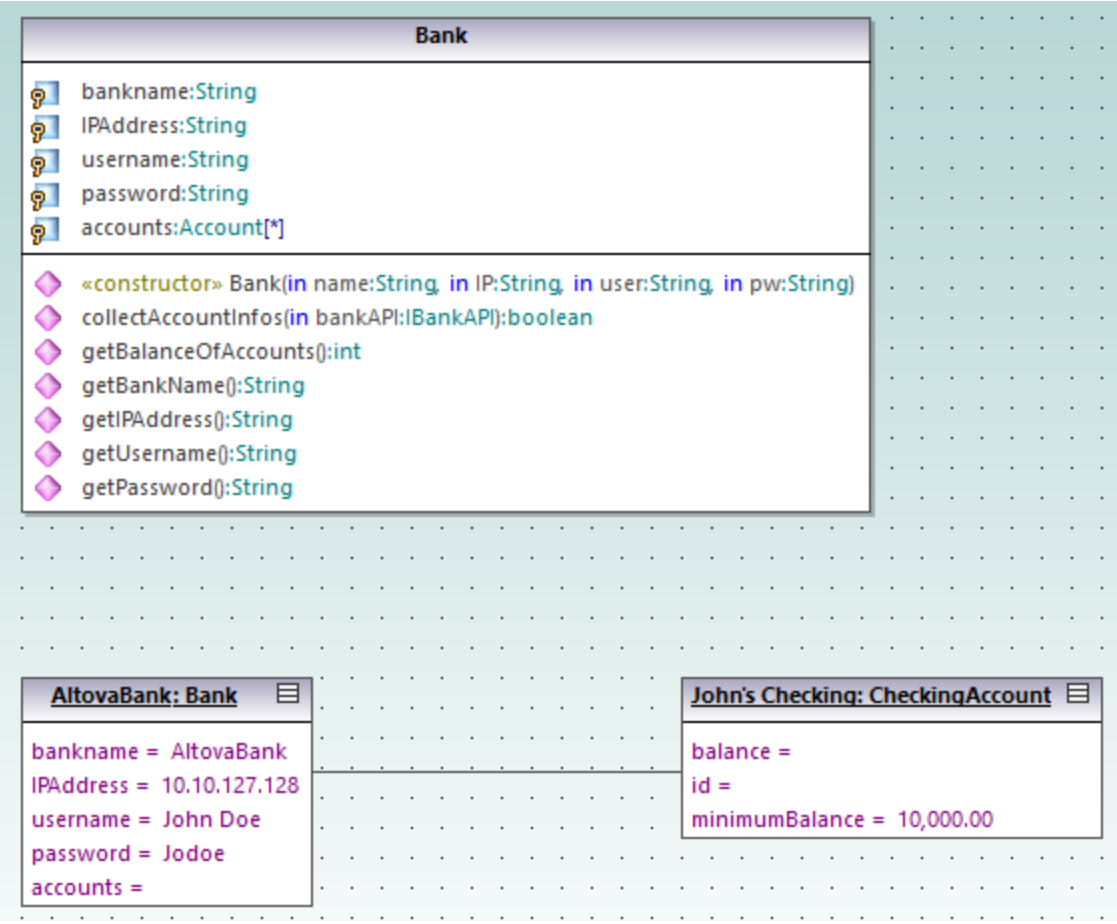
This tutorial section illustrates the following tasks:

- Combine class and object diagrams into one diagram
- Create objects/instances and define the relationships between them
- Format association/links
- Enter real-life data into objects/instances

To proceed, run UModel and open the **BankView-start.ump** project (see also Opening the Tutorial Project 19). The project includes a predefined object diagram "Sample Accounts", which will be used to illustrate the tasks above.

Combining objects and classes into one diagram

In the **Model Tree** window, navigate to the following path: **Root | Design-phase | BankView | com | altova | bankview**. Then double-click the icon next to the "Sample Accounts" diagram.

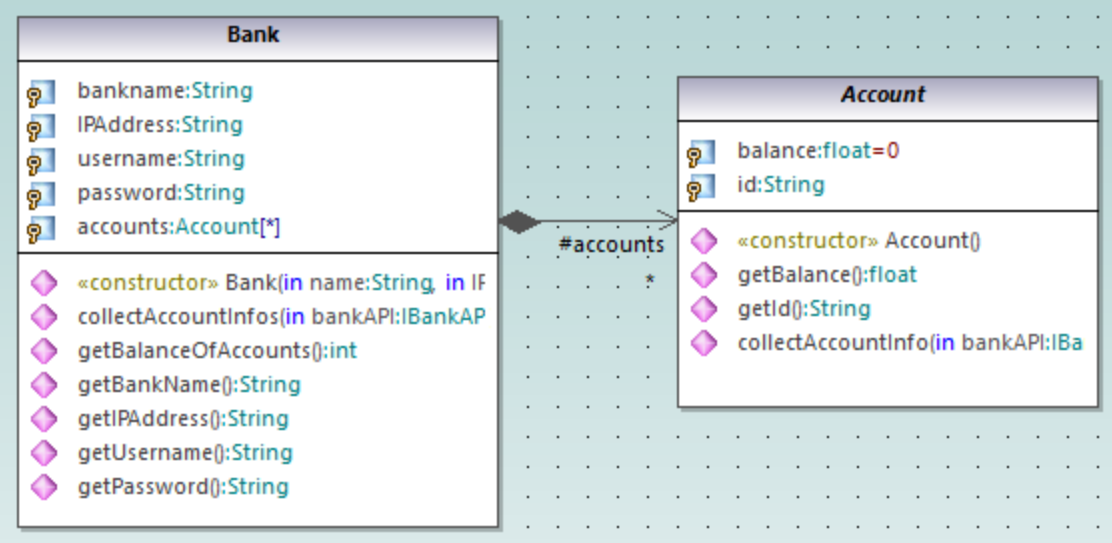


"Sample Accounts" diagram

© 2019-2025 Altova GmbH

This object diagram combines both classes and instances of them (objects). Specifically, `AltovaBank:Bank` is the object/instance of the `Bank` class, while `John's checking: CheckingAccount` is an instance of the class `CheckingAccount` class (not yet added to the diagram).

Let's now add the missing `Account` class to the diagram, by dragging it from the **Model Tree** into the diagram. Notice that the composite association between `Bank` and `Account` is displayed automatically (this association was defined in one of the previous tutorial sections, see Class Diagrams 31).

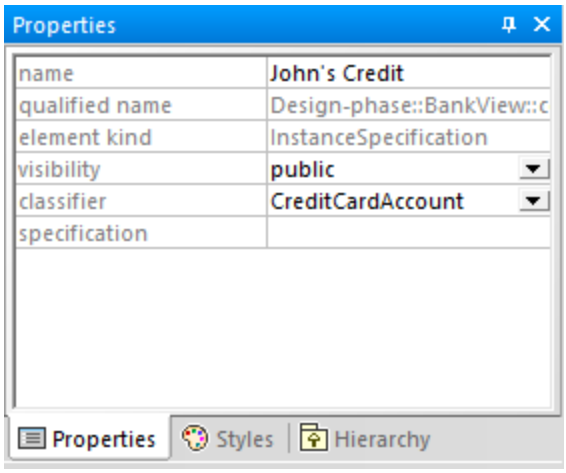


Adding a new object/instance (Approach 1)

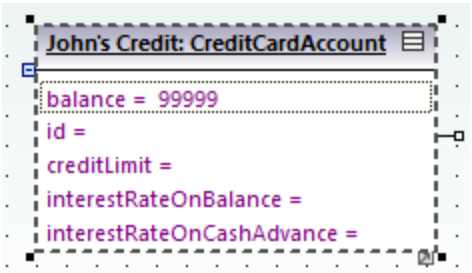
Let's now add a new object to the diagram, called `John's Credit`. This object will instantiate the `CreditCardAccount` class.

1. Click the **InstanceSpecification** toolbar button, and then click inside the diagram, below the object `John's Checking: Checking Account`.
2. Change the name of the new instance to `John's Credit`, and press **Enter**.

3. Select the new instance to display its properties in the **Properties** window.
4. In the **Properties** window, next to "classifier", select **CreditCardAccount** from the drop-down list.



The instance has now changed appearance to display all properties of the class. Double-click any property to enter a value, for example:



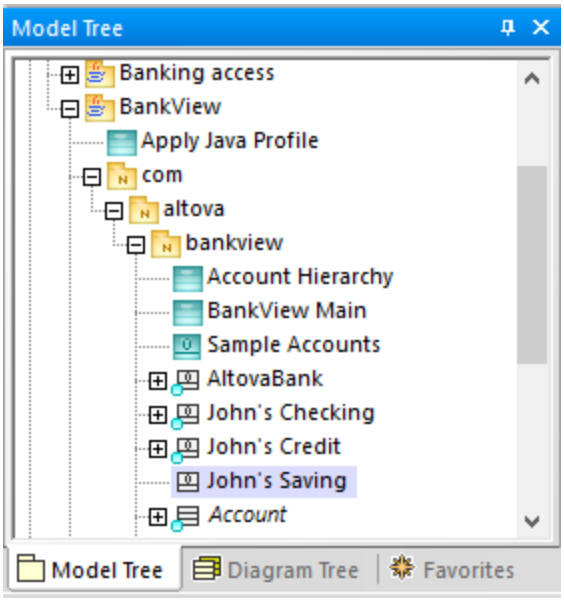
To show or hide specific nodes, right-click the instance and select **Show/hide node content (Ctrl+Shift+H)** from the context menu.

Adding a new object/instance (Approach 2)

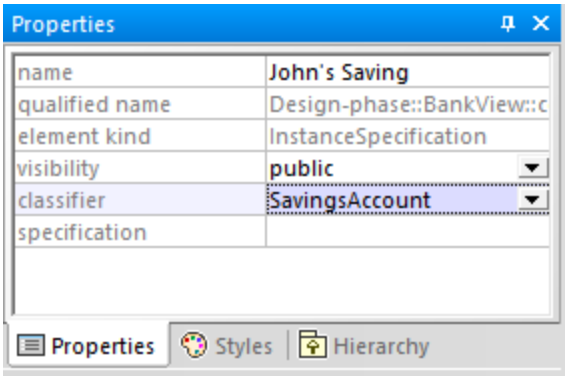
We will now add a new instance of the class `SavingsAccount`, this time using a different approach:

1. In the **Model Tree** window, right-click the `bankview` package, and select **New element | InstanceSpecification**.
2. Rename the new instance to `John's Saving`, and press **Enter** to confirm. The new object is added to the package and sorted accordingly.

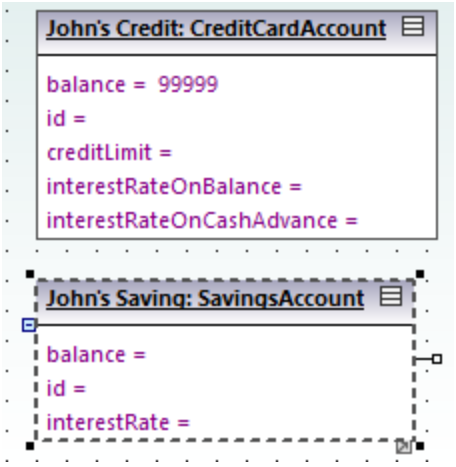
© 2019-2025 Altova GmbH



3. While the object is still selected in the **Model Tree** window, select **SavingsAccount** next to "classifier" in the **Properties** window.



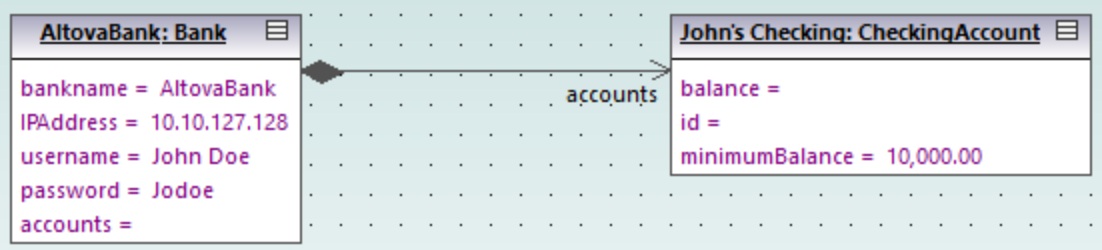
4. Drag the object `John's Saving` from the **Model Tree** window into the diagram, placing it below the object `John's Credit`.



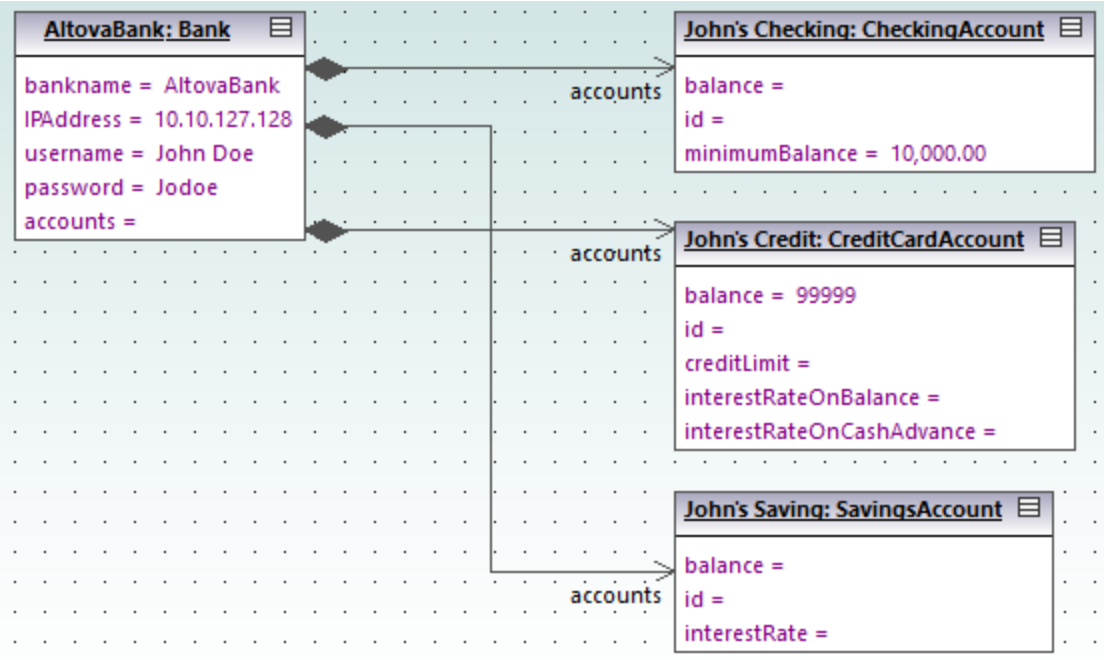
Creating links between objects

Links are the instances of class associations, and describe the relationships between objects/instances at a fixed moment in time.

1. Click the existing link (association) between the object `AltovaBank: Bank` and the object `John's Checking: CheckingAccount`.
2. In the **Properties** window, next to "classifier", select the entry **Account - Bank**. The link now changes to a composite association, in accordance with the class definitions.



3. Click the **InstanceSpecification** toolbar button, and position the cursor over the object `John's Credit: CreditAccount`. The cursor now appears as a + sign.
4. Drag from the object `John's Credit: CreditAccount` to `AltovaBank: Bank` to create a link between the two.
5. In the **Properties** window, next to "classifier", select the entry **Account - Bank**.
6. Finally, using the methods outlined above, create a link between the object `AltovaBank: Bank` and the object `John's Saving: SavingsAccount`.

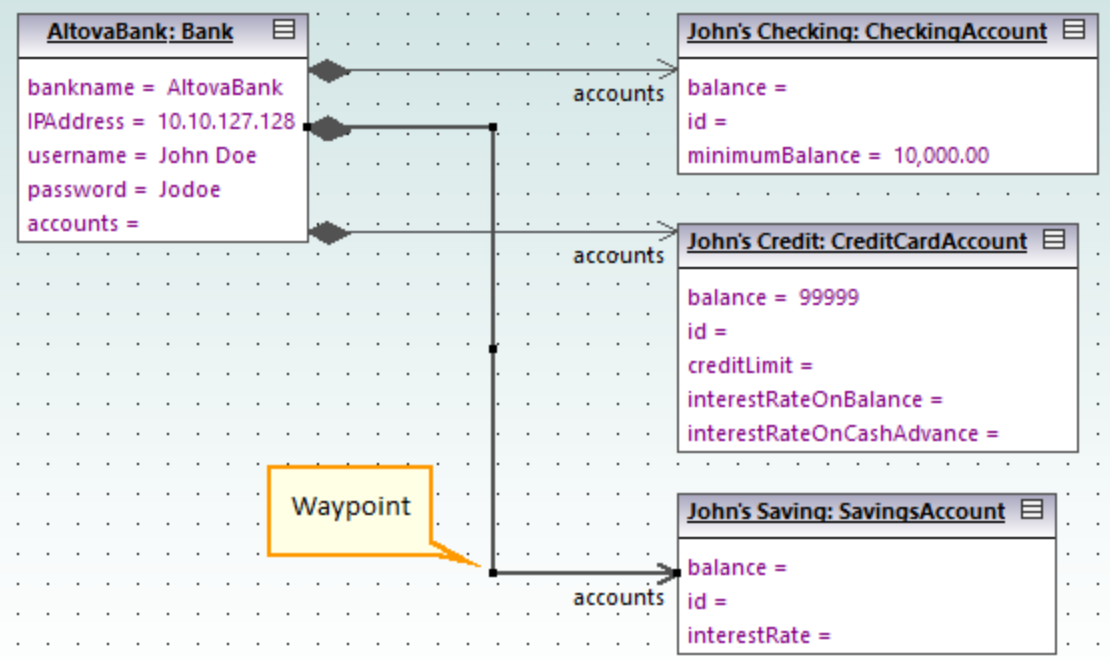


Note that changes made to the association type in any class diagram are automatically updated in the object diagram.

© 2019-2025 Altova GmbH

Formatting association/link lines in a diagram

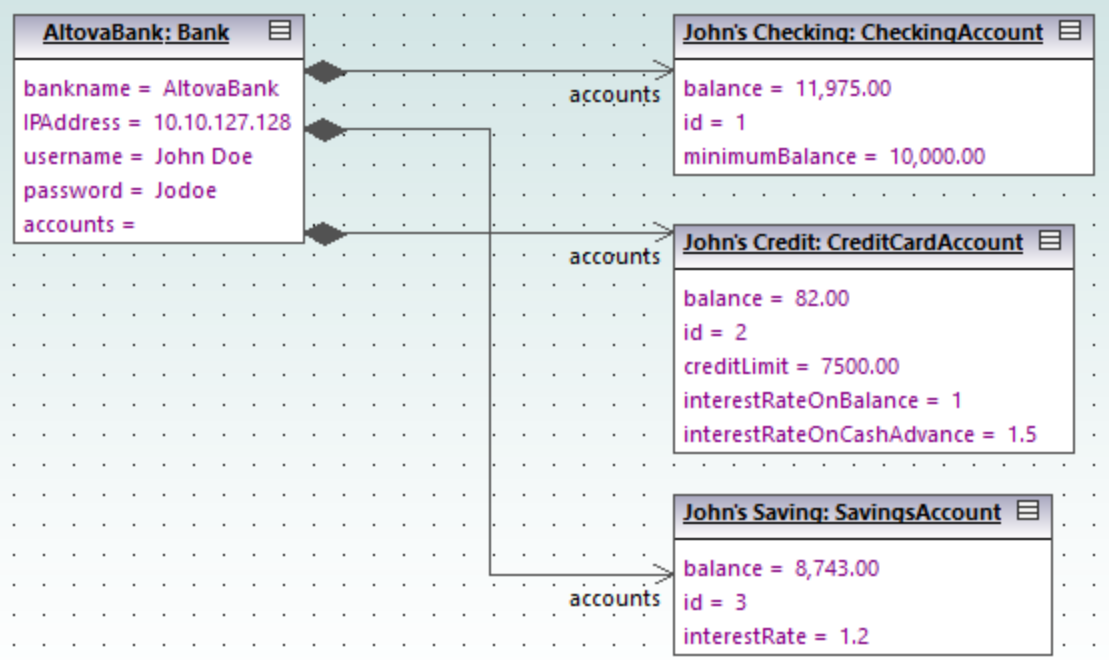
To format links between objects, place the cursor on the line and drag to the desired position. To reposition the line both horizontally and vertically, drag the corner waypoint, as illustrated below.



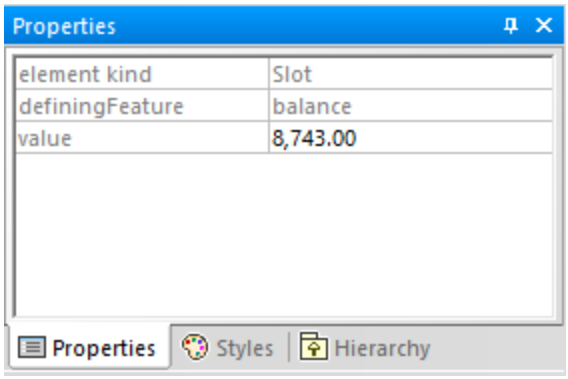
Links in an object diagram

Entering sample data into objects

The instance value of an attribute/property in an object is called a *slot*. To describe the state of an object, double-click the slots and enter sample instance data after the "=" character, for example:



Object slots can also be filled from the **Properties** window, by selecting the object and entering the appropriate text next to "value", for example:



© 2019-2025 Altova GmbH

2.5 Component Diagrams

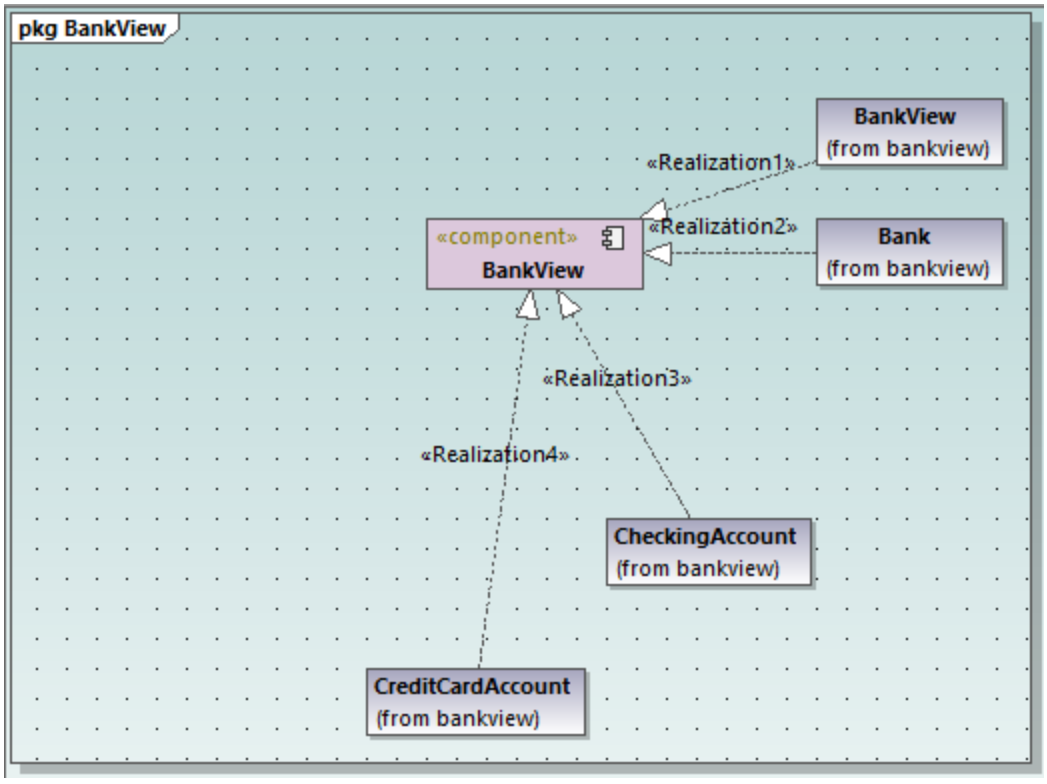
This tutorial section illustrates the following tasks:

- Create realization dependencies between classes and components
- Change the appearance of lines used in the diagram
- Add usage dependencies to an interface
- Use "ball-and-socket" interface notation

To proceed, run UModel and open the **BankView-start.ump** project (see also Opening the Tutorial Project 19). The project includes several predefined object diagrams which will be used to illustrate the tasks above. It is assumed you have already followed the tutorial section Creating Derived Classes 40 to create the class `SavingsAccount`.

Creating realization dependencies between classes and components

In the **Diagram Tree** window, expand "Component Diagrams", and double-click the "BankView realization" diagram icon. This diagram already contains the `BankView` component and several classes connected to it with dependencies of type "ComponentRealization". The text "from bankview" inside each class indicates the name of the package where the class belongs.

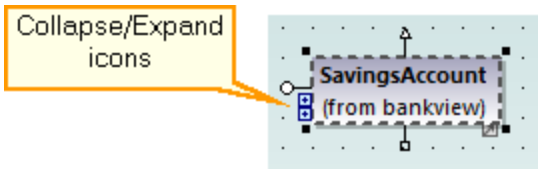


"Bank View realization" diagram

Let's now add a new class to the diagram and also create a realization dependency between the new class and the `BankView` component.

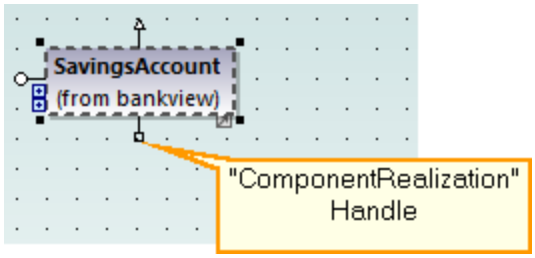
1. In the **Model Tree** window, locate the `SavingsAccount` class in the `bankview` package. If this class is missing, follow the tutorial section Creating Derived Classes 40 to create it first.
2. Drag the `SavingsAccount` class from the **Model Tree** into the diagram.

By default, the class is displayed with all compartments expanded. Click the collapse/expand icons to the left of the class to show or hide properties and operations.



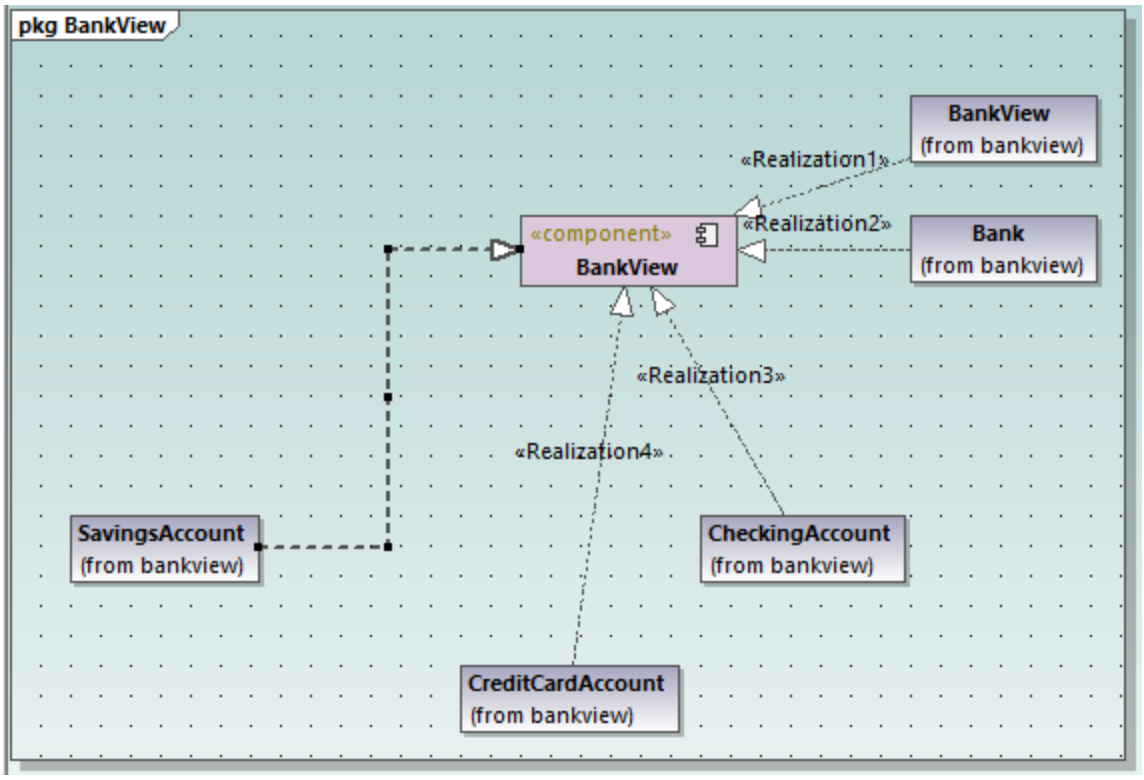
To create a realization dependency between the class and the component, do one of the following:

- Click the **Realization** toolbar button and drag from the `SavingsAccount` class to the `BankView` component.
- Move the cursor over the "ComponentRealization" handle of the class and drag to the **BankView** component.



The realization dependency between `SavingsAccount` and `BankView` has now been created.

© 2019-2025 Altova GmbH



To give a name to the new dependency line (for example, "Realization5"), first select the line, and then start typing its name directly. Alternatively, select the line, and then edit the **Name** property in the **Properties** window.

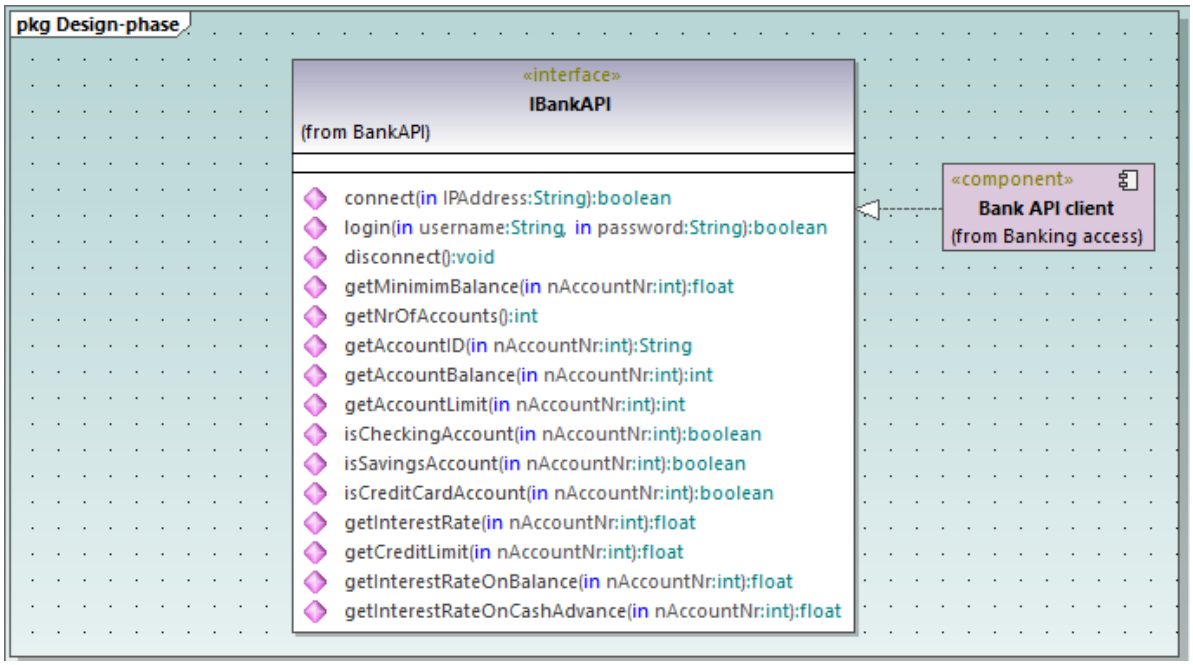
Changing the appearance of diagram lines

Let's now change the line appearance from "curved" to "direct line", as follows:

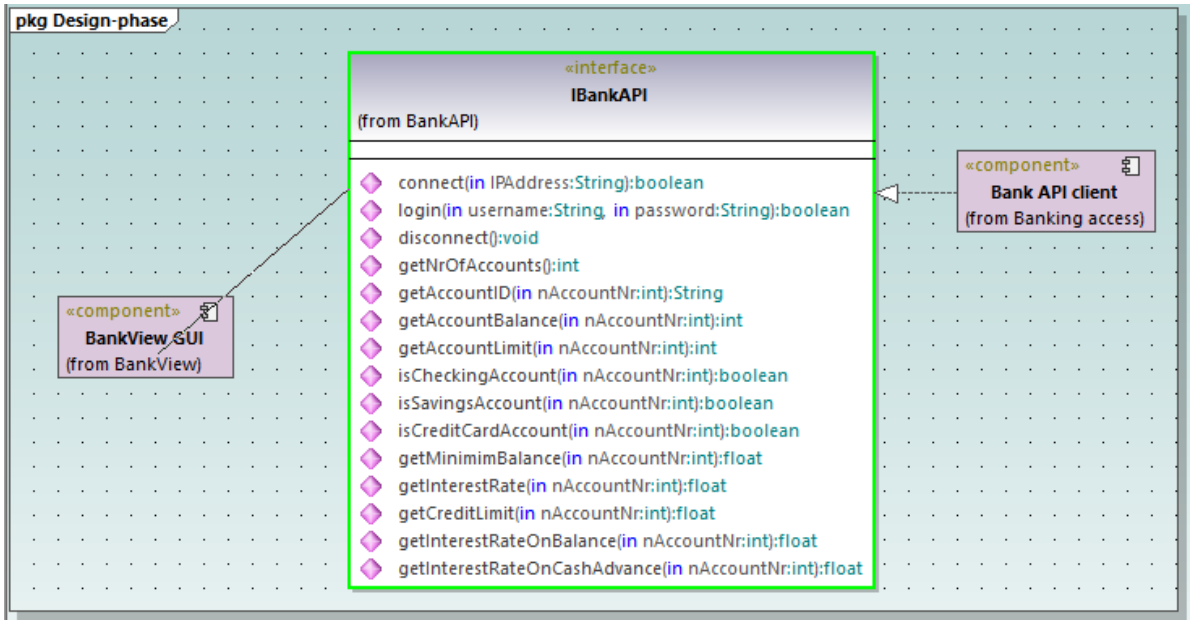
1. Select the line created previously (that is, the one between `SavingsAccount` and `BankView`).
2. Click the **Direct Line** toolbar button.

Adding usage dependencies to an interface

1. In the **Model Tree** window, navigate to **Root | Design-phase** and double-click the icon next to the "Overview" diagram. The "Overview" component diagram is opened and displays the currently defined system dependencies between components and interfaces.



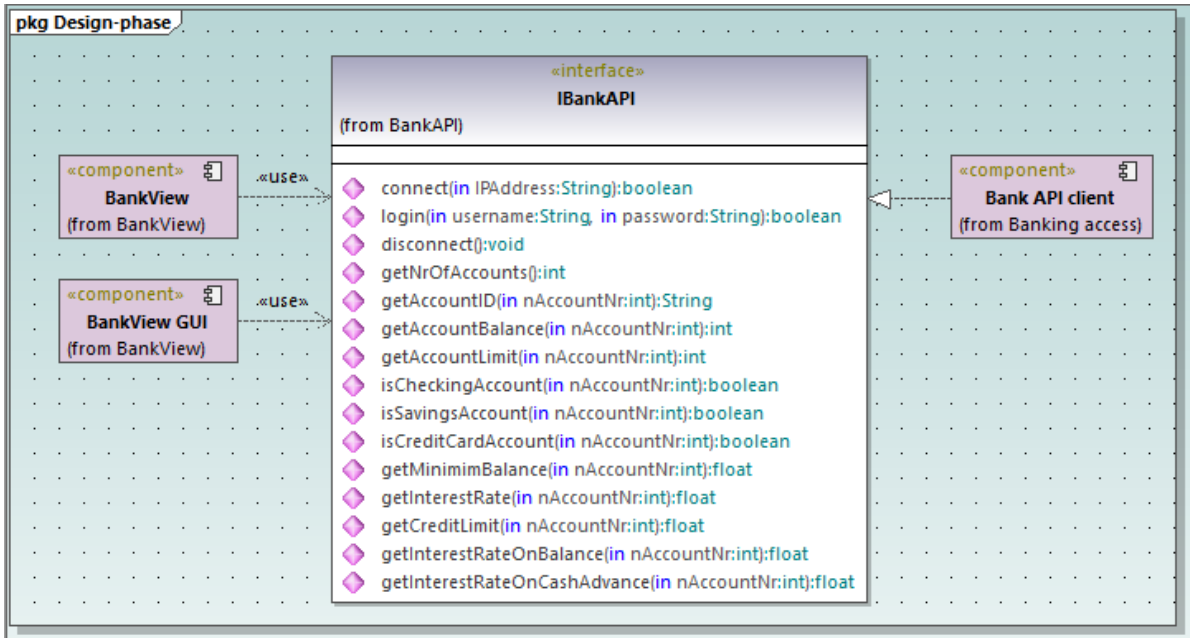
2. In the **Model Tree** window, navigate to **Root | Component View | BankView** and drag the `BankView GUI` package into the diagram.
3. Also drag the `BankView` package into the diagram.
4. Click the **Usage** toolbar button and drag from the `BankView GUI` package to the `IBankAPI` Interface.



5. Repeat the previous step for the package `BankView`.

© 2019-2025 Altova GmbH

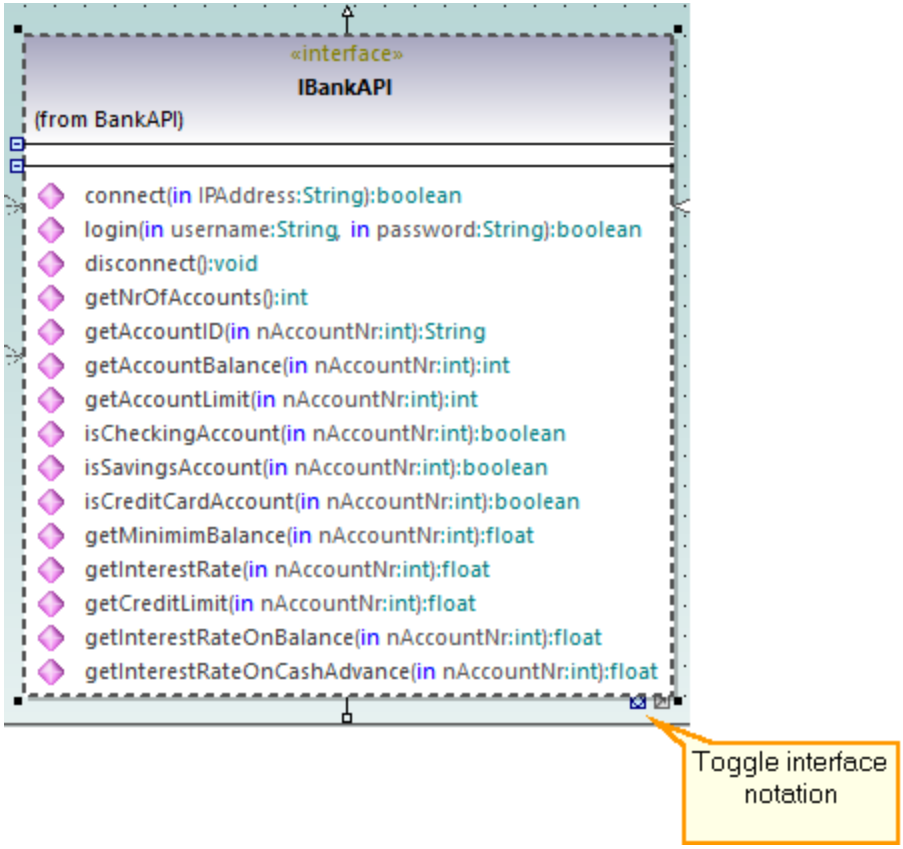
As illustrated below, both packages now have a usage dependency to the interface. Namely, the `IBankAPI` interface is required by the packages `BankView` and `BankView GUI`. As for the package `Bank API Client`, it provides the interface.



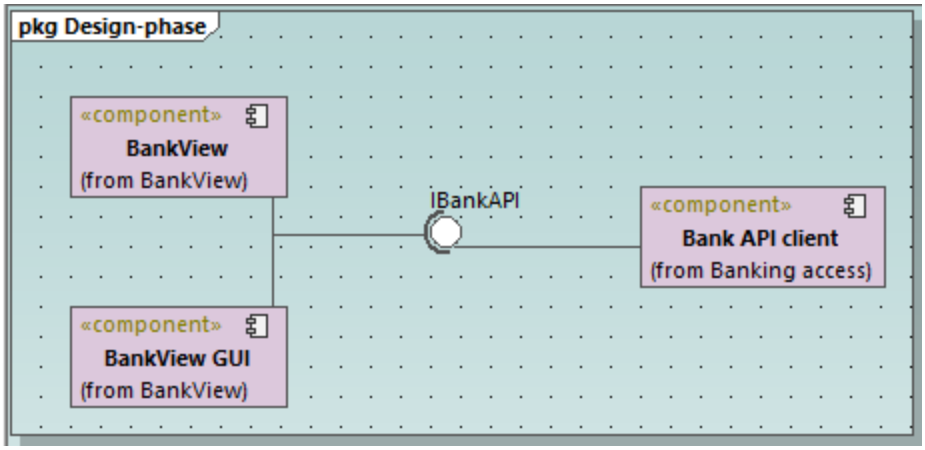
Using "ball-and-socket" notation

Optionally, it is possible to convert the current diagram notation to "ball-and-socket" style notation, as follows:

- Select the interface, and then click the **Toggle Interface Notation** button in its lower-right corner.



The diagram has now changed to "ball-and-socket" notation.



To switch back to the previous notation style, select the interface, and then click the **Toggle interface notation** button again.

© 2019-2025 Altova GmbH

2.6 Deployment Diagrams

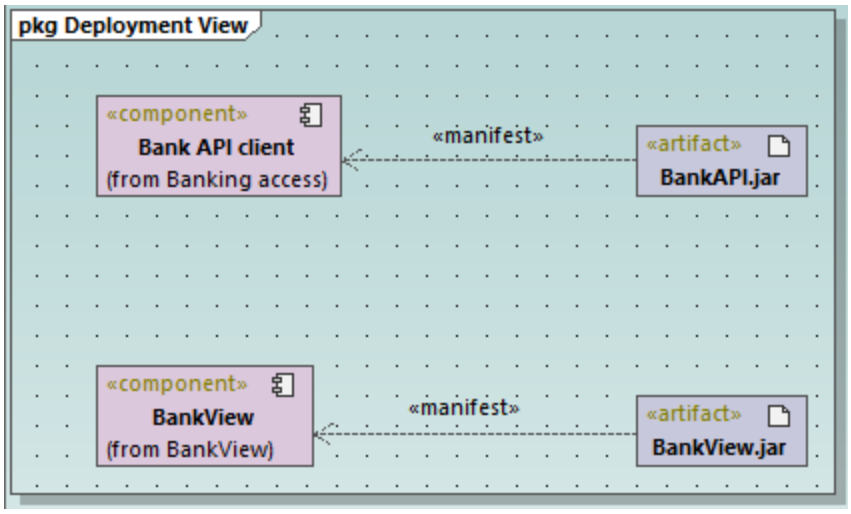
This tutorial section illustrates the following tasks:

- Add a dependency between two artifacts in a Deployment diagram
- Add elements to a Deployment diagram
- Embed artifacts into a node in a Deployment diagram
- Creating artifact elements (for example, properties, operations, nested artifacts)

To proceed, run UModel and open the **BankView-start.ump** project (see also Opening the Tutorial Project 19).

Adding a dependency between two artifacts in a Deployment diagram

In the **Diagram Tree** window, under "Deployment Diagrams", double-click the icon next to the "Artifacts" diagram to open it. As illustrated below, this diagram shows the manifestation of the `Bank API client` and the `BankView` components, to their respective compiled Java .jar files.



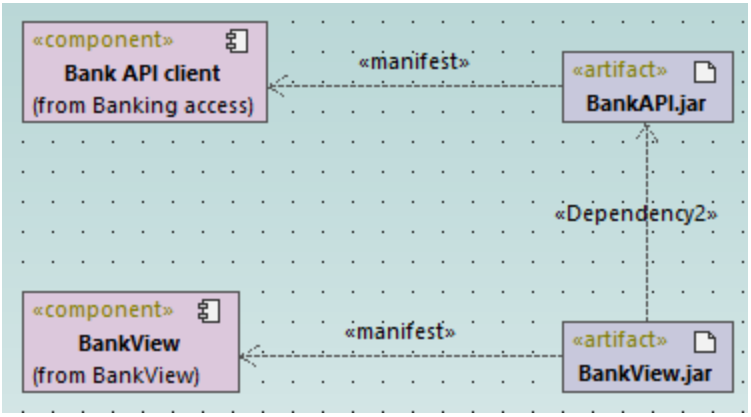
"Artifacts" diagram

These manifestations were created using a technique similar to other relationships previously illustrated in this tutorial, as follows:

1. Click the **Manifestation** toolbar button.
2. Move the mouse cursor over the artifact and drag into the component.

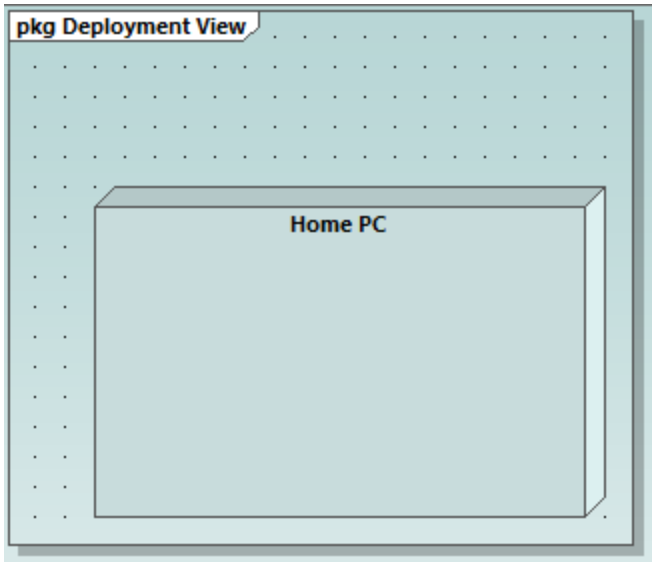
Using the same technique, let's also add a dependency between the two .jar files, as follows:

1. Click the **Dependency** toolbar button.
2. Move the cursor over the `BankView.jar` artifact and drag into the `BankAPI.jar` artifact.
3. Select the dependency line and type "Dependency2".



Adding elements to a Deployment diagram

In the **Diagram Tree** window, under "Deployment Diagrams", double-click the icon next to the "Deployment" diagram to open it. This diagram is deliberately incomplete and consists of a single node, which represents a home PC. In the following steps, we will be adding more elements to this diagram.

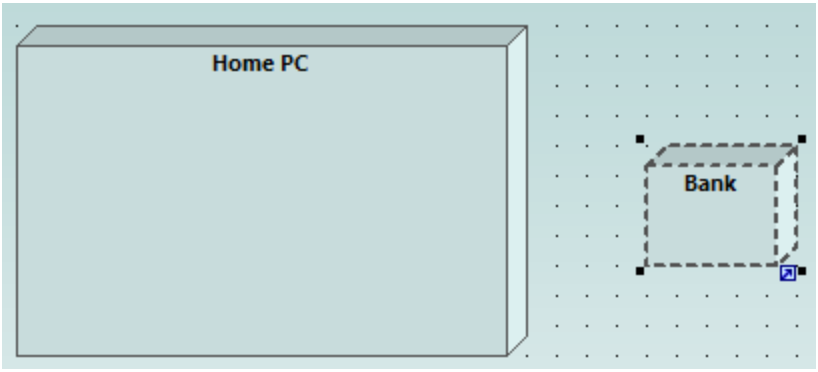


"Deployment" diagram

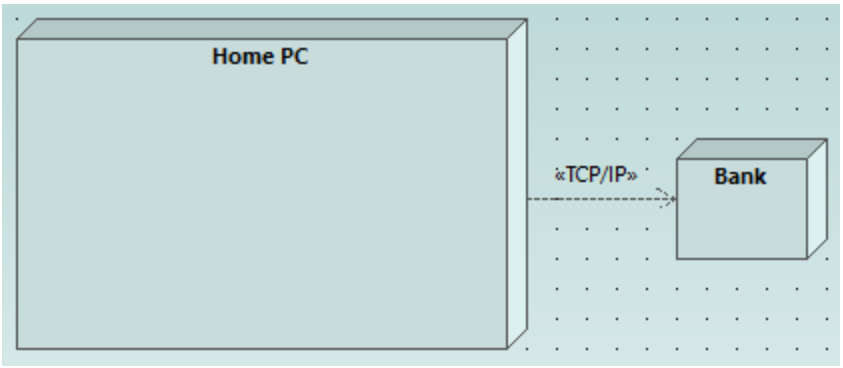
Assuming that the goal is to illustrate a TCP/IP connection between the home PC and a bank, let's add the required elements:

1. Click the **Node** toolbar button, and click right of the Home PC node to insert it.
2. Rename the node to "Bank", and drag one of its edges to enlarge it.

© 2019-2025 Altova GmbH

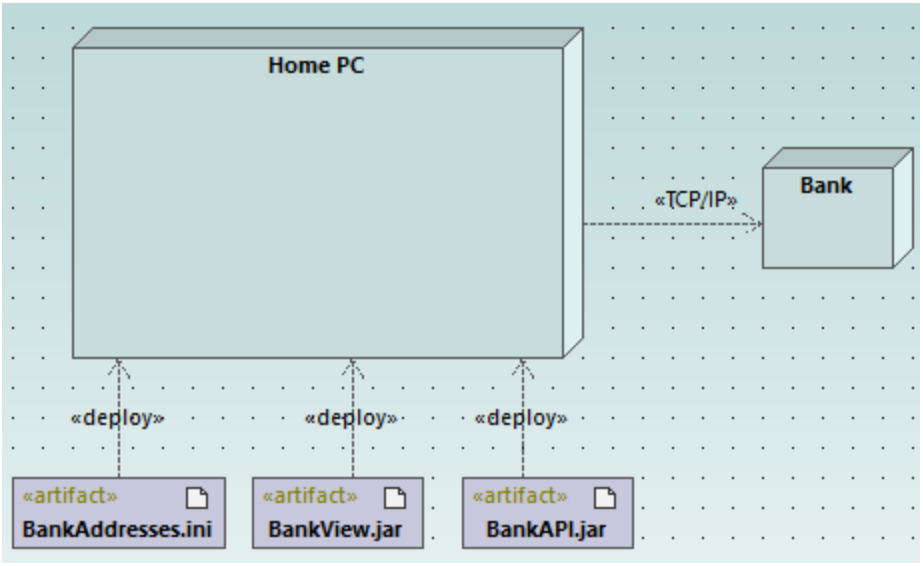


3. Click the **Dependency** toolbar button, and then drag from the "Home PC" node to the "Bank" node. This creates a dependency between the two nodes.
4. Select the dependency line and enter "TCP/IP" as name of the new dependency. (Alternatively, edit the **Name** property in the **Properties** window).

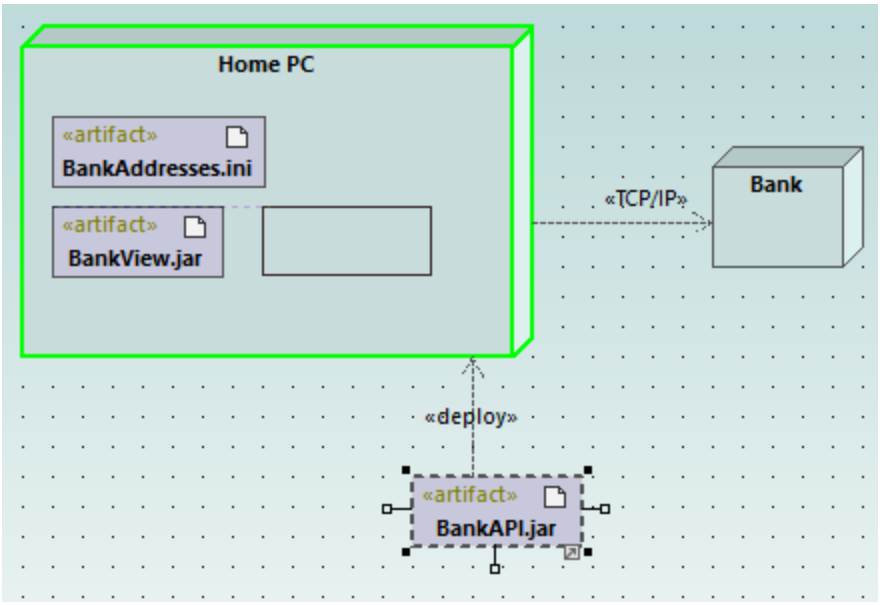


Embedding artifacts

In the **Model Tree** window, expand the "Deployment View" package, and then drag all of the following artifacts into the diagram: **BankAddresses.ini**, **BankAPI.jar**, and **BankView.jar**. The project is preconfigured to include deploy dependencies between these artifacts and the "Home PC" node, so all these dependencies are now visible in the diagram:



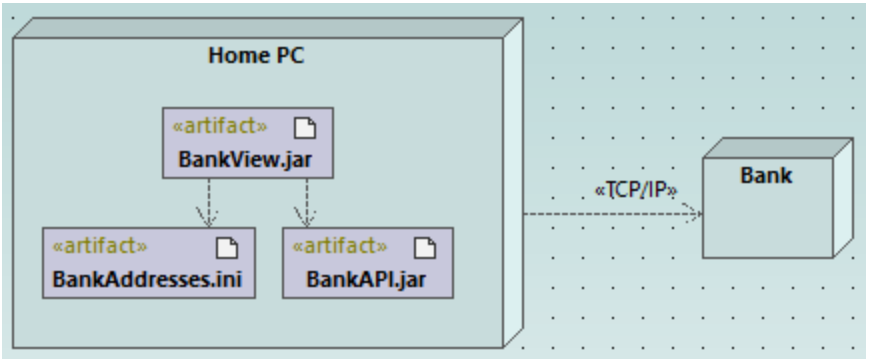
You can also embed the artifacts into the "Home PC" node, by dragging each of the artifacts into it. Notice that the deploy dependencies are no longer visible on the diagram, although they continue to exist logically.



Artifacts embedded into the node can also have dependencies between them. To illustrate this:

1. Click the **Dependency** toolbar button and, holding the **Ctrl** key pressed, drag from the "BankView.jar" artifact into the "BankAddresses.ini".
2. While holding the **Ctrl** key pressed, drag from the "BankView.jar" artifact into the "BankAPI.jar" artifact.

© 2019-2025 Altova GmbH



Note: Dragging an artifact out of a node onto the diagram always creates a deployment dependency automatically.

Creating artifact elements (properties, operations, nested artifacts)

In UML, artifacts can be composed of properties, operations, and other elements, including nested artifacts. To create such nested elements, right-click the artifact in the **Model Tree** window and select the appropriate action from the context menu (for example, **New Element | Operation**, or **New Element | Property**). The new element will appear nested below the selected artifact in the **Model Tree** window.

2.7 Forward Engineering (from Model to Code)

This example illustrates how to create a new UModel project and generate program code from it (a process known as "forward engineering"). For the sake of simplicity, the project will be very simple, consisting of only one class. You will also learn how to prepare the project for code generation and check that the project uses the correct syntax. After generating program code, you will modify it outside UModel, by adding a new method to the class. Finally, you will learn how to merge the code changes back into the original UModel project (a process known as "reverse engineering").

The code generation language used in this tutorial is Java; however, similar instructions are applicable for other code generation languages.

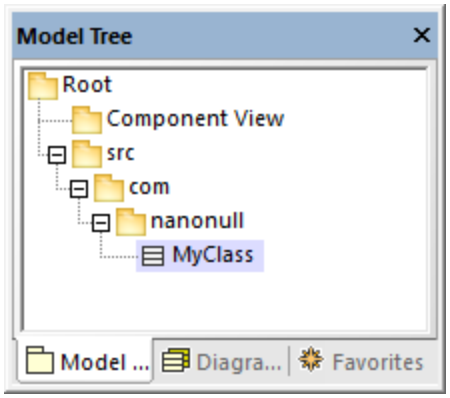
Creating a new UModel project

You can create a new UModel project as follows:

- On the **File** menu, click **New**. (Alternatively, press **Ctrl+N**, or click the New toolbar button.)

At this stage, the project contains only the default "Root" and "Component View" packages. These two packages cannot be deleted or renamed. "Root" is the top grouping level for all other packages and elements in the project. "Component View" is required for code engineering; it typically stores one or more UML components that will be realized by the classes or interfaces of your project; however, we didn't create any classes yet. Therefore, let's first design the structure of our program, as follows:

1. Right-click the "Root" package in the Model Tree window and select **New Element | Package** from the context menu. Rename the new package to "src".
2. Right-click "src" and select **New Element | Package** from the context menu. Rename the new package to "com"
3. Right-click "com" and select **New Element | Package** from the context menu. Rename the new package to "nanonull".
4. Right-click "nanonull" and select **New Element | Class** from the context menu. Rename the new class to "MyClass".



Preparing the project for code generation

To generate code from a UModel model, the following requirements must be met:

© 2019-2025 Altova GmbH

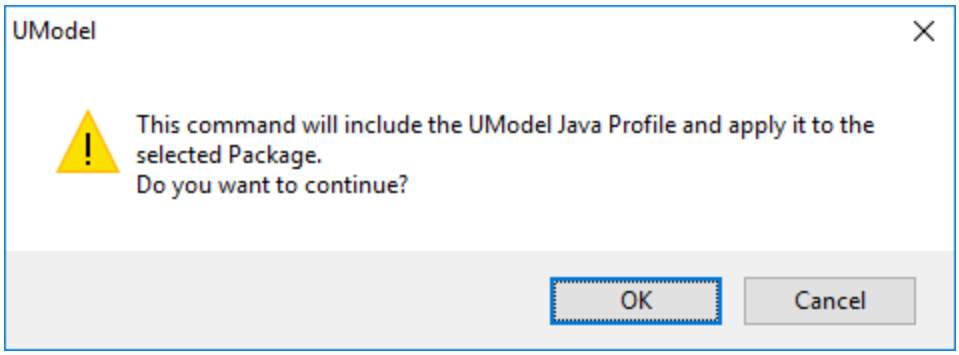
- A Java, C#, or VB.NET namespace root package must be defined.
- A component must exist which is realized by all classes or interfaces for which code must be generated.
- The component must have a physical location (directory) assigned to it. Code will be generated in this directory.
- The component must have the property **use for code engineering** enabled.

All of these requirements are explained in more detail below. Note that you can always check if the project meets all code generation requirements, by validating it:

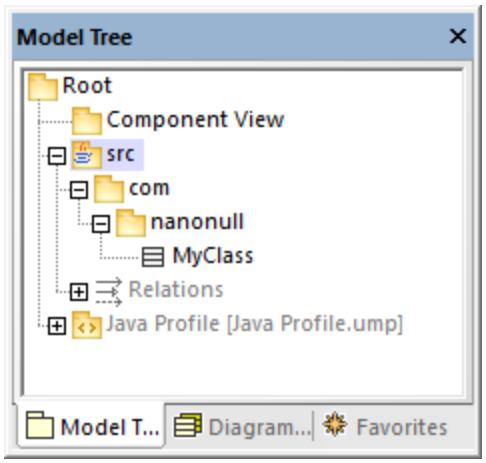
- On the **Project** menu, click **Check Project Syntax**. (Alternatively, press **F11**.)

If you validate the project at this stage, the Messages window displays a validation error ("*No Namespace Root found! Please use the context menu in the Model Tree to define a Package as Namespace Root*"). To resolve this, let's assign the package "src" to be the namespace root:

- Right-click the "src" package and select **Code Engineering | Set As Java Namespace Root** from the context menu.
- When prompted that the UModel Java Profile will be included, click **OK**.



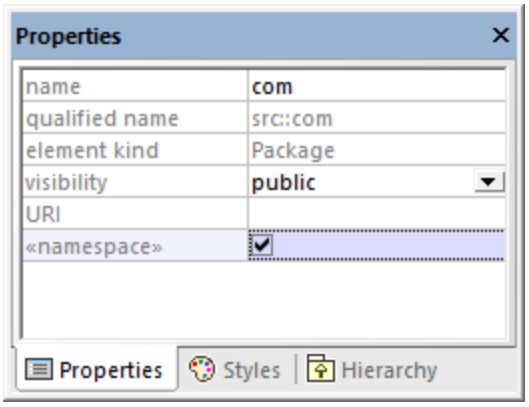
Notice the package icon has now changed to , which signifies that this package is a Java namespace root. Additionally, a Java Profile has been added to the project.



The actual namespace can be defined as follows:

1. Select the package "com" in the **Model Tree** window.

2. In the **Properties** window, enable the **<<namespace>>** property.

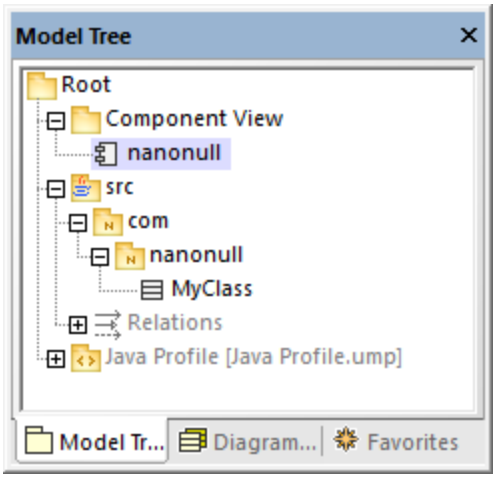


3. Repeat the step above for the "nanonull" package.

Notice that the icon of both "com" and "nanonull" packages has now changed to , which indicates these are now namespaces.

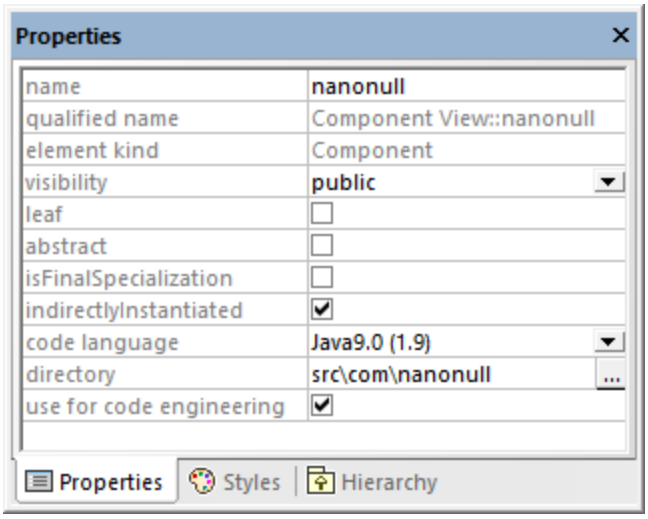
Another requirement for code generation is that a component must be realized by at least a class or an interface. In UML, a component is a piece of the system. In UModel, the component lets you specify the code generation directory and other settings; otherwise, code generation would not be possible. If you validate the project at this stage, a warning message is displayed in the **Messages** window: "*MyClass has no ComponentRealization to a Component - no code will be generated*". To solve this, a component must be added to the project, as follows:

1. Right-click "Component View" in the Model Tree window, and select **New Element | Component** from the context menu.
2. Rename the new Component to "nanonull".



3. In the **Properties** window, change the **directory** property to a directory where code should be generated (in this example, "src\com\nanonull"). Notice that the property **use for code engineering** is enabled, which is another prerequisite for code generation.

© 2019-2025 Altova GmbH



4. Save the UModel project to a directory and give it a descriptive name (in this example, **C:\UModelDemo\Tutorial.ump**).

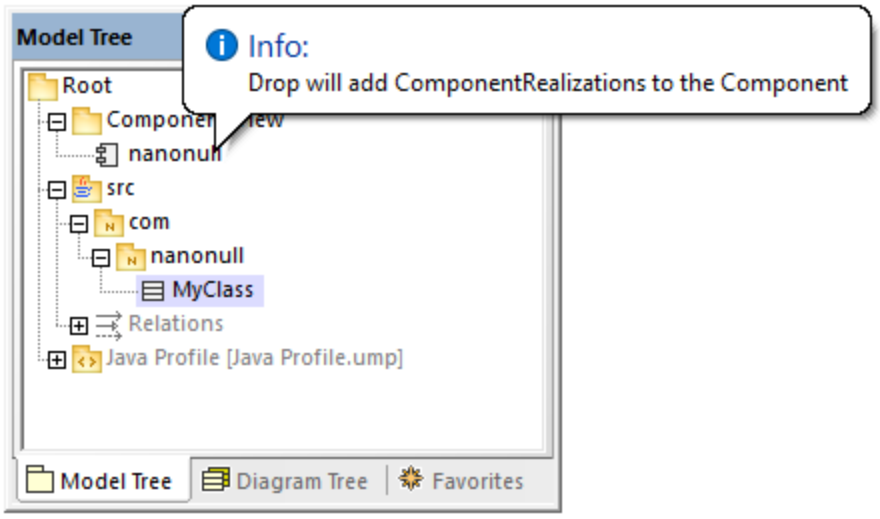
Note: The code generation path can be absolute or relative to the .ump project. If it is relative as in this example, a path such as **src\com\nanonull** would create all the directories in the same directory where the UModel project was saved.

We have deliberately chosen to generate code to a path which includes the namespace name; otherwise, warnings would occur. By default, UModel displays project validation warnings if the component is configured to generate Java code to a directory which does not have the same name as the namespace name. In this example, the component "nanonull" has the path "C:\UModelDemo\src\com\nanonull", so no validation warnings will occur. If you want to enforce a similar check for C# or VB.NET, or if you want to disable the namespace validation check for Java, do the following:

1. On the **Tools** menu, click **Options**.
2. Click the **Code Engineering** tab.
3. Select the relevant check box under **Use namespace for code file path**.

The component realization relationship can be created as follows:

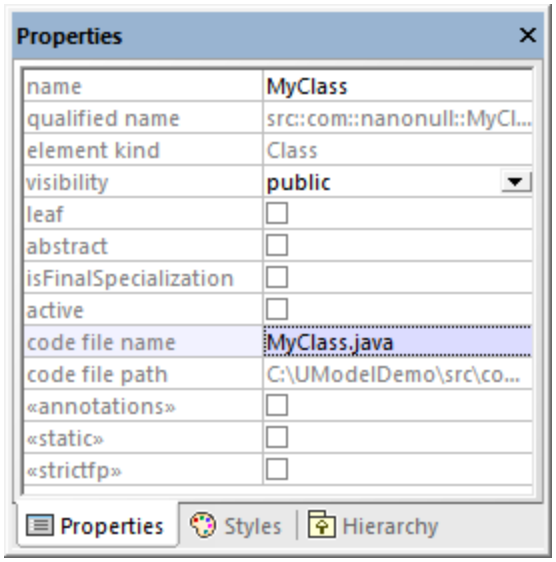
- In the **Model Tree** window, drag from the `MyClass` created previously and drop onto component `nanonull`.



The component is now realized by the project's only class `MyClass`. Note that the approach above is just one of the ways to create the component realization. Another way is to create it from a component diagram, as illustrated in the tutorial section Component Diagrams 53.

Next, it is recommended that the classes or interfaces which take part in code generation have a file name. Otherwise, UModel will generate the corresponding file with a default file name and the **Messages** window will display a warning ("*code file name not set - a default name will be generated*"). To remove this warning:

1. Select the class `MyClass` in the **Model Tree** window.
2. In the **Properties** window, change the property **code file name** to the desired file name (in this example, `MyClass.java`).



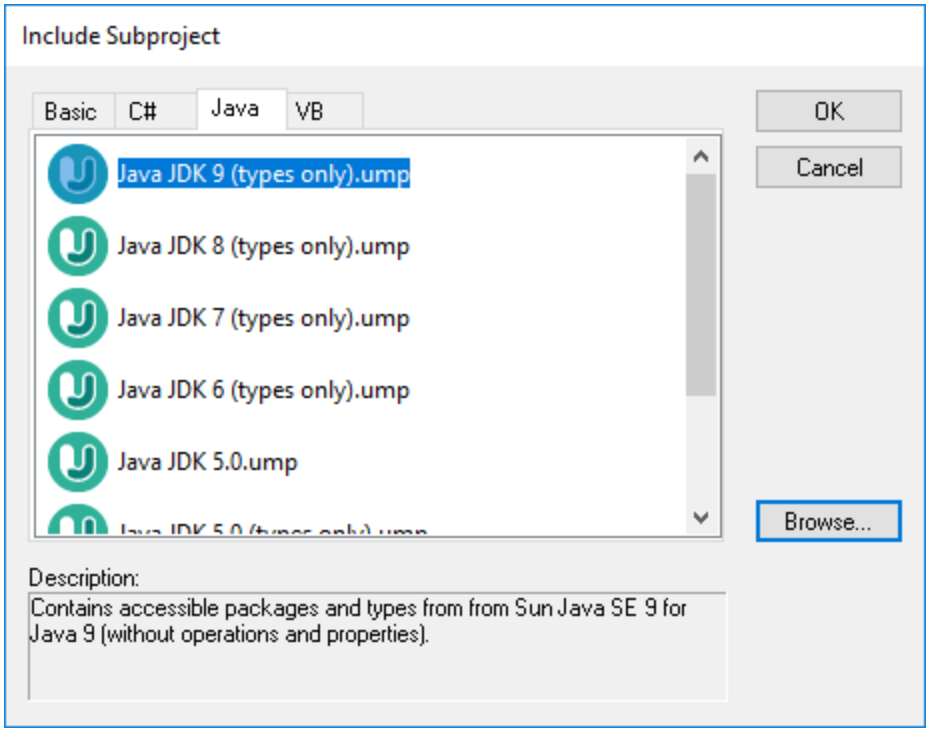
Including the JDK types

Although this step is optional, it is recommended that you include the Java Development Kit (JDK) language types, as a subproject of your current UModel project. Otherwise, the JDK types will not be available when you

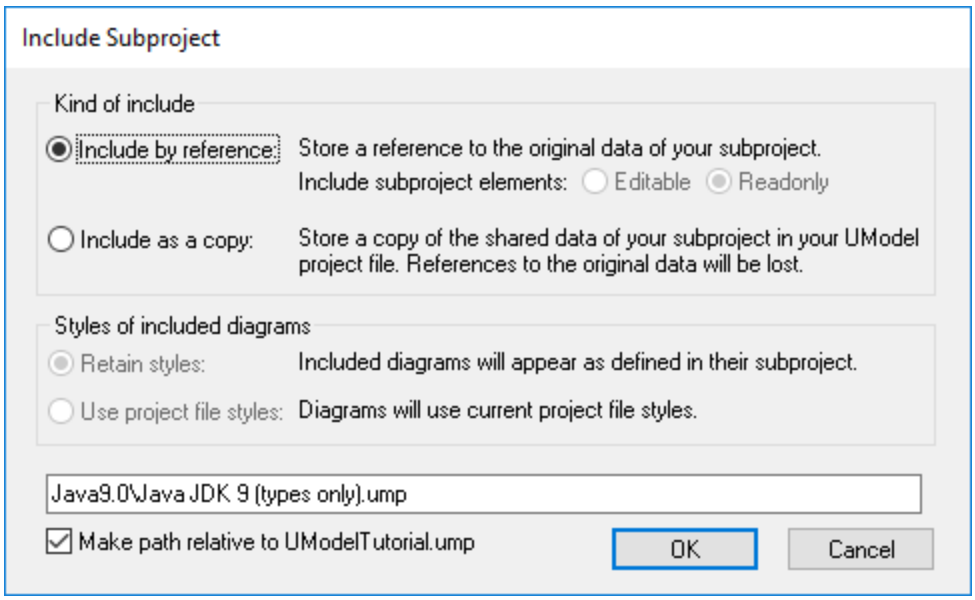
 © 2019-2025 Altova GmbH

create the classes or interfaces. This can be done as follows (the instructions are similar for C# and VB.NET):

1. On the **Project** menu, click **Include Subproject**.
2. Click the **Java** tab and select the **Java JDK 9 (types only)** project.



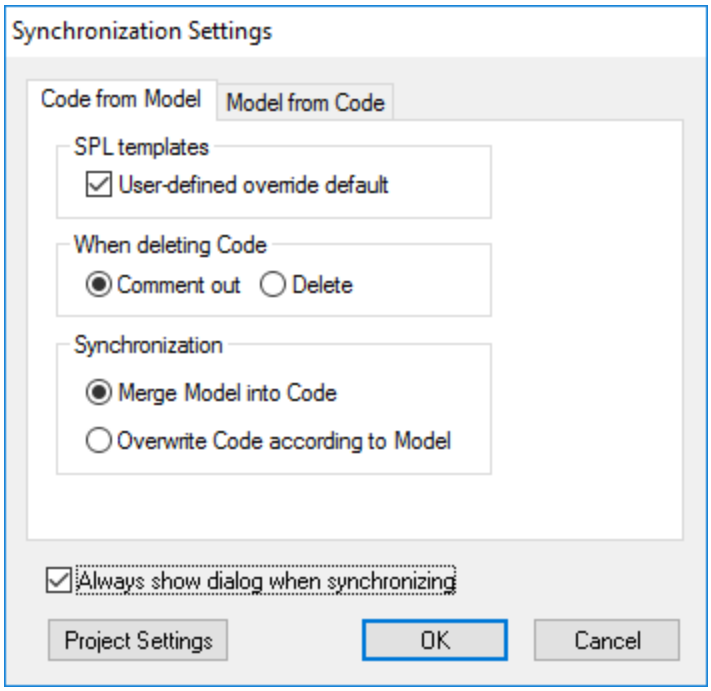
3. When prompted to include by reference or as a copy, select **Include by reference**.



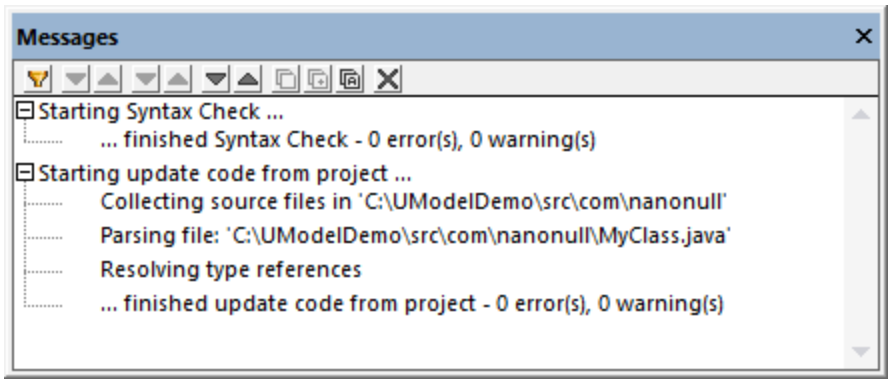
Generating code

Now that all prerequisites have been met, code can be generated as follows:

1. On the **Project** menu, click **Merge Program Code from UModel Project**. (Alternatively, press **F12**.) Note that this command will be called **Overwrite Program Code from UModel Project** if the **Overwrite Code according to Model** option was selected previously on the "Synchronization Settings" dialog box illustrated below.



2. Leave the default synchronization settings as is, and click **OK**. A project syntax check takes place automatically, and the **Messages** window informs you of the result:



Modifying code outside of UModel

Generating program code is just the first step to developing your software application or system. In a real life scenario, the code would go through many modifications before it becomes a full-featured program. For the scope of this example, open the generated file **MyClass.java** in a text editor and add a new method to the class, as shown below. The **MyClass.java** file should look as follows:

```
package com.nanonull;
```


© 2019-2025 Altova GmbH

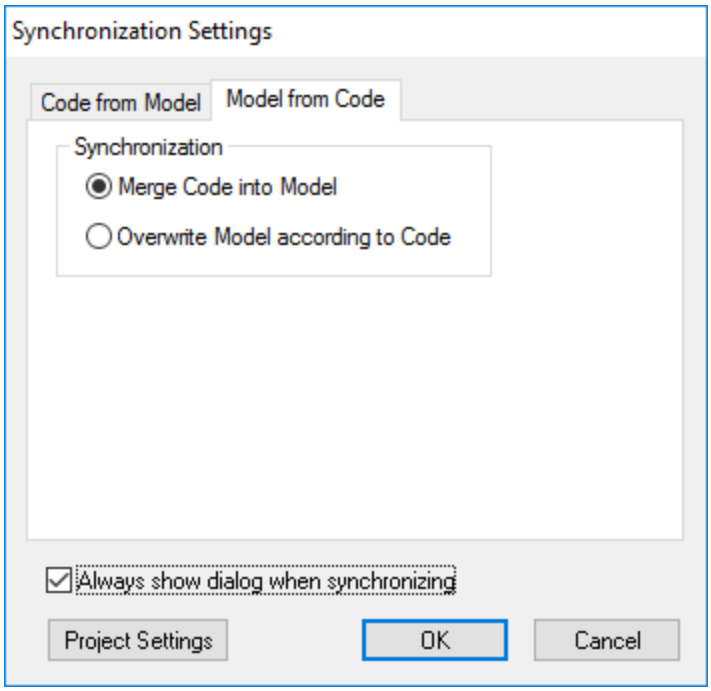
```
public class MyClass{
   public float sum(float num1, float num2){
    return num1 + num2;
  }
}
```

MyClass.java

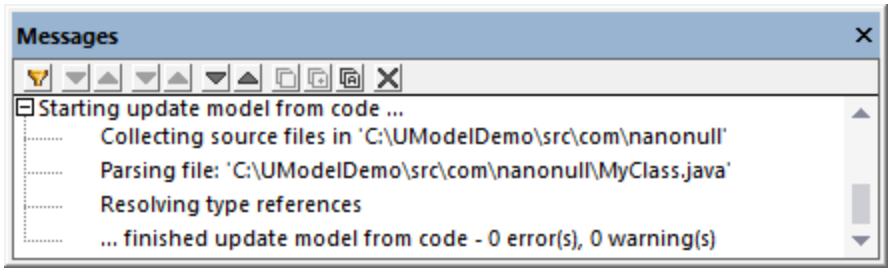
Merging code changes back into the model

You can now merge the code changes back into the model, as follows:

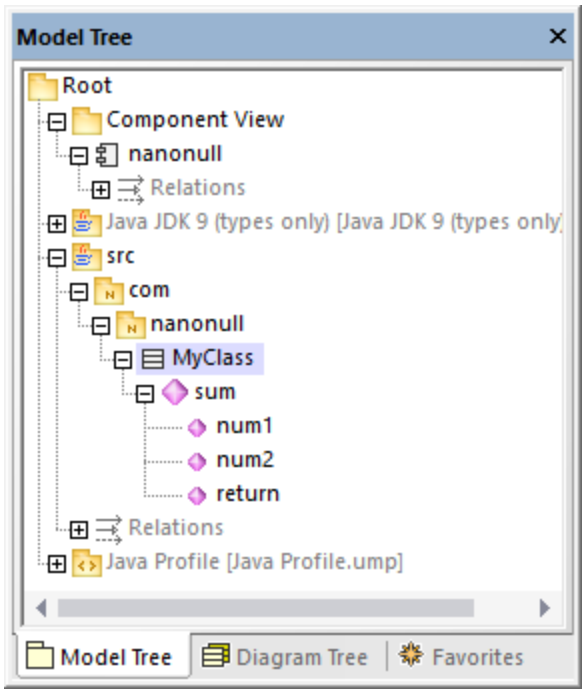
1. On the **Project** menu, click **Merge UModel Project from Program Code** (Alternatively, press **Ctrl + F12**).



2. Leave the default synchronization settings as is, and click OK. A code syntax check takes place automatically, and the **Messages** window informs you of the result:



The operation `sum` (which has been reverse engineered from code) is now visible in the Model Tree window.



© 2019-2025 Altova GmbH

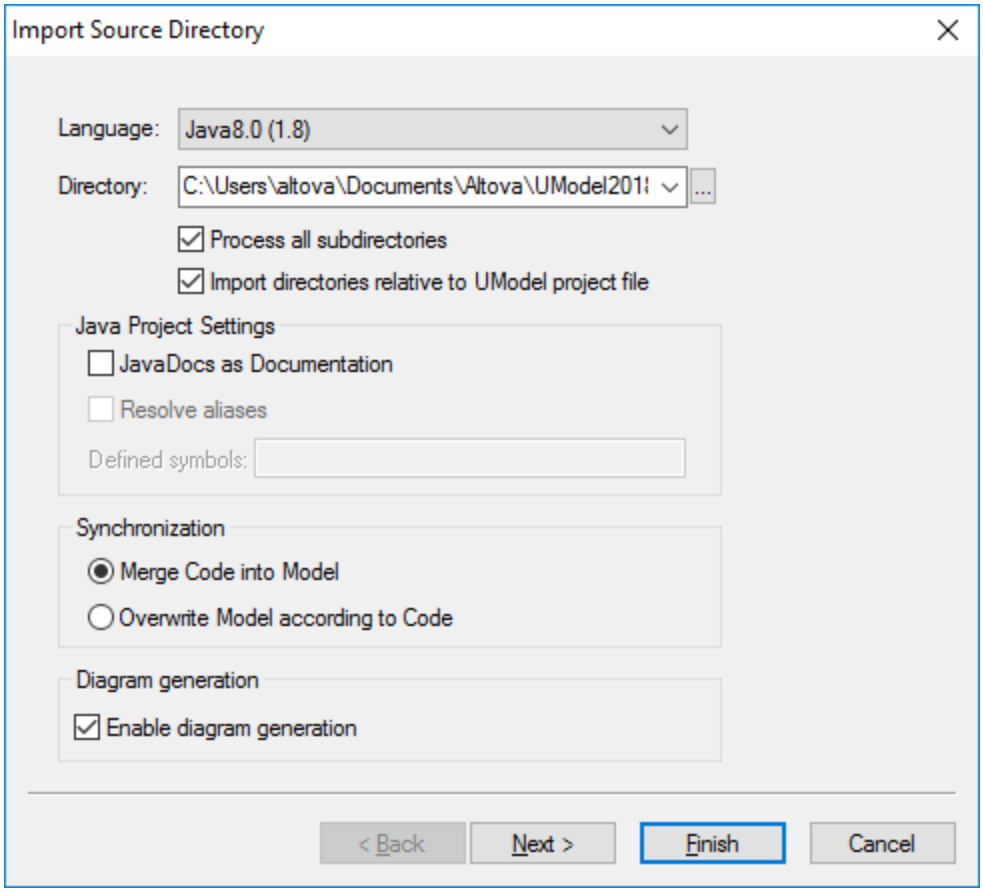
2.8 Reverse Engineering (from Code to Model)

This tutorial section illustrates how to import existing program code from a directory into a new UModel project (reverse engineering). You will also add a new class into the model, prepare it for code generation, and then merge changes back into the Java code (forward engineering). Although this tutorial illustrates importing Java code, the process is similar if you would like to import existing C# or VB.NET code.

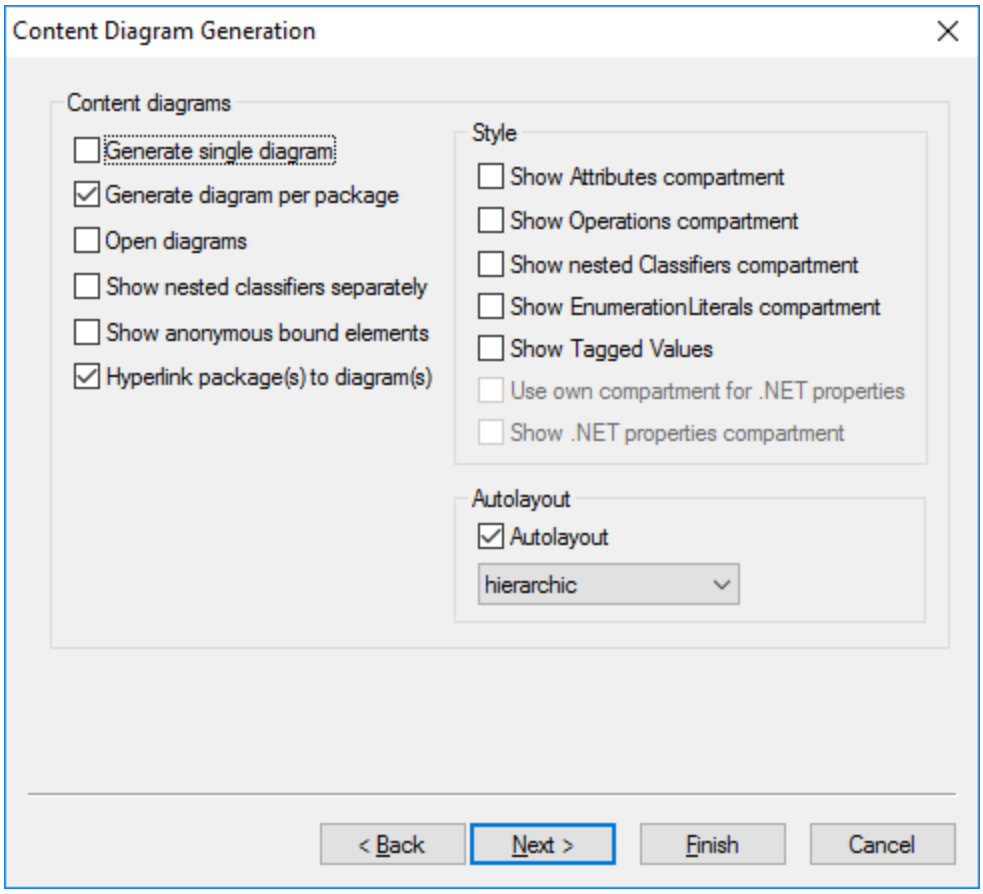
Note: The sample Java code used in this tutorial is available as a ZIP archive at the following path: **C:\Users\<username>\Documents\Altova\UModel2025\UModelExamples\OrgChart.zip**. Please unzip the archive to the same directory before starting the tutorial.

Importing existing code from a directory

1. On the **File** menu, click **New**.
2. On the **Project** menu, click **Import Source Directory**.
3. Select the language of the source code (in this example, Java).
4. Click the Browse button ..., select the **OrgChart** directory unzipped previously, and click **Next**. Notice the **Enable diagram generation** check box is selected, which instructs UModel to generate Class Diagrams 387 and Package Diagrams 405 from the source code.

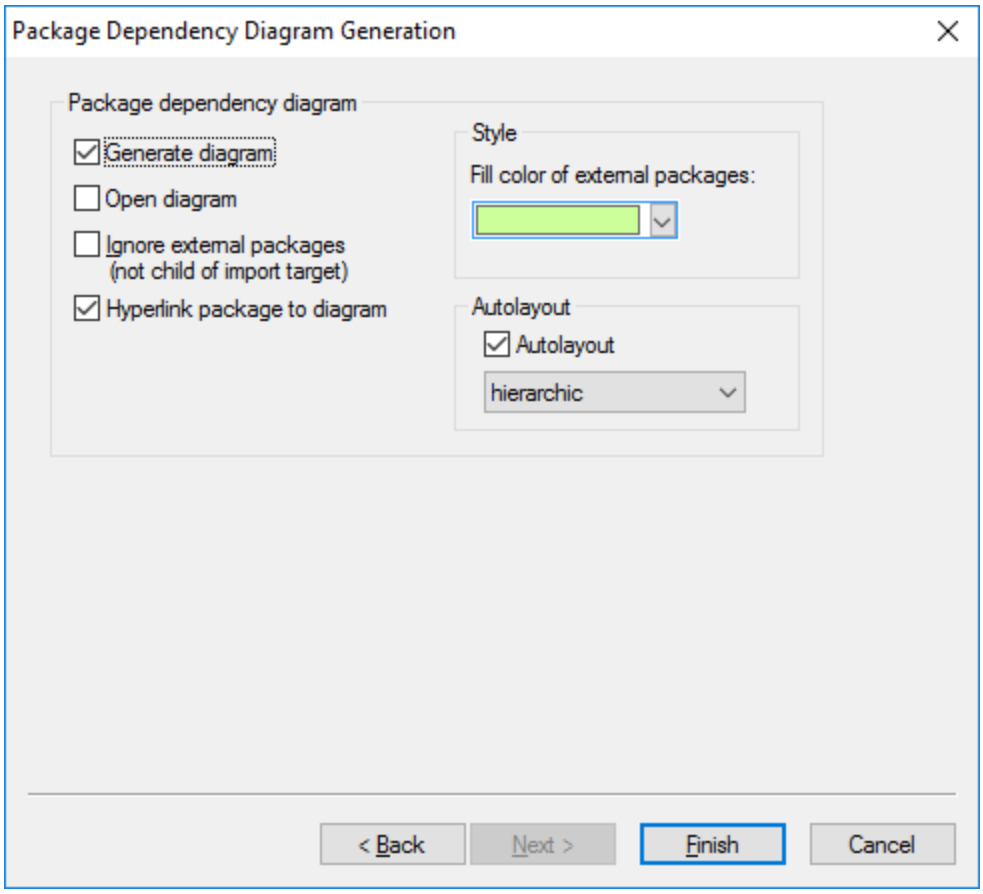


5. Select the **Generate diagram per package** option. This instructs UModel to create a new diagram for each package. The diagram styling options can be changed later if necessary.

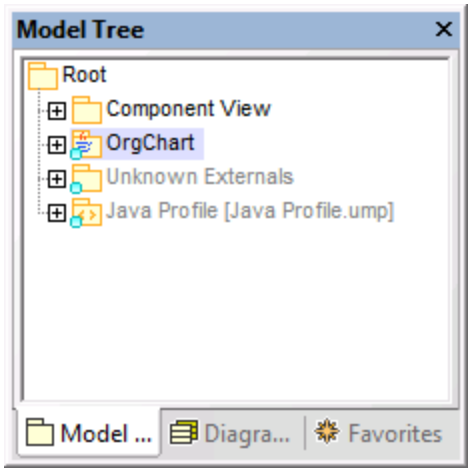


6. Click **Next** to continue. This dialog box allows you to define the package dependency generation settings.

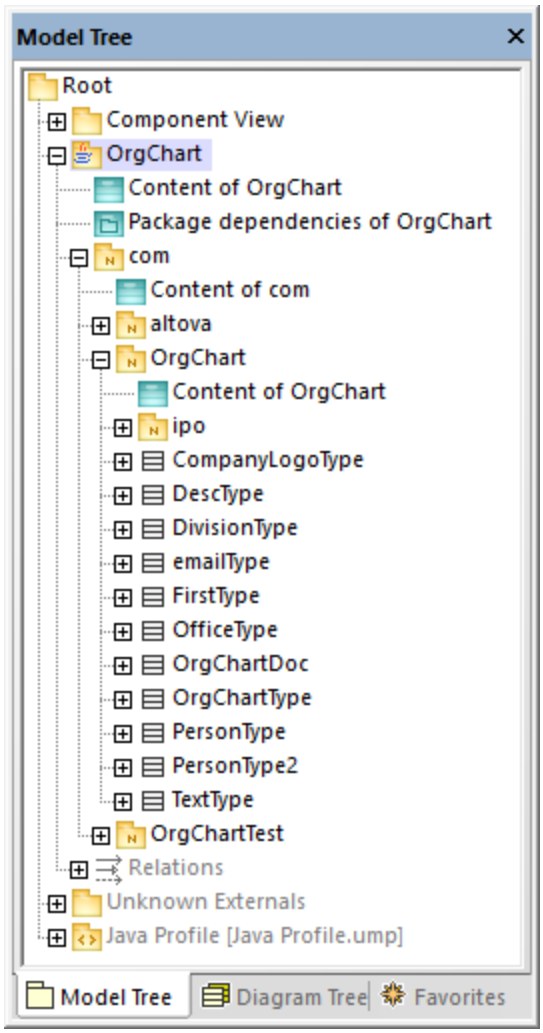
© 2019-2025 Altova GmbH



7. Click **Finish**. When prompted, save the new model to a directory on your system. The data is parsed, and a new package called "**OrgChart**" is created.

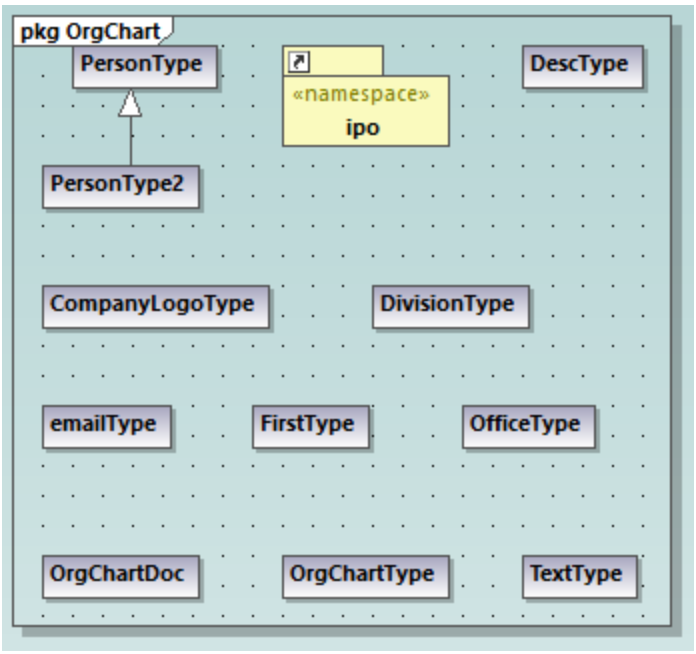


8. Expand the new package and keep expanding the sub packages until you get to the **OrgChart** package (**com | OrgChart**). Double-click the "**Content of OrgChart**" diagram icon:



The "Content of OrgChart" diagram is now displayed in the main pane.

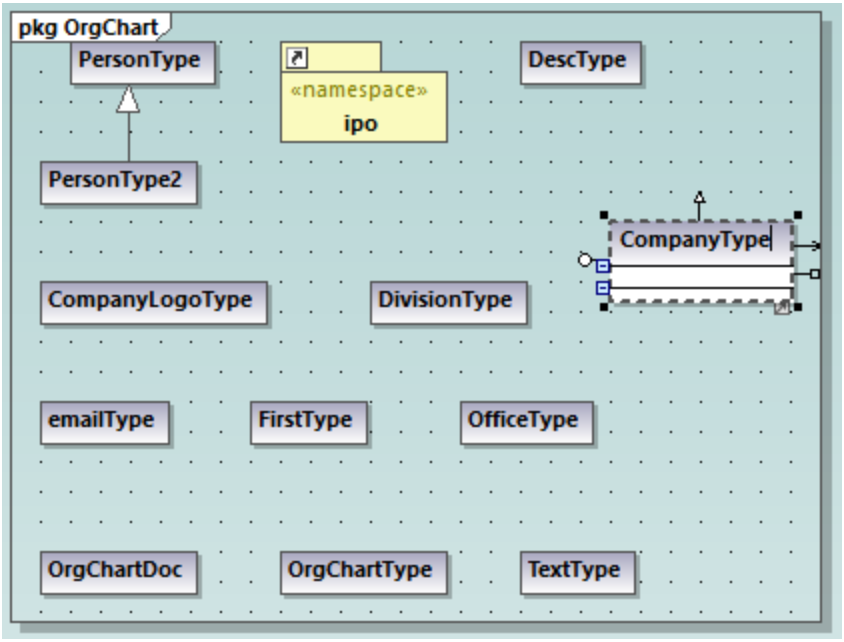
© 2019-2025 Altova GmbH



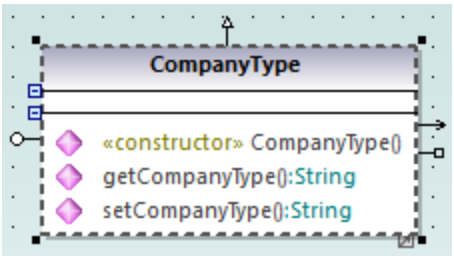
Adding a new class to the OrgChart diagram

At this stage, you have fully reverse engineered some existing Java code and created a model out of it, which also includes several automatically generated diagrams. We will now go one step further, and extend the model to include a new class.

1. Right-click inside the "Content of OrgChart" diagram, and then select **New | Class** from the context menu.
2. Click the header of the new class, and enter **CompanyType** as the name of the new class.



3. Add new operations to the class using the **F8** shortcut key. For the purpose of this example, add the following operations: `CompanyType()`, `getCompanyType():String`, `setCompanyType():String`.

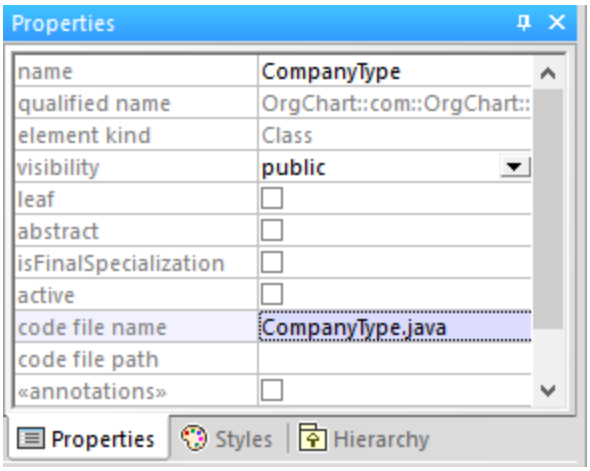


Note: Since the class name is `CompanyType`, the operation `CompanyType()` is automatically assigned the `<<constructor>>` stereotype.

Making the new class available for code generation

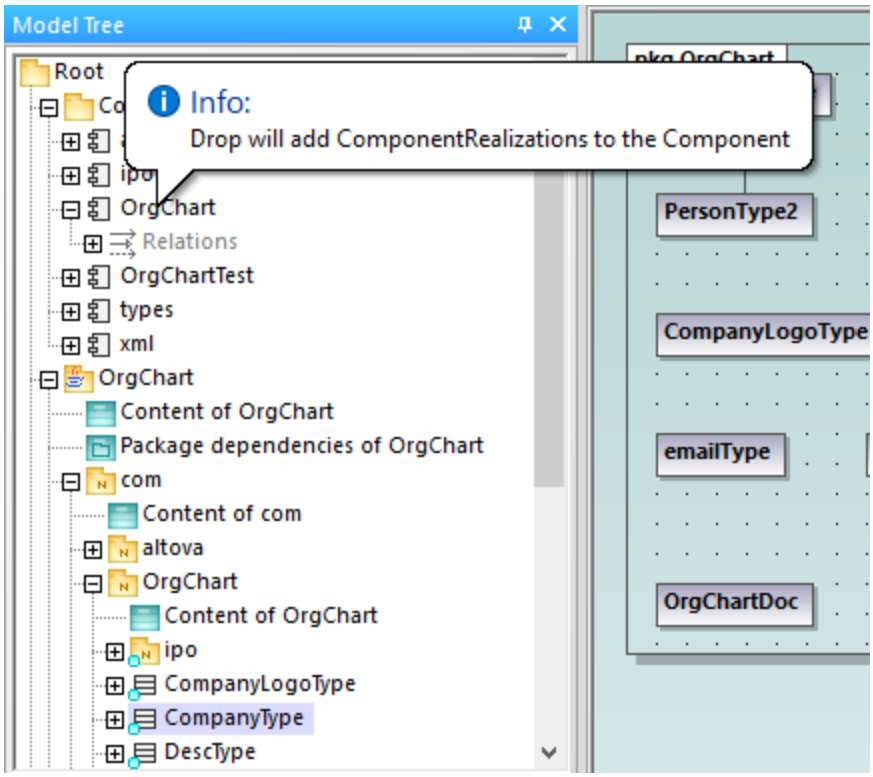
Now that the model has been extended with a new class, you will most likely want to update the underlying code accordingly, in order to keep both in sync. However, if you press **F11** to check the project syntax at this stage, a warning is displayed in the Messages window: *'CompanyType' has no Component Realization to a Component - ComponentRealization to Component 'OrgChart' will be generated*. The reason is that the new class requires realization to a component before code can be generated from it, as explained in Round-Trip Engineering (Model-Code-Model) 64. In some cases (including this example), UModel can generate the required realization automatically; however, you can also define the realization dependency manually, as follows:

1. While the `CompanyType` class is selected in the diagram, locate the property "code file name" in the Properties window and enter "CompanyType.java" as file name.

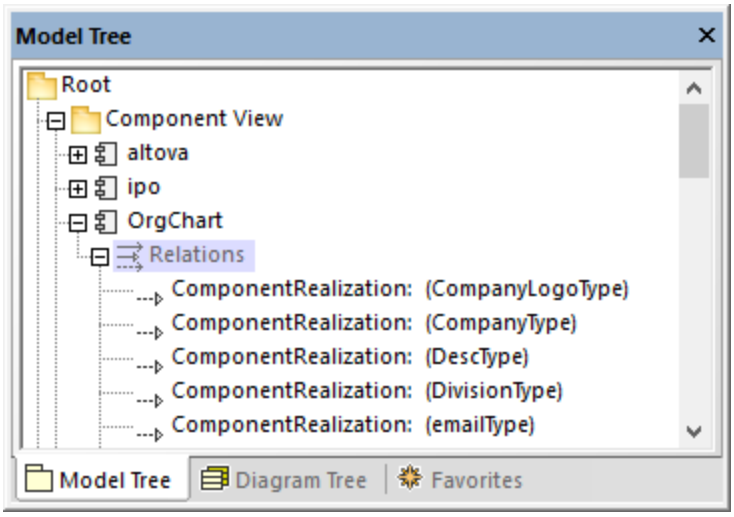


2. Click the new `CompanyType` class in the Model Tree, drag upwards and drop onto the **OrgChart** component below the Component View package. A notification appears when the mouse pointer is over a component.

© 2019-2025 Altova GmbH



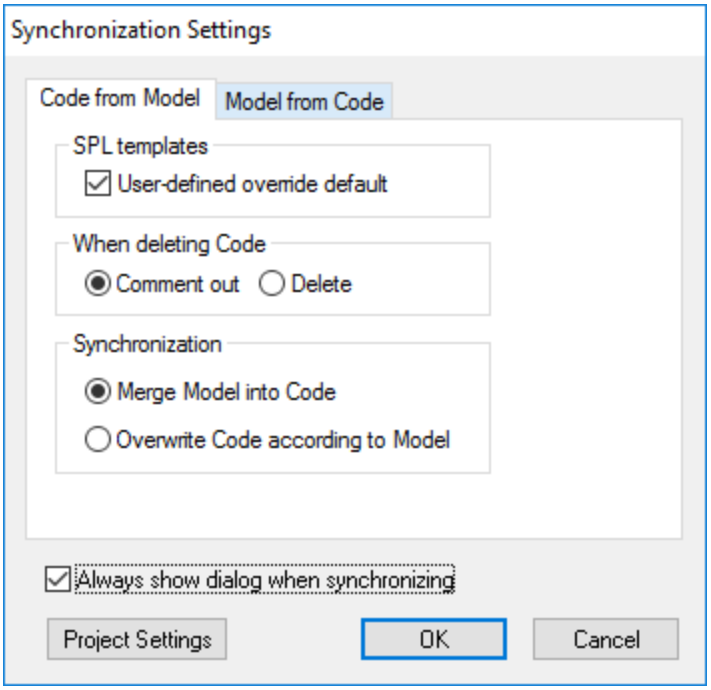
This method creates a relation of type "ComponentRealization" between a class and a component. An alternative way to do this is to draw the relation in a component diagram, see Component Diagrams 53 . Expand the **Relations** item below **OrgChart** to see the newly created relation.



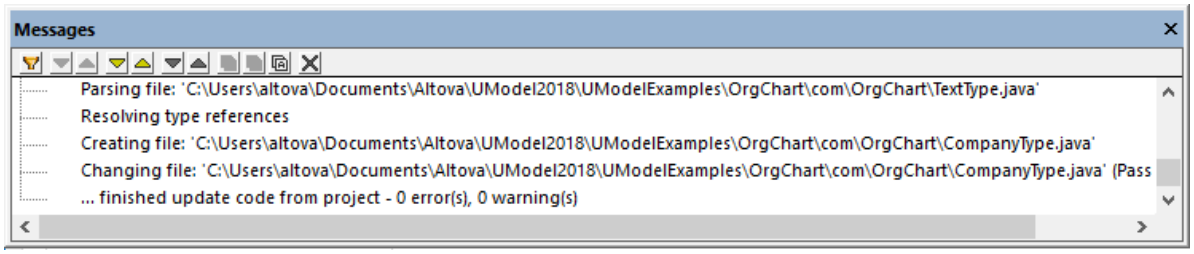
Merging program code from a package

In UModel, you can generate code at package level, component level, or for the entire project, see also Synchronizing the Model and Source Code 217 . In this example, we will generate code at component level, as follows:

1. In the Model Tree window, locate the OrgChart component in the "Component View".
2. Right-click the OrgChart component, and select **Code Engineering | Merge Program code from UModel Component** from the context menu.



The messages window displays the syntax checks being performed and status of the synchronization process.



When the process completes, the new **CompanyType.java** class has been added to the folder **... \OrgChart\com\OrgChart\.**

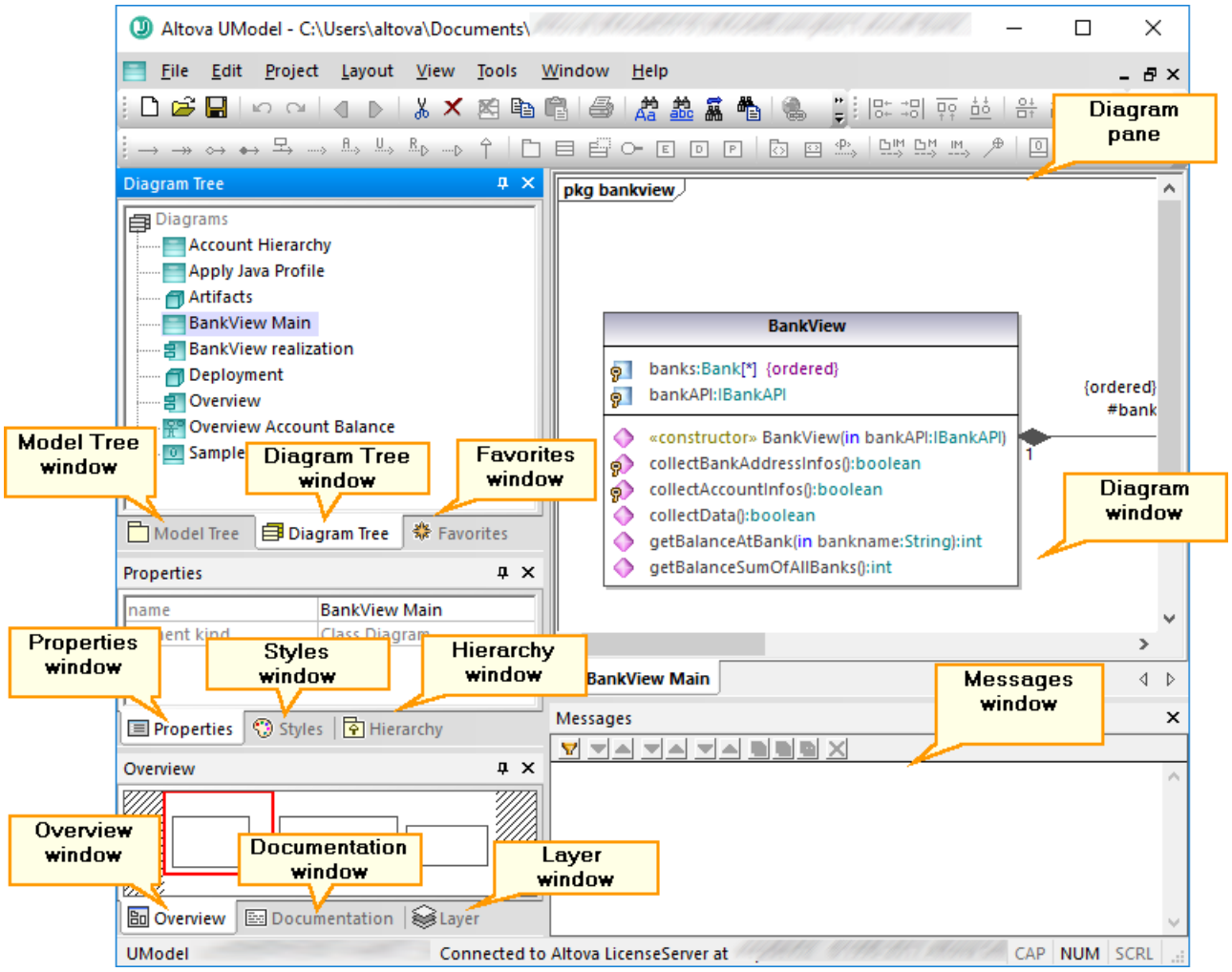
> All method bodies and changes to the code will either be commented out or deleted depending on the setting in the "When deleting code" group, in the Synchronization settings dialog box.

You have now completed a full round-trip code engineering cycle with UModel.

© 2019-2025 Altova GmbH

3 UModel Graphical User Interface

The UModel graphical user interface consists of the main diagram pane, as well as several smaller helper windows where you can enter or view data. The diagram pane serves as a parent container for any diagram windows that are open. To cycle through all open diagram windows, press **Ctrl+Tab**.



UModel graphical user interface

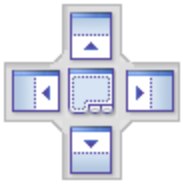
By default, the helper windows on the left side are docked in groups of three, and the Messages window appears below the diagram pane. You can, however, move and dock or undock any window as necessary. All windows can be searched using the **Find** combo box in the Main toolbar, or by pressing **Ctrl+F**. See also Finding and Replacing Text 114.

To dock or undock a window:

- Right-click its title bar, and select **Docking** (or **Floating**, respectively) from the context menu.

To move a window:

1. Click the window's title bar and drag to a new position. Several docking helpers appear.



2. Drag the window over a top, right, left, or bottom handle to dock it to the new position.

To reset all toolbars and windows to their default state:

- On the **Tools** menu, click **Restore toolbars and Windows**.

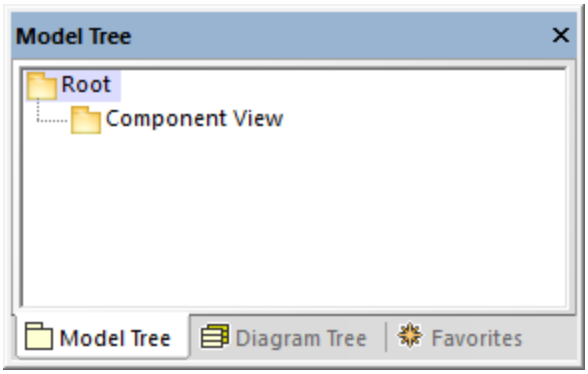
This chapter provides reference information about the parts that make up the UModel graphical user interface, as follows:

- Model Tree Window 83
- Diagram Tree Window 87
- Favorites Window 88
- Properties Window 89
- Styles Window 90
- Hierarchy Window 91
- Overview Window 93
- Documentation Window 94
- Layer Window 95
- Messages Window 96
- Diagram Window 98
- Diagram Pane 99

© 2019-2025 Altova GmbH

3.1 Model Tree Window

The Model Tree window enables you to view and manipulate all items (packages, classes, diagrams, relationships, and so on) in the UModel project.



Model Tree window

When you create a new UModel project, two packages are available by default, the "Root" and "Component View" packages. These two packages are the only ones that cannot be renamed or deleted. The "Root" package serves as starting point for modeling all other elements, while the "Component View" package is required for code engineering.

You can create additional packages, classes, diagrams, and their hierarchy either from this window or directly from a diagram, see Creating Elements 109. For additional operations that you can take against items in the Model Tree, see the How to Model... 108 chapter.

Note: UModel includes several example projects that you can explore in order to learn the modeling basics and the graphical user interface. These can be found at the following path: **C:\Users\<username>\Documents\Altova\UModel2025\UModelExamples**.

Showing, hiding, and sorting items in the Model Tree

To configure what should be displayed in the Model Tree window, as well as the sorting options, right-click inside the window, and then select the required menu option. To view all actions that can be taken against items displayed in the Model Tree window, right-click the item and observe the context menu options.

Collapsing and expanding items in the Model Tree

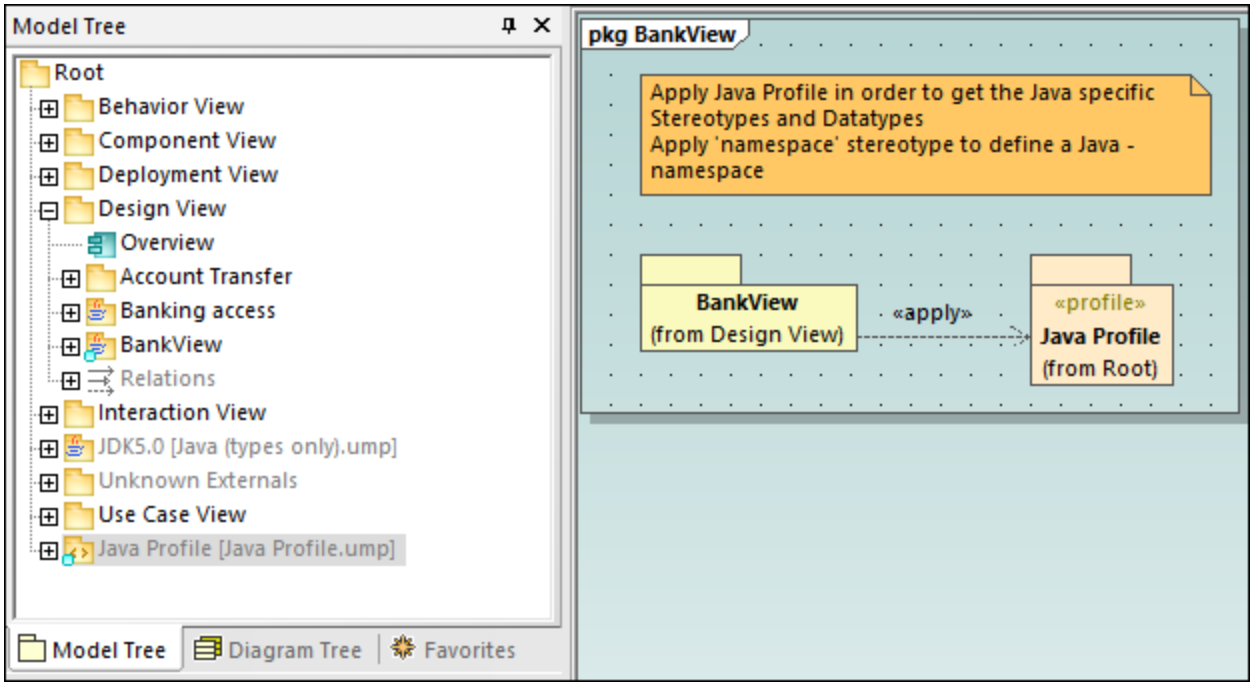
To expand items (for example, packages) in the Model Tree window:

- Press the * (asterisk) key to expand the current item and all child items
- Press the + (plus) key to expand the current item only.

To collapse the packages, press the - (dash) keyboard key. To collapse all items, click the "Root" package and press - (dash). Note that you can use both the standard keyboard keys and the numeric keypad keys to achieve this.

Identifying active diagram items

When a diagram is open in the Diagram pane, the Model Tree window shows some items with a light-blue dot at their base. These are items that are displayed in the active diagram (like "BankView" and "Java Profile" in the example below):



Icon reference

The Model Tree window may display a large number of icons which correspond to elements and diagrams in your project, the code engineering packages, as well as the imported profiles or subprojects. Specifically, it may display the following package types:

Icon	Description
	Standard UML Package
	Java namespace root package. Used to generate or reverse engineer Java code
	C# namespace root package. Used to generate or reverse engineer C# code
	Visual Basic namespace root package. Used to generate or reverse engineer VB.NET code
	XML Schema namespace root package. Used to generate XML schemas from the model, or import them into the model, see XML Schema Diagrams 424.
	Database namespace root package. Used to import databases into the model, and change their structure from the model, see UModel and Databases 486.

© 2019-2025 Altova GmbH

Icon	Description
	A namespace package (a package with the <<namespace>> stereotype applied to it)
	A UML profile

The diagrams that can appear in the Model Tree window are listed below.

Icon	Description
	Activity Diagram
	BPMN 1 (Business Process Modeling Notation) Business Process Diagram
	BPMN 2 Business Process Diagram
	BPMN 2 Choreography Diagram
	BPMN 2 Collaboration Diagram
	Class Diagram
	Communication Diagram
	Component Diagram
	Composite Structure Diagram
	Database Diagram
	Deployment Diagram
	Interaction Overview Diagram
	Object Diagram
	Package Diagram
	Profile Diagram
	Protocol State Machine Diagram
	Sequence Diagram
	State Machine Diagram
	SysML diagrams (9 diagram types)
	Timing Diagram
	Use Case Diagram
	XML Schema Diagram

Below are some examples of UML modeling elements that can appear in the Model Tree window. For more information about UML elements and the diagram types where they occur, see the UML Diagrams 296 chapter.

Icon	Description
	Class
	Property
	Operation
	Parameter
	Actor
	Use Case
	Component
	Node
	Artifact
	Interface
	Class Instance (Object)
	Class instance slot
	Relations
	Constraints

© 2019-2025 Altova GmbH

3.2 Diagram Tree Window

The Diagram Tree window displays any diagrams contained in the current UModel project.

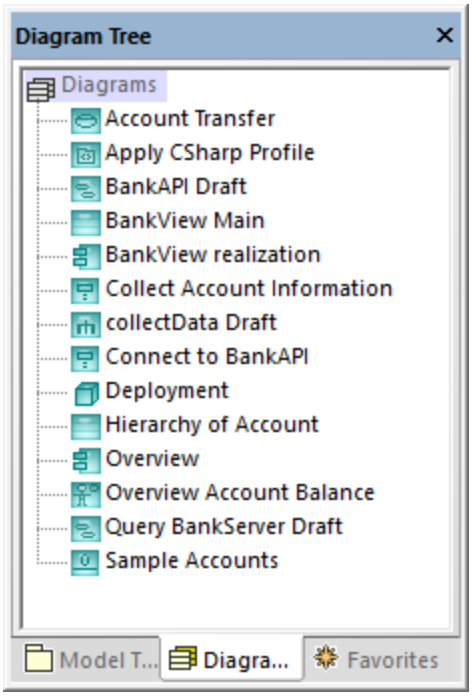


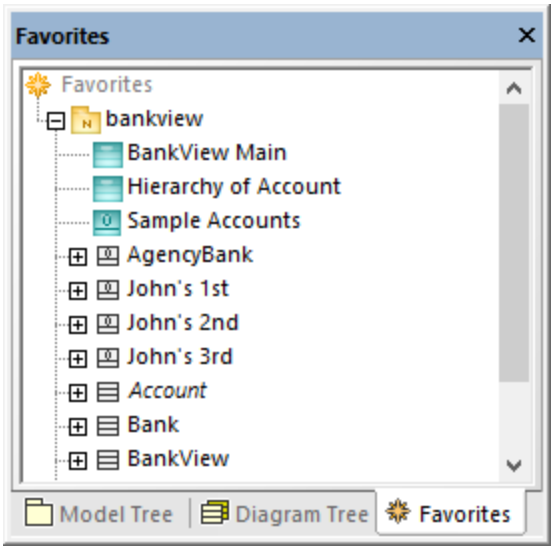
Diagram Tree window

Diagrams in this window can be shown either as an alphabetical list, or grouped by type. To change the display option, right-click in the window, and select or clear the **Group by Diagram type** option.

For instructions about creating, opening, and generating diagrams, including how to model their content, refer to the How to Model... 108 chapter. For specific information about each diagram type, refer to the UML Diagrams 296 chapter.

3.3 Favorites Window

The Favorites window displays any modeling elements or diagrams that you have added as favorites. "Favorites" represent a personal, custom-picked list of modeling elements or diagrams that you can use for quick access, for example.



Favorites window

By default, the contents of the Favorites window are automatically saved when you save the project. You can change this option from the **Tools | Options** menu, **File** tab. The relevant option name is **Load and save with project file | Favorites**.

Items in the Favorites window are not copies or clones; they represent the actual elements or diagrams. Most actions that you take in the Model Tree window are also applicable in the Favorites window, including adding or deleting elements. For more information, see the How to Model... 108 chapter.

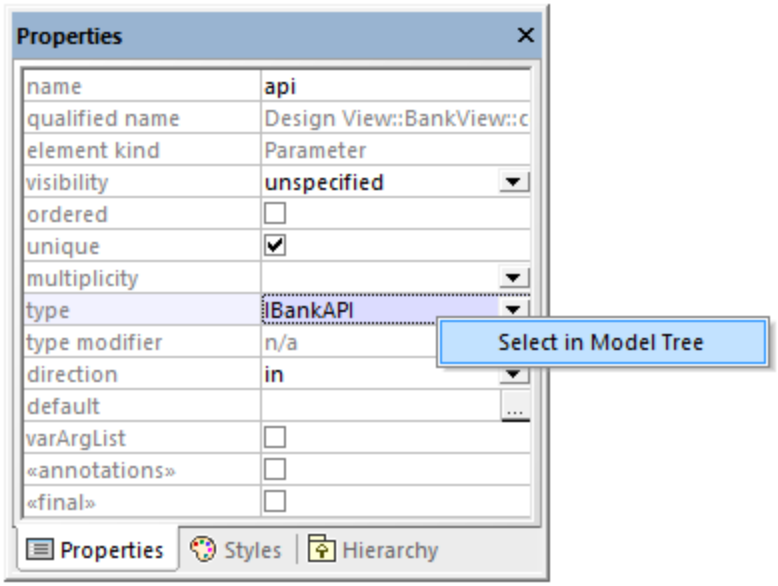
© 2019-2025 Altova GmbH

3.4 Properties Window

The Properties window shows information about an item that is currently selected (in focus). The "in focus" element can be an element selected in the Model Tree window (or other windows), an element selected on the diagram, or even a diagram itself.

The Properties window also enables you to change the properties of the currently selected element or relationship. The available properties depend on the kind of the element that is selected. There are properties which are read-only and grayed out (such as "element kind") and properties that you can modify (for example, "name").

If an operation or property takes a parameter, you can quickly jump to the respective parameter type in the Model Tree window, directly from the Properties window. To do this, right-click the "type" property of the parameter in the Properties window and select **Select in Model Tree** from the context menu. The same is applicable for return parameters.



Properties window

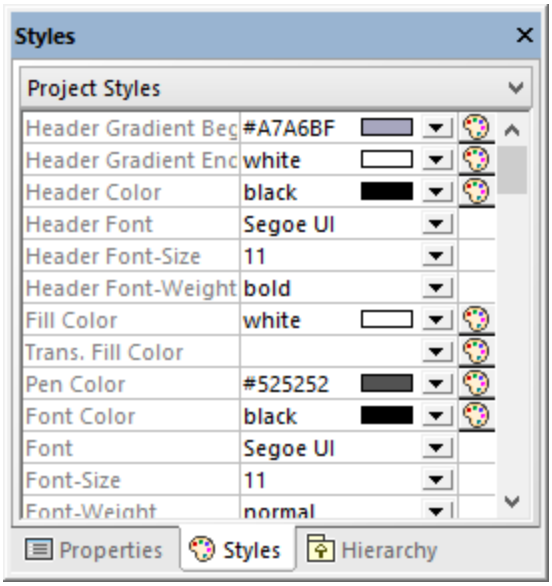
Changing a property of an element from the Properties window is immediately reflected in the diagram. Likewise, making a change in the diagram (for example, changing the visibility of an operation from `public` to `private`) affects the applicable property in the Properties window.

Properties that are enclosed within guillemets represent stereotypes (for example, `«final»`). You can add custom stereotypes to the project, in which case they would appear as properties in the Properties window, in addition to the default ones. For more information, see Example: Creating and Applying Stereotypes 416.

3.5 Styles Window

The Styles window enables you to view or change the visual appearance of diagrams or elements that are currently selected (in focus). The style attributes fall into two general groups:

- Formatting settings (for example, font size, weight, color, etc)
- Display settings (for example, show background color, grid, visibility settings, etc).



Styles window

Changing a property from the Styles window is immediately reflected in the user interface. Likewise, making a style change in another place (for example, setting the visibility of the diagram grid using the **Show Grid** toolbar button) affects the applicable property in the Styles window.

The Styles window has a dropdown list in the upper part, which enables you to select the level at which the style change is to be applied (for example, at individual element level, or at project level). For more information, see:

- Changing the Style of Elements 122
- Changing the Style of Diagrams 128
- Changing the Style of Lines and Relationships 137

© 2019-2025 Altova GmbH

3.6 Hierarchy Window

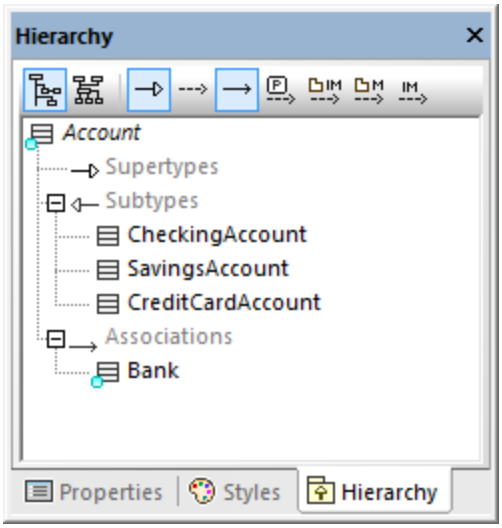
The Hierarchy window displays all relations of the currently selected modeling item, in two different views. The modeling element can be selected in a diagram, in the Model Tree window, or in the Favorites window.

Items in the Hierarchy window can be displayed in two views:

- Tree view
- Graph view

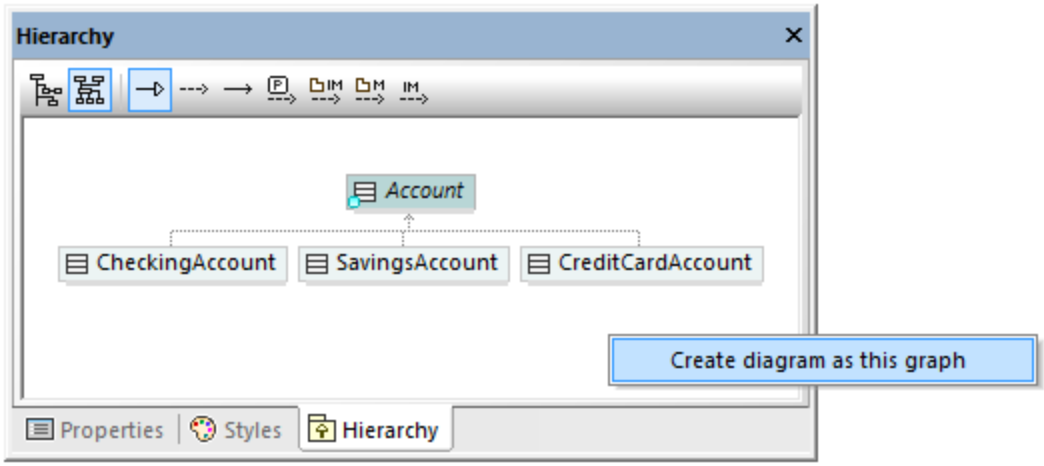
To switch between views, click the **Show tree view** or **Show graph view** buttons in the upper-left corner of the window.

The *tree view* shows multiple relations of the currently selected element, as a tree. Click the buttons at the top of the window to select types of relations that are to be shown. In the image below, only generalizations and associations are selected to be shown.



Hierarchy window (tree view)

The *graph view* shows a single set of relations in a hierarchical overview, as a diagram. In this view, only one of the relation buttons can be active at any one time. In the image below, the **Show Generalizations** button is currently active.



Hierarchy window (graph view)

In the graph view, you can generate diagrams that include the elements visible in the window. To do this, right-click inside the window, and select **Create diagram as this graph** from the context menu.

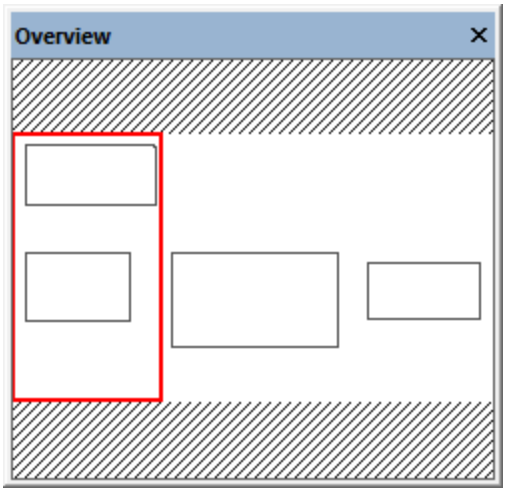
Settings pertaining to Hierarchy window can be changed using the menu option **Tools | Options | View**, in the **Hierarchy** group in the lower section of the dialog box.

The Hierarchy window is navigable: double-click one of the element icons, inside the window, to display the relations of that element. This applies both in the tree view and in the graph view.

© 2019-2025 Altova GmbH

3.7 Overview Window

The Overview window displays an outline view of the currently active diagram. This is especially handy when you need to scroll very large diagrams. To scroll the diagram, click and drag the red rectangle.

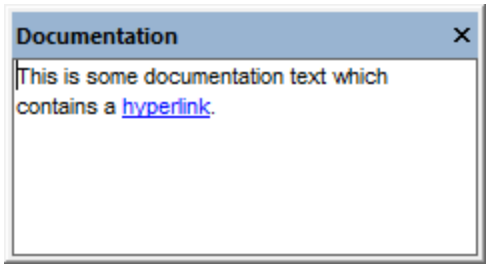


Overview window

See also Zooming into/out of Diagrams 135.

3.8 Documentation Window

The Documentation window enables you to document any of the UML elements available in the Model Tree window. To add documentation to an element, first click the element, and then enter text in the Documentation window. This window supports the standard editing shortcuts, including **Select All** (**Ctrl+A**), **Cut** (**Ctrl+X**), **Copy** (**Ctrl+C**) and **Paste** (**Ctrl+V**).



Documentation window

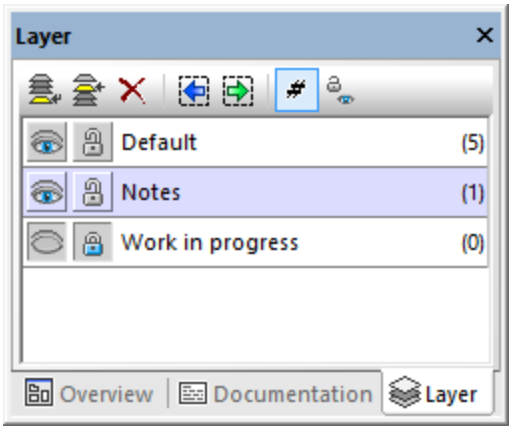
Text inside the Documentation window can be spell-checked. To do this, right-click inside the window, and select **Documentation Spelling** from the context menu.

Documentation text can also be exported as comments in the generated source code, or imported from source code comments during reverse engineering. For more information, see Documenting Elements 121.

 © 2019-2025 Altova GmbH

3.9 Layer Window

The Layer window enables you to define multiple layers for any UModel diagram. Layers allow you to make logical groupings of modeling elements on a diagram. For example, you can create, in addition to the default layer, some extra layers that would store notes with some internal information, or unfinished classes.

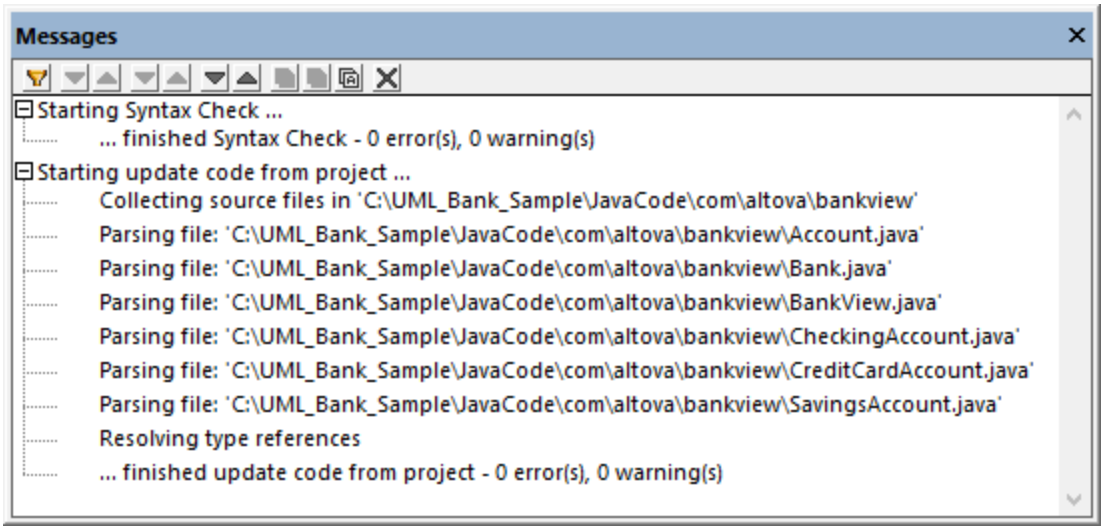


Layer window

For more information, see Adding Layers to Diagrams 132.

3.10 Messages Window

The Messages window displays any of the following message types: information messages, warnings, and errors. Such messages may occur when you check the project syntax (see Checking Project Syntax 173), or when you perform code engineering tasks. For more information about code engineering, see Generating Program Code 170 and Importing Source Code 192.



Messages window

The table below lists possible message types and their icons.

Icon	Description
none	Indicates an information message.
⚠	Indicates a warning message. Warnings are less critical than errors, but they may still prevent code from being imported or generated.
❗	Indicates an error message. When an error occurs, code generation or import fails.

The buttons available at the top of the Messages window enable you to take the following actions:

Icon	Description
	Filter messages by severity: information messages, and warnings. Select **Check All** to include all severity levels (this is the default behavior). Select **Uncheck All** to remove all severity levels from the filter.
	Jump to the next error.
	Jump to the previous error.
	Jump to the next warning.

© 2019-2025 Altova GmbH

Icon	Description
	Jump to the previous warning.
	Jump to the next line.
	Jump to the previous line.
	Copy the selected line to the clipboard.
	Copy the selected line to the clipboard, including any lines nested under it.
	Copy the full contents of the **Messages** window to the clipboard.
	Clear the Messages window.

When UModel runs as a Visual Studio or Eclipse plug-in, and parsing errors occur, you can quickly jump to the source code file where the error originates directly from the Messages window. To do this, click the parsing error in the Messages window. For more information, see UModel Plug-in for Visual Studio 570 and UModel Plug-in for Eclipse 581.

3.11 Diagram Window

Whenever you create a new diagram, or open an existing one, a new Diagram window is loaded in the Diagram Pane[99]. The diagram window provides the canvas (drawing area) where you design UML diagrams. Various modeling commands are available when you right-click either the diagram canvas itself, or any element on it.

Importantly, the toolbar buttons and the context menu commands in UModel change based on the type of diagram that is currently active (in focus). For example, if you click inside a Class diagram, the toolbar buttons will include only elements applicable to class diagrams. To view the diagram type, click inside an empty area in the diagram, and observe the "element kind" property displayed in the Properties window[89]. The diagram type can also be distinguished by the icon accompanying the diagram, see Creating Diagrams[124].

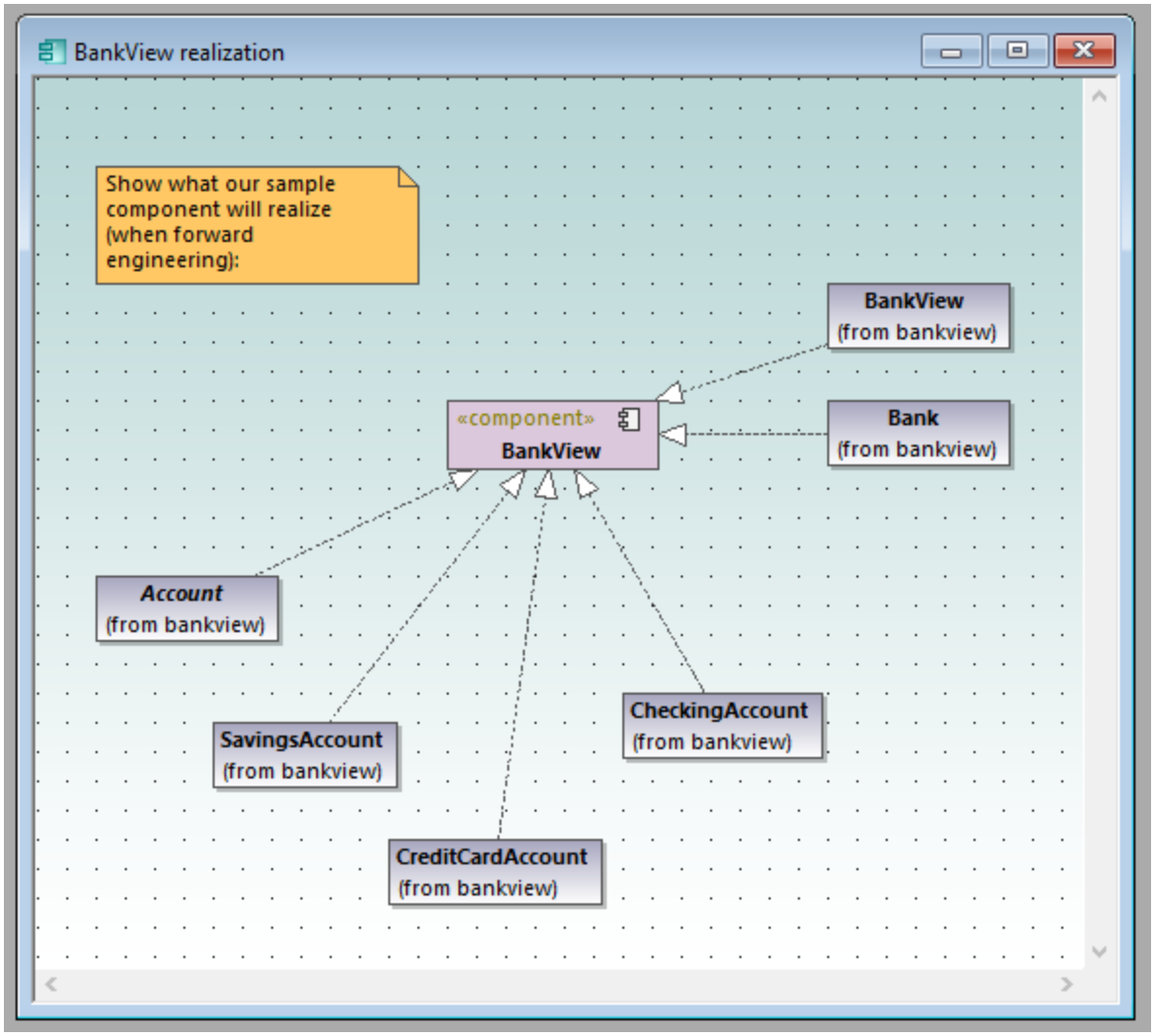


Diagram window

For information about creating new diagrams, opening existing ones, and manipulating elements inside the diagram, see the How to Model...[108] chapter.

© 2019-2025 Altova GmbH

3.12 Diagram Pane

The diagram pane hosts all diagram windows that are currently open. For information about creating new diagrams, opening existing ones, and manipulating elements inside the diagram, see the How to Model... 108 chapter.

The image below illustrates the diagram pane with four diagram windows open and positioned using the **Window | Cascade** menu command.

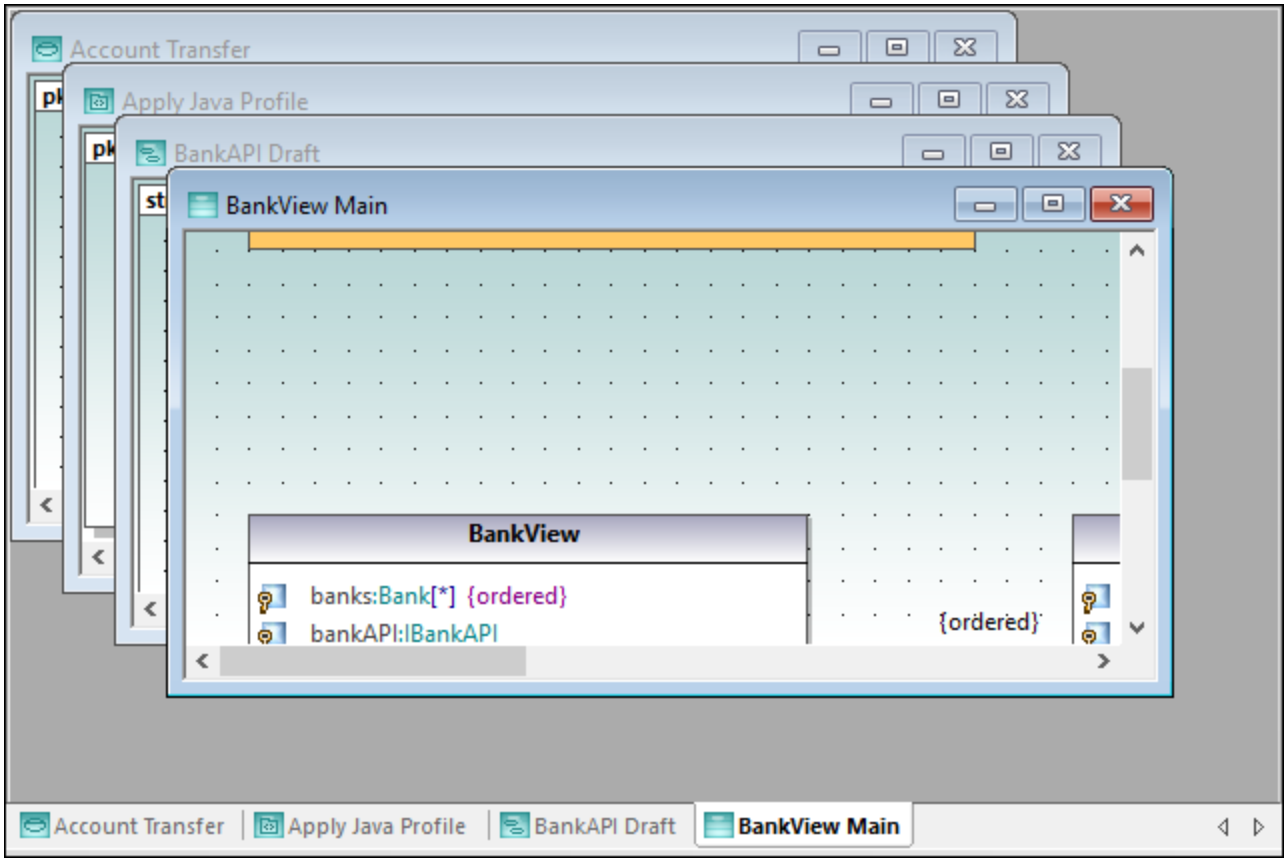
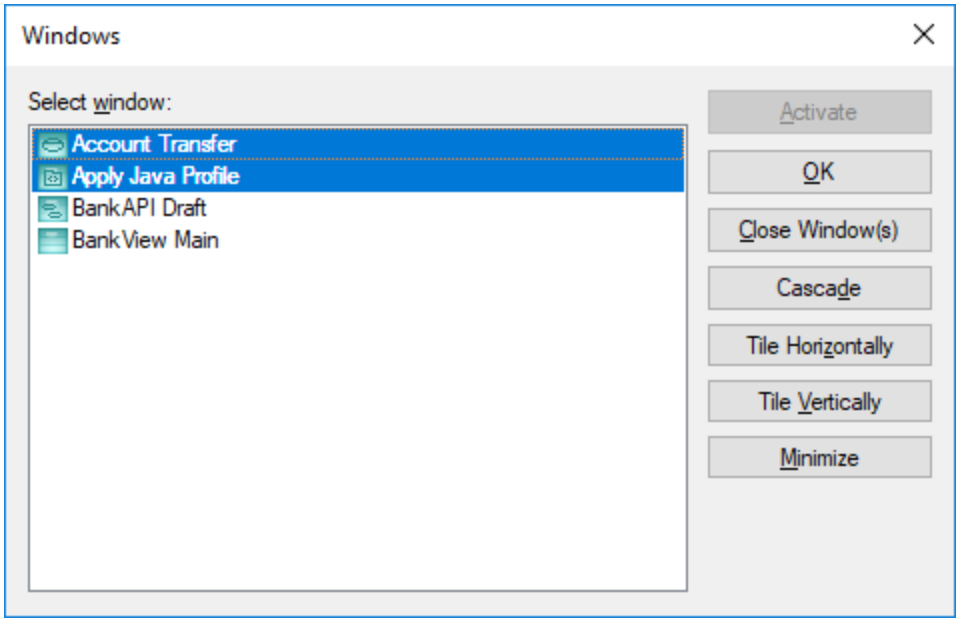


Diagram pane

Several commands applicable to the current diagram window are available when you right-click the corresponding window tab at the lower area of the diagram pane.

To apply miscellaneous commands to windows inside the diagram pane, use the commands available in the **Window** menu. Several window manipulation commands are also available on the Window dialog box (to open this dialog box, select the menu command **Window | Windows**).



Windows dialog box

To select multiple windows on the dialog box above, hold down the **Ctrl** key pressed and click the corresponding entries.

To cycle through all open diagram windows, press **Ctrl+Tab**.

© 2019-2025 Altova GmbH

4 UModel Command Line Interface

In addition to the graphical user interface, UModel also has a command line interface. To open the command line interface, run the **UModelBatch.exe** file available in the **C:\Program Files\Altova\UModel2025** directory. If you run UModel 32-bit on a 64-bit operating system, the path is **C:\Program Files (x86)\Altova\UModel2025**.

The command line parameter syntax is shown below, and can be displayed in the command prompt window by entering: `umodelbatch /?`

Note: If the path or file name contains spaces, enclose it in quotes, for example: "C:\Program Files\...\MyProject.ump".

```
usage: UModelBatch.exe [project] [options]

/? or /help ... display this help information

project     ... project file (*.ump)
/new[=file] ... create/save/save as new project, see Creating, Loading, and Saving Projects in Batch Mode 106
/set        ... set options permanent
/gui        ... display UModel user interface

commands (executed in given order):
/chk        ... check project syntax
/isd=path   ... import source directory
/isp=file   ... import source project file
                (*.project,*.xml,*.jpx,*.csproj,*.csdproj,*.vcxproj,*.vbproj,*.vbdproj
,*.sln,*.bdsproj)
/ibt=list   ... import binary types (specify binary[typenames] list)
                (';'=separator, '*'=all types, '#' before assembly names)
/ixd=path   ... import XML schema directory
/ixs=file   ... import XML schema file (*.xsd)
/m2c        ... update program code from model (export/forward engineer)
/c2m        ... update model from program code (import/reverse engineer)
/ixf=file   ... import XMI file
/exf=file   ... export to XMI file
/inc=file   ... include file
/mrg=file   ... merge file
/doc=file   ... write documentation to specified file
/lue[=cpri] ... list all elements not used on any diagram (i.e. unused)
/ldg        ... list all diagrams
/lcl        ... list all classes
/lsp        ... list all shared packages
/lip        ... list all included packages

options for save as new project:
/npad=opt   ... adjust relative file paths (Yes | No | MakeAbsolute)

options for import commands:
/iclg=lang  ... code language (Java1.4 | Java5.0 | Java6.0 | Java7.0 | Java8.0 |
Java9.0 |
```

```
                                Java10.0 | Java11.0 | Java12.0 | Java13.0 | Java14.0 |
Java15.0 |
                                C#1.2 | C#2.0 | C#3.0 | C#4.0 | C#5.0 | C#6.0 | C#7.0 |
C#7.1 | C#7.2 | C#7.3 | C#8.0 | C#9.0 |
                                VB7.1 | VB8.0 | VB9.0 |
                                C++98 | C++11 | C++14 | C++17)
/ipsd[=0|1] ... process sub directories (recursive)
/irpf[=0|1] ... import relative to UModel project file
/ijdc[=0|1] ... JavaDocs as Java comments
/icdc[=0|1] ... DocComments as C# comments
/icds[=lst] ... C# defined symbols
/ivdc[=0|1] ... DocComments as VB comments
/ivds[=lst] ... VB defined symbols (custom constants)
/icppdm[=lst] ... C++ defined macros
/icpphi[=0|1] ... read only C++ header files
/icpphc[=0|1] ... treat .h files a .cpp files
/icppms[=0|1] ... enable C++ Microsoft Compiler compatibility
/icppmv[=ver] ... MSVC version to use (1900 | 1800 | 1700 | 1600 | 1500 | 1400 | 1310
| 1300 | 1200)
/icppsy[=0|1] ... auto detect C++ system include files
/icppid[=lst] ... list of C++ include directories to use
/icppsd[=lst] ... list of C++ system include directories to use
/icppag[=arg] ... Additional C++ arguments for the compiler
/imrg[=0|1] ... synchronize merged
/iudf[=0|1] ... use directory filter
/iflt[=lst] ... directory filter (presets /iudf)

options for import binary types (after /iclg):
/ibrt=vers  ... runtime version
/ibpv=path  ... override of PATH variable for searching native code libraries
/ibro[=0|1] ... use reflection context only
/ibua[=0|1] ... use add referenced types with package filter
/ibar[=flt] ... add referenced types package filter (presets /ibua)
/ibot[=0|1] ... import only types
/ibuv[=0|1] ... use minimum visibility filter
/ibmv[=key] ... keyword of required minimum visibility (presets /ibuv)
/ibsa[=0|1] ... suppress attribute sections / annotation modifiers
/iboa[=0|1] ... create only one attribute per attribute section
/ibss[=0|1] ... suppress 'Attribute' suffix on attribute type names

options for diagram generation:
/dgen[=0|1] ... generate diagrams
/dopn[=0|1] ... open generated diagrams
/dsac[=0|1] ... show attributes compartment
/dsoc[=0|1] ... show operations compartment
/dscc[=0|1] ... show nested classifiers compartment
/dstv[=0|1] ... show tagged values
/dudp[=0|1] ... use .NET property compartment
/dspd[=0|1] ... show .NET property compartment

options for export commands:
/ejdc[=0|1] ... Java comments as JavaDocs
/ecdc[=0|1] ... C# comments as DocComments
/evdc[=0|1] ... VB comments as DocComments
/espl[=0|1] ... use user defined SPL templates
```


© 2019-2025 Altova GmbH

```
   /ecod[=0|1] ... comment out deleted
   /emrg[=0|1] ... synchronize merged
   /egfn[=0|1] ... generate missing file names
   /eusc[=0|1] ... use syntax check

   options for XMI export:
   /exid[=0|1] ... export UUIDs
   /exex[=0|1] ... export UModel specific extensions
   /exdg[=0|1] ... export diagrams (presets /exex)
    /exuv[=ver] ... UML version (UML2.0 | UML2.1.2 | UML2.2 | UML2.3 | UML2.4 | UML2.5 |
UML2.5.1)

   options for merge file:
   /mcan=file  ... common ancestor file

   options for documentation generation:
   /doof=fmt   ... output format (HTML | RTF | MSWORD | PDF)
   /dsps=file  ... SPS design file
```

Example 1: Import Java source code and preserve settings

The following command imports source code and creates a new project file. Notice that the project path contains spaces and is enclosed in quotes.

```
"C:\Program Files\Altova\UModel2025\UModelBatch.exe" /new="C:\My
Projects\Fred.ump" /isd="X:TestCases\UModel\Fred" /set /gui /iclg=Java8.0 /ipsd=1 /ijdc=1
/dgen=1 /dopn=1 /dmax=5 /chk
```

The meaning of all options is as follows:

`/new`	Specifies that the newly-created project file should be called "Fred.ump" in **C:\My Projects**
`/isd`	Specifies that the source directory should be **X:\TestCases\UModel\Fred**
`/set`	Specifies that any options used in the command line tool will be saved in the registry (When subsequently starting UModel, these settings become the default settings).
`/gui`	Display the UModel graphical user interface during batch processing.
`/iclg`	UModel will import the code as Java 8.0.
`/ipsd=1`	Recursively process all subdirectories of the root directory specified in the /isd parameter.
`/ijdc=1`	Create JavaDoc from comments where appropriate.
`/dgen=1`	Generate diagrams.
`/dopn=1`	Open generated diagrams.
`/chk`	Perform a syntax check.

Example 2: Synchronize code from the model

The following command updates code from an existing project file ("C:\UModel\Fred.ump").

```
"C:\Program Files\Altova\UModel2025\UModelBatch.exe" "C:
\UModel\Fred.ump" /m2c /ejdc=1 /ecod=1 /emrg=1 /egfn=1 /eusc=1
```

The meaning of all options is the same as in the previous examples, plus:

/m2c	Update the code from the model.
/ejdc	Comments in the project model should be generated as JavaDoc.
/ecod=1	Comment out any deleted code.
/emrg=1	Synchronize the merged code.
/egfn=1	Generate any missing file names in the project.
/eusc=1	Use the syntax check.

Example 3: Import Java binaries into the model

Let's assume that some Java binary .class files exist in the **C:\JavaProject\bin** directory, and you want to import these binaries into UModel. To do this, run the following command:

```
"<C:\Program Files\Altova\UModel2025\UModelBatch.exe>" /new="C:
\JavaProject\Result.ump" /ibt=*C:
\JavaProject\bin /iclg=Java8.0 /ibrt=JDK1.8.0_144 /dgen=1 /chk
```

The options used are as follows:

/new	Creates a new UModel project at the specified path.
/ibt	Instructs UModel to import binary types. The asterisk before the path indicates that all binary types at that path must be imported.
/iclg	Specifies the code generation language ("Java8.0", in this example).
/ibrt	Specifies the runtime environment ("JDK1.8.0_144" in this example). This is the same value that appears on the "Import Binary Types" dialog box in the "Runtime" drop-down list, see Importing Java, C# and VB.NET Binaries 204. You can also use a value like "jdk-10.0.1" as set in the JAVA_HOME environment variable. For C#, you can use the value /ibrt:any or otherwise values as they appear in the GUI in the "Runtime" drop-down list, making sure to omit any spaces. Examples: /ibrt:any /ibrt:.NET5 /ibrt:.NETFramework4.8(v4.8.3752)

© 2019-2025 Altova GmbH

	The option "any" is the same as selecting "any (use disassembler)" from the "Runtime" drop-down list and is the recommended option.
`/dgen=1`	Generate diagrams.
`/chk`	Perform a syntax check after import.

4.1 Creating, Loading, and Saving Projects in Batch Mode

When you run **UModelBatch.exe** with a command like `UModelBatch MyProject.ump`, you can use the following parameters:

`/new`	This parameter defines the path and file name of the new UModel project file (*.ump) to create. It can also be used to load an existing project and save it under a different name, for example: `UmodelBatch.exe MyFile.ump /new=MyBackupFile.ump`
`/set`	This parameter overwrites the current default settings in the registry with the options you specify.
`/gui`	This parameter displays the UModel graphical user interface (GUI) during the batch process.

The examples below illustrate how to create, load, or save projects in full batch mode (in other words, the `/gui` parameter is not set).

new

UModelBatch /new=xxx.ump (options)
creates a new project, executes options, xxx.ump is always saved (regardless of options)

auto save

UModelBatch xxx.ump (options)
loads project xxx.ump, executes options, xxx.ump is saved **only** if document has changed (like **/ibt**)

save

UModelBatch xxx.ump (options) /new
loads project xxx.ump, executes options, xxx.ump is **always** saved (regardless of options)

save as

UModelBatch xxx.ump (options) /new=yyy.ump
loads project xxx.ump, executes options, always saves xxx.ump as yyy.ump (regardless of options)

The examples below illustrate how to create, load, or save projects in batch mode with UModel user interface visible (the `/gui` parameter is set).

new

UModelBatch /gui /new (options)
creates a new project, executes options, nothing saved, the GUI is left open

save new

UModelBatch /gui /new=xxx.ump (options)
creates a new project, executes options, xxx.ump saved, the GUI is left open

user mode

UModelBatch /gui xxx.ump (options)
loads project xxx.ump, executes options, nothing saved, the GUI is left open

save

© 2019-2025 Altova GmbH

UModelBatch /gui xxx.ump (options) /new
loads project xxx.ump, executes options, xxx.ump is saved, the GUI is left open

save as

UModelBatch /gui xxx.ump (options) /new=yyy.ump
loads project xxx.ump, executes options, xxx.ump is saved as yyy.ump, the GUI is left open

The project will be saved successfully provided that no critical errors occur while executing the options.

5 How to Model...

Altova website: UML modeling

This chapter provides instructions for creating and manipulating UML elements, diagrams, and relationships from the UModel graphical user interface. It is intended as a "how to" guide to modeling with UModel. The enclosed instructions are generic across UModel and not specific to a particular element or diagram type, unless explicitly mentioned. For information applicable to (and grouped by) each diagram type, refer to the UML Diagrams 296 chapter.

The information in this chapter is organized into the following categories: Elements, Diagrams, Relationships, and Stereotypes.

Elements	Diagrams	Relationships	Stereotypes
Creating Elements 109	Creating Diagrams 124	Creating Relationships 136	Stereotypes and Tagged Values 146
Inserting Elements from the Model into a Diagram 110	Generating Diagrams 125	Changing the Style of Lines and Relationships 137	Tagged Values 147
Renaming, Moving, and Copying Elements 112	Opening Diagrams 127	Viewing Element Relationships 139	Applying Stereotypes 148
Deleting Elements 113	Deleting Diagrams 128	Associations 139	Showing or Hiding Tagged Values 150
Converting Elements 114	Changing the Style of Diagrams 128	Collection Associations 142	
Finding and Replacing Text 114	Aligning and Resizing Modeling Elements 130	Containment 145	
Checking Where and If Elements Are Used 116	Type Autocompletion in Classes 134		
Constraining Elements 117	Zooming into/out of Diagrams 135		
Hyperlinking Elements 118	Adding Layers to Diagrams 132		
Documenting Elements 121			
Changing the Style of Elements 122			

Note: UModel includes several example projects that you can explore in order to learn the modeling basics and the graphical user interface. These can be found at the following path: **C:\Users\<username>\Documents\Altova\UModel2025\UModelExamples**.

© 2019-2025 Altova GmbH

5.1 Elements

5.1.1 Creating Elements

With UModel, new elements can be created as follows:

- From the Model Tree 83 window. With this approach, elements are added to the model only, and you can insert them later into diagrams if necessary.
- From any diagram window. Any elements added to a diagram are also automatically added to the model as well. Should you need to delete an element later, you can choose whether it should be removed from the diagram only, or deleted from the model as well.

To add elements from the Model Tree window:

- In the Model Tree 83 window (or Favorites 88 window), right-click the element (for example, package) under which you want the new element to appear, and select **New Element | <Element Name>** from the context menu. For example, to add a new package under the "Root" package, right-click the "Root" package, and select **New Element | Package**.

To add elements from the Diagram window:

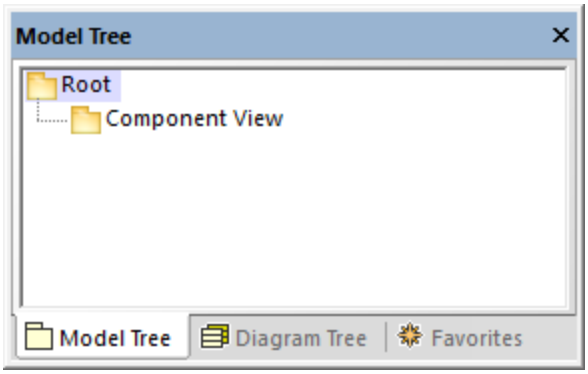
1. Create a new diagram (see Creating Diagrams 124) or open an existing one (see Opening Diagrams 127).
2. Do one of the following:
   a. Right-click inside the diagram and select **New | <Element Name>** from the context menu.
   b. Click the toolbar button of the element you wish to add, and then click inside the diagram. To insert multiple elements of the same type, hold down the **Ctrl** key before clicking inside the diagram.

Packages

As you model elements, you will likely need to work with packages more often than with other elements. Each entry marked with a folder symbol in the Model Tree window represents a UML package. Packages in UModel serve as containers for all other UML modeling elements (including diagrams, classes, and so on) and have the following behavior:

- They can be created at any position in the Model Tree.
- They can be moved or copied to other packages (as well as into valid model diagrams), see Renaming, Moving, and Copying Elements 112.
- They can be used as source or target elements when code is generated or synchronized with the model, see Forward Engineering (from Model to Code) 64 and Reverse Engineering (from Code to Model) 73.

When you create a new UModel project, two packages are available by default, the "Root" and "Component View" packages. These two packages are the only ones that cannot be renamed or deleted. The "Root" package serves as starting point for modeling all other elements, while the "Component View" package is required for code engineering.



Default UModel packages

5.1.2 Inserting Elements from the Model into a Diagram

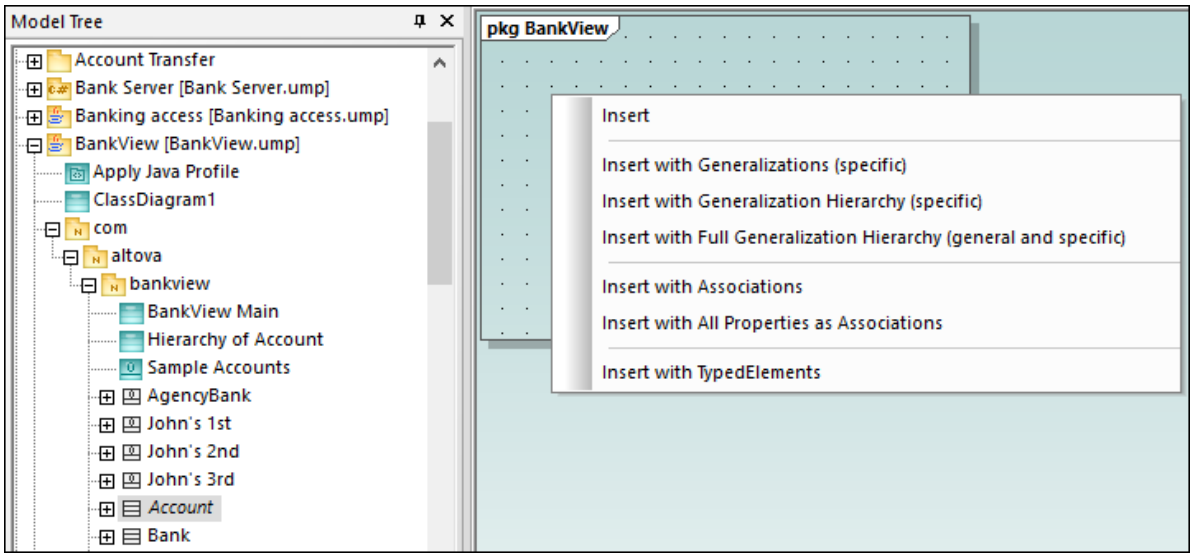
Elements present in the model can be inserted into a diagram either individually or as a group. To select multiple elements from the Model Tree window, hold down the **Ctrl** key while clicking each item. There are two ways to insert elements into a diagram: drag left, and drag right.

- **Drag left** (holding down the left mouse button and releasing it in the diagram) inserts elements immediately at the cursor position. In this case, any associations, dependencies etc. that exist between the currently inserted elements and the new one, are automatically displayed.
- **Drag right** (holding down the right mouse button and releasing it in the diagram) opens a context menu from which you can select the specific associations, generalizations you want to display.

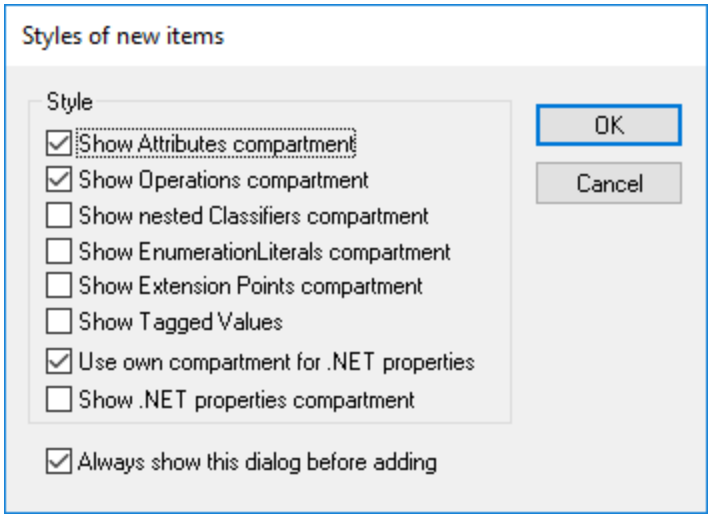
For example, let's suppose that you want to create a new class diagram from a class that already exists in the model. To illustrate this scenario, open the sample project **Bank_MultiLanguage.ump** available at the following path: **C:\Users\<username>\Documents\Altova\UModel2025\UModelExamples**. Assuming that you want to replicate the "Account Hierarchy" diagram in a new class diagram, do the following:

1. Right-click the **bankview** package and select **New Diagram | Class Diagram**.
2. Locate the abstract `Account` class in the model tree, and use **drag right** to place it in the new diagram. For this example, we would like to display the class together with its derived classes. To achieve this, select **Insert with Generalization Hierarchy (specific)** from the context menu.

© 2019-2025 Altova GmbH



3. Select or clear the check boxes for specific items you want to appear in the diagram.



4. Click OK. The `Account` class, together with its three subclasses, is inserted into the diagram. The Generalization arrows are also automatically displayed. To automatically arrange the classes inside the diagram, run the menu command **Layout | Autolayout All | Hierarchic**.

If you had selected the **Insert** command instead of **Insert with Generalization Hierarchy (specific)**, the class would have been added to the diagram without any derived classes. Note that you can still display the generalization hierarchy later, as follows:

- Right-click the `Account` class in the diagram and select **Show | Generalization hierarchy** from the context menu. As a result, the derived classes are inserted into the diagram as well.

5.1.3 Renaming, Moving, and Copying Elements

You can cut, copy, rename and move elements in the Model Tree 83 window and inside diagrams of the same type. These actions may also be possible across diagrams of different type if applicable. You can also copy or move elements from the Model Tree window into a diagram, provided that the diagram is allowed to contain the corresponding element according to the UML specification.

To rename an element:

- Double-click the element name and edit it.
- Alternatively, click the element and press **F2**.

The procedures above apply regardless of the window in which the element is displayed, including the Model Tree window, Properties window, and the Diagram window.

> The "Root" and "Component View" packages are displayed at all times in the Model Tree window and cannot be renamed or deleted.

To copy or move elements:

- Use the standard Windows commands **Cut**, **Copy**, or **Paste**. These commands can be triggered from keyboard shortcuts (**Ctrl+X**, **Ctrl+C**, **Ctrl+V**, respectively), from the corresponding toolbar buttons, as well as from the **Edit** menu.
- Alternatively, drag an element to a destination package (or element). Dragging an element moves it. Holding down the **Ctrl** key and dragging an element creates a copy of it.

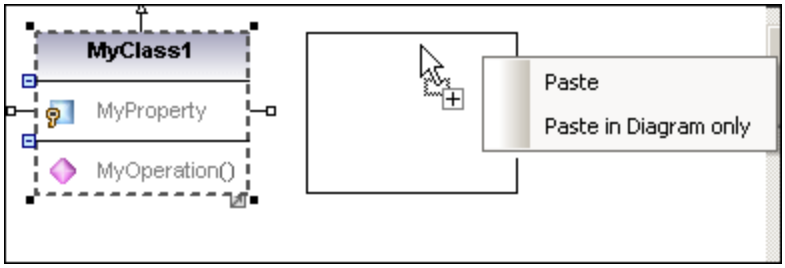
For example, in a diagram, you can move a class member to another class by dragging it from the source class to the destination class. To copy the class member rather than moving it, first select it, and then drag it to the destination class while holding down the **Ctrl** key.

If you paste a class into the same package, the new class is created with a sequential number appended to the end, for example, "MyClass1". Likewise, if you paste a property inside the same class, the new property is created with a sequential number appended to the end, for example, "MyProperty1". The same applies for other class members, such as operations and enumerations. The same logic is also applicable when you paste elements in the same diagram, provided that the diagram belongs to the same package as the elements that are being pasted.

If you paste a class into a different package, the new class will have the same name as the original class. The same logic applies when you copy class members (such as properties, operations, and so on) to a different class.

By default, any element that is pasted into a diagram is automatically added to the model as well (and thus is visible in the Model Tree window). However, you can also copy and paste an element into the current diagram only, without adding it to the model. To do this, first copy the element, right-click on the diagram, and then select **Paste in Diagram only** from the context menu. The **Paste in Diagram only** command also appears when you drag an existing item into the same diagram while holding the **Ctrl** key pressed.

© 2019-2025 Altova GmbH



In the example above, **Paste** will create the new class in the diagram and add it to the model as well, while **Paste in Diagram only** will only display a second view of it on the diagram. Note that copies created using the second approach are merely additional views of the original element and link to it; they are not standalone copies. (For example, renaming a property in the duplicated class will automatically apply the same change to the original class.)

5.1.4 Deleting Elements

Elements can be deleted in one of the following ways:

- From the Model Tree window. Use this approach if the element should be deleted from the project as well as any diagrams where it is present.
- Directly from diagrams where they occur. In this case, you can choose whether the element should be removed from the diagram only, or deleted from the model (project) as well.

To delete elements from the project and all related diagrams (approach 1):

1. In the Model Tree window, click the element you want to delete. Hold the **Ctrl** key down to select multiple elements.
2. Press **Delete**.

To delete elements from the project and all related diagrams (approach 2):

1. Open a diagram and click the element you want to delete. Hold the **Ctrl** key down to select multiple elements.
2. Press **Delete**. A dialog box appears asking to confirm that you want to delete the element both from the project and the diagram.
3. Click **Yes**. The element is deleted both from the diagram and the project.

To delete elements from the diagram but not from the project:

1. Open a diagram and click the element(s) you want to remove. Hold the **Ctrl** key down to select multiple elements.
2. Hold down the **Ctrl** key and press **Delete**. The elements are deleted from the diagram but still kept in the project.

Before you delete elements from a project, you may want to check if they are used in any diagrams.

- Right-click an element in the Model Tree, and then select **Show element in all diagrams** from the context menu.

Likewise, when a diagram is open, you can quickly select an element in the Model Tree, as follows:

- Right-click the element on the diagram, and select **Select in Model Tree** from the context menu.
- Alternatively, click the element on the diagram and press **F4**.

5.1.5 Converting Elements

Some of the elements support quick conversion to some other element kind. This action may be useful, for example, if you started designing a class but would like to change it later to an interface, or vice versa. More specifically, the following kinds of elements support conversion to any other item in the list:

- Class
- Interface
- Enumeration
- PrimitiveType
- DataType

You can convert the element kinds listed above either from the Diagram window[87] or from the Model Tree[83].

To convert elements:

1. Open a diagram that includes classes, interfaces, enumerations, primitive types or data types (for example, a class diagram). Alternatively, locate any of these element kinds in the Model Tree.
2. Right-click the element of interest (for example, a class) and select **Convert To | <element kind>** from the context menu.

After conversion, the name of the element is preserved. If possible, the data associated with the element is also preserved. For example, a conversion from interface to class or from class to interface preserves data such as properties or operations. However, a conversion from a class or interface to an enumeration will result in data loss. In such cases, if necessary, you can restore the previous state of the element by running the **Undo** (**Ctrl+Z**) command.

5.1.6 Finding and Replacing Text

You can search for modeling elements, diagrams, text, and so on, inside any of following windows:

- Diagram window
- Model Tree window
- Diagram Tree window
- Favorites window
- Documentation window
- Messages window

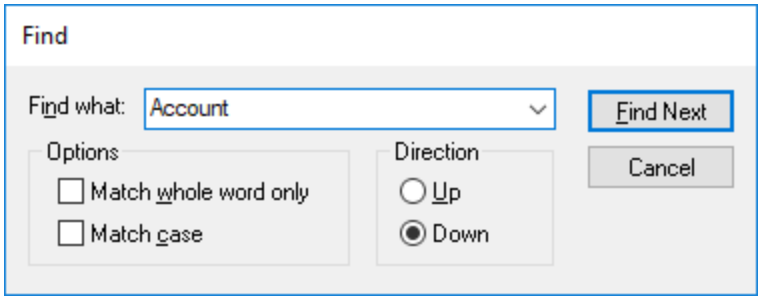
The search scope is applicable to the window where the cursor is currently placed. Therefore, if you want to search for text inside a diagram, for example, click inside the diagram first. Likewise, if you want to search for an item in the UModel project, click inside the Model Tree window first.

© 2019-2025 Altova GmbH

To search for text or elements:

1. Click inside the window where you want to find text.
2. Do one of the following:
   a. Type the search text in the text box of the main toolbar, and then click **Find Next** or press **F3**. To go to the previous occurrence, press **Shift+F3**.

   b. On **Edit** menu, click **Find** (or press **Ctrl+F**).



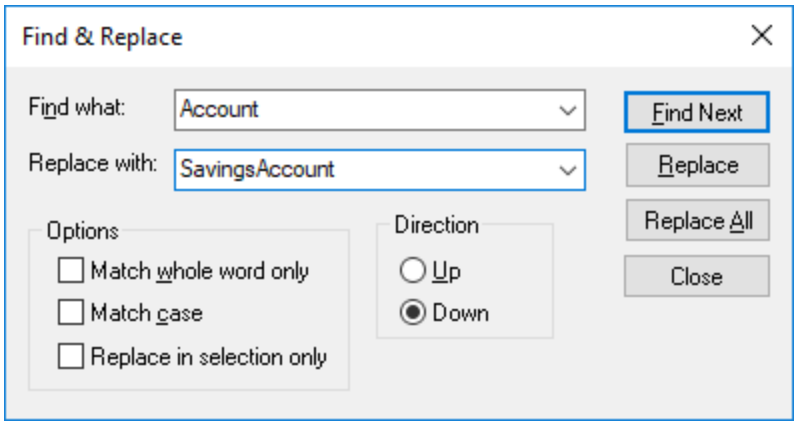
Find and replace

You can also find and replace text (for example, in order to quickly rename modeling elements). When the element is found, it is highlighted in the diagram as well as in the Model Tree. Search and replace works in the following windows:

- Diagram window
- Model Tree window
- Diagram Tree window
- Favorites window
- Documentation window

To find and replace text:

1. Click inside the window where you want to find/replace text.
2. Do one of the following:
   c. Click the **Replace** toolbar button.
   d. On the **Edit** menu, click **Replace** (or press **Ctrl+H**).



5.1.7 Checking Where and If Elements Are Used

While navigating the elements in the Model Tree, you might want to see where, or if, the element is actually present in a model diagram. To find where elements are used, do one of the following:

- Right-click the element in the Model Tree window, and select **Show element in all diagrams** (or, if a diagram is currently open, **Show element in active diagram**).

You can also find elements not used in any diagram either for the entire project, or for individual packages.

To find unused elements in the entire project:

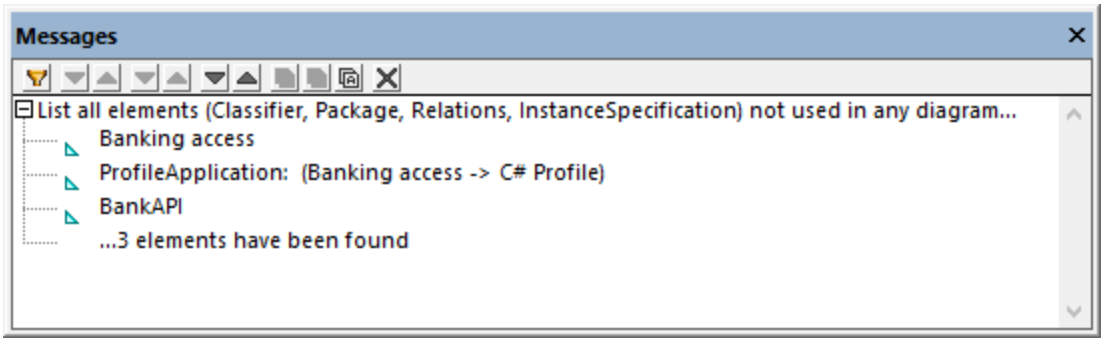
- On the **Project** menu, click **List elements not used in any diagram**.

To find unused elements for a specific package:

- Right-click the package you would like to inspect, and select **List elements not used in any diagram.**

A list of unused elements appears in the Messages window. Note that the unused elements are displayed for the currently selected package and its subpackages. Items inside parentheses are elements which have been configured to appear in the unused list, from **Tools | Options | View** tab.

© 2019-2025 Altova GmbH



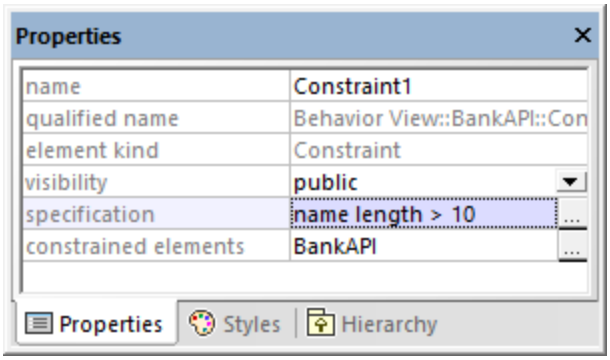
Click the element name in the Messages window to locate it in the Model Tree.

5.1.8 Constraining Elements

Constraints can be defined for most model elements in UModel. Note that constraints are not checked by the syntax checker, because they are not part of the code generation process.

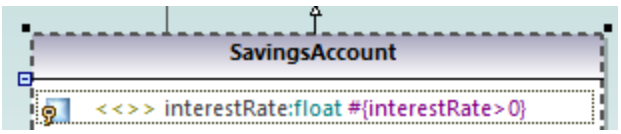
To constrain an element (from the Model Tree):

1. Right-click the element you want to constrain, and select **New Element | Constraints | Constraint**.
2. Enter the name of constraint and press **Enter**.
3. Type the constraint text in the "specification" field of the Properties window (for example, `name length > 10`).



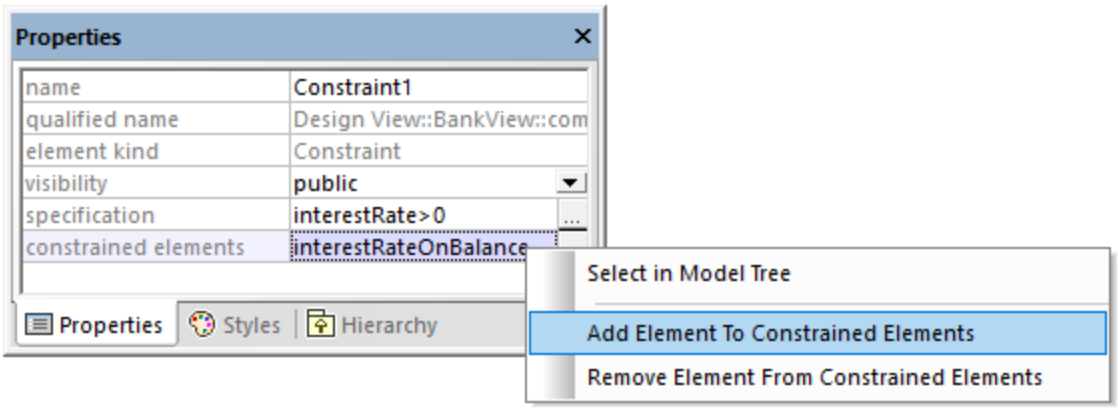
To constrain an element (from a diagram):

1. Double-click the specific element to be able to edit it.
1. Type "#", and then type the constraint text inside curly braces, for example, `#{interestRate >=0}`.

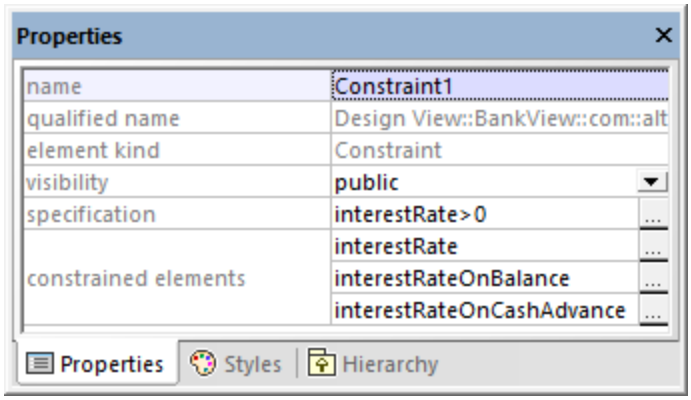


To assign constraints to multiple modeling elements:

1. Select a constraint in the Model Tree window.
2. Right-click the "constrained elements" property the Properties window, and select **Add element to constrained elements**.



3. Select the specific element you want to assign the current constraint to. Hold down the **Ctrl** key to select multiple elements.



The "constrained elements" field contains the names of the modeling elements it has been assigned to. For example, in the image above, `Constraint1` has been assigned to the following properties: `interestRate`, `interestRateOnBalance`, `interestRateOnCashAdvance`.

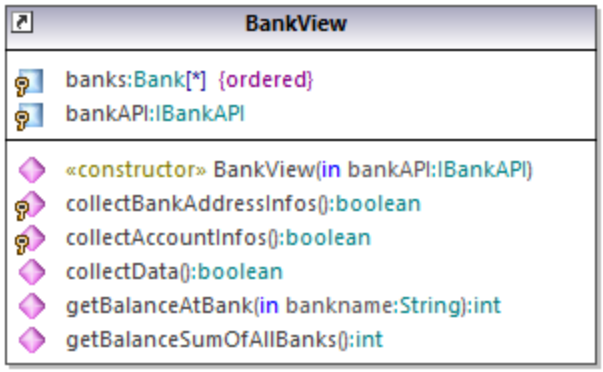
5.1.9 Hyperlinking Elements

You can manually create hyperlinks between most modeling elements (except lines) and any of the following:

- Other elements (either on the diagram or in the Model Tree)
- Diagrams
- Files external to the project (for example, PDF, Word, or Excel documents, graphics files, and so on)
- Web pages

© 2019-2025 Altova GmbH

A single element can have one or more hyperlinks of any of the kinds mentioned above. In a diagram, elements that contain hyperlinks can be easily recognized by the hyperlink icon that is visible next to them (either in the right or left corner). To open the hyperlink target, right-click the hyperlink icon on the element and select the target. If there is only one hyperlink defined, you can also click and access the target directly.



Class containing hyperlinks

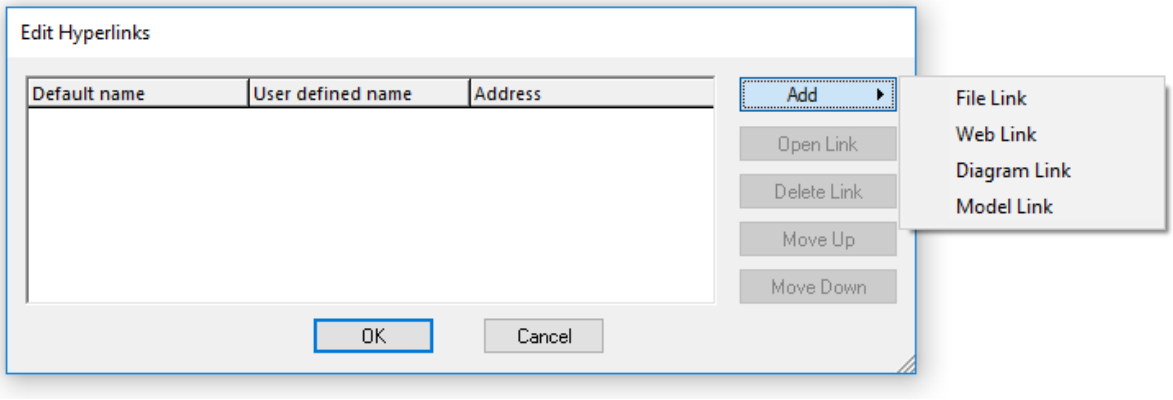
Tip: As you navigate through the UModel graphical user interface, either with or without hyperlinks, you can easily go back and forward between views by clicking the **Back** or **Forward** toolbar buttons, respectively.

You can automatically generate hyperlinks between dependent packages and diagrams when importing source code or binary files into a model, provided that you selected the specific settings on the import dialog box. For more information, see Importing Source Code 192 and Importing Java, C# and VB.NET Binaries 204. Also, when you generate UML documentation from the project, you can choose whether to include hyperlinks in the generated output, see Generating UML documentation 285.

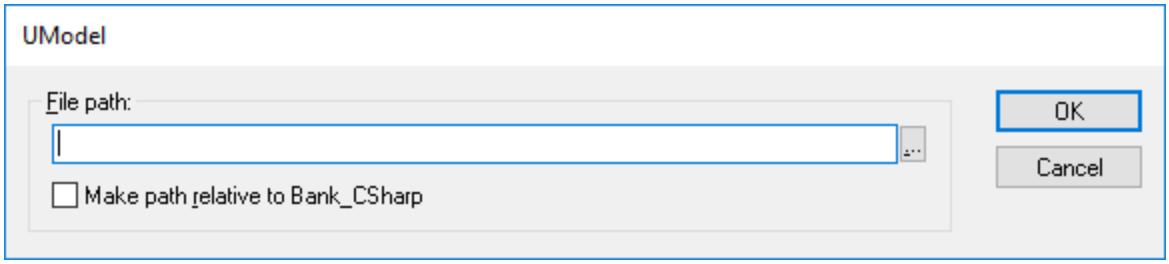
You can create hyperlinks not only from elements that appear in the diagram or in the Model Tree window, but also from text within notes, as well as text in the Documentation window, as shown in the instructions below.

To create a hyperlink from an element:

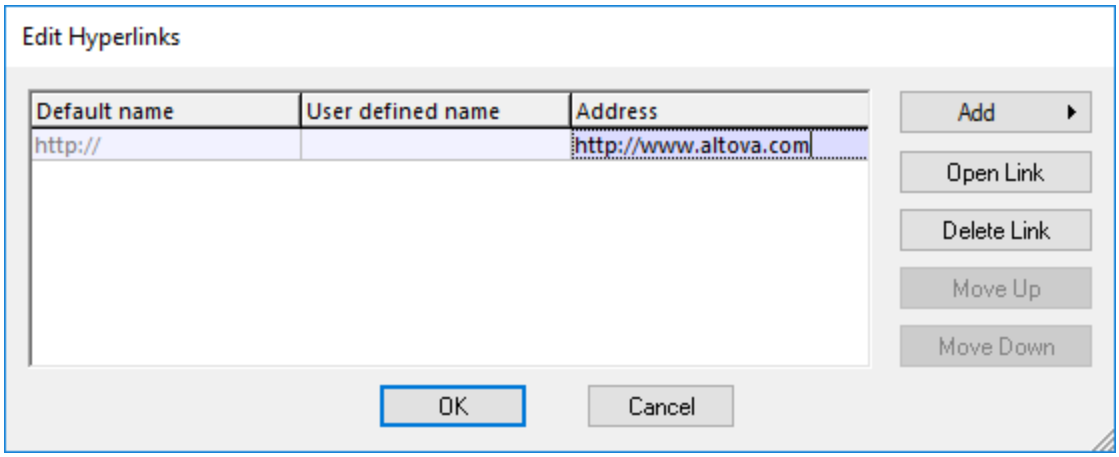
1. Right-click an element on a diagram or in the Model Tree window, and select **Hyperlinks | Insert/Edit Hyperlinks** from the context menu.
2. Click **Add**, and select a hyperlink kind (element, diagram, file, or a Web link).



3. Do one of the following:
   - To create a diagram or hyperlink, select the target element or diagram when prompted.
   - To create a file hyperlink, click the Ellipsis button and browse for the target file.



   - To create a Web link, type the target address in the "Address" column of the dialog box, for example:



4. Optionally, enter a custom link name in the "User defined name" column. If defined, this custom name will be displayed in the UModel's graphical interface instead of the target path (or address).

To create a hyperlink inside a note:

- Select some text inside the note, right-click it and then select **Insert/Edit Hyperlinks** from the context menu. The same instructions apply for text in the Documentation window.

© 2019-2025 Altova GmbH

To change or remove a hyperlink:

- Right-click the hyperlink icon on the element (or the hyperlinked text), and use the appropriate command in the "Edit Hyperlinks" dialog box.

5.1.10 Documenting Elements

You can add documentation comments to modeling elements as follows:

- Click the element (either in the diagram or in the Model Tree window).
- Enter text in the Documentation window.

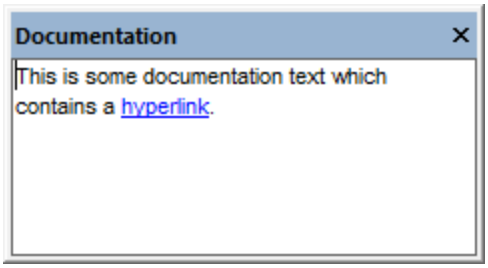
Any documentation text will be saved together with the project.

When an element is selected, its documentation is visible at all times in the Documentation window, if available. You can also display documentation as a comment on the diagram, as follows:

- Right-click the element on the diagram, and select **Show | Annotating Comments** from the context menu.

Documentation hyperlinks

To create a hyperlink inside the Documentation window, select some text inside the window, right-click it and then select **Insert/Edit Hyperlinks** from the context menu. The hyperlink target can be a Web site, a diagram, a file, or another element, see also Hyperlinking Elements 118.



Documentation window

Code generation and documentation comments

If you generate code from class diagrams, any comments applied to classes and their members (in class diagrams) can be exported to the generated code as well. To do this, select the check box **Write Documentation as Java Docs** (for Java) or **Write Documentation as DocComments** (for C#, VB.NET) before generating program code, see also Code Generation Options 175.

Likewise, if you reverse engineer program code into a model, the code comments can be imported into the model. To do this, select the check box **JavaDocs as Documentation** (for Java) or **DocComments as Documentation** (for C#, VB.NET) before reverse engineering program code, see also Code Import Options 194.

For information about how comments in program code (or XML schemas) map to UModel comments, refer to the mapping tables for each language:

- C# Mappings 224
- VB.NET Mappings 244
- Java Mappings 258
- XML Schema Mappings 264

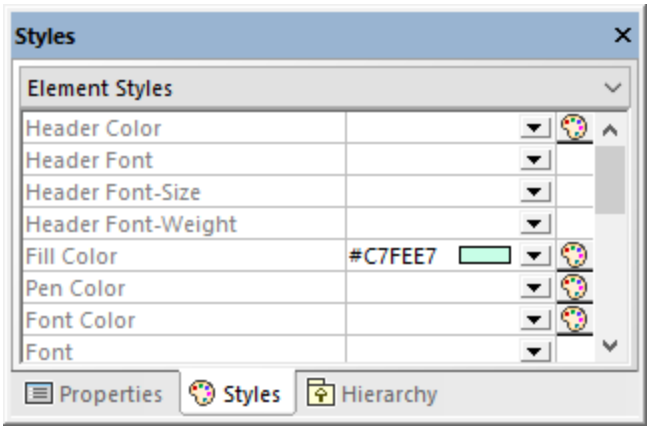
5.1.11 Changing the Style of Elements

You can change the appearance (style) of modeling elements, including their color, font size, font weight, background color, line thickness, and others. The appearance of elements can be changed at various levels: globally for all elements in the project, selectively for all elements of the same family (for example, classes), or for each individual element. For information about changing the style of the diagram itself, see Changing the Style of Diagrams 128.

If you would like to use custom images instead of conventional element representations in diagrams, this is possible by extending your project with custom profiles and stereotypes. For more information, see Example: Customizing Icons and Styles 421.

To change the appearance of elements:

1. Click the element on a diagram.
2. Notice the dropdown list at the top of the Styles Window and do one of the following as applicable:
   a. To edit the properties of the current element only, select "Element Styles" from the list.



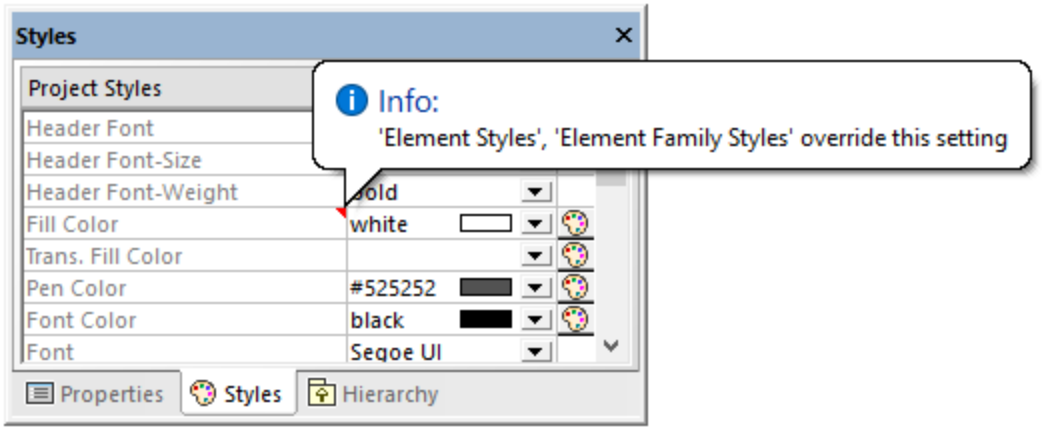
   b. To edit the properties of all elements of the same kind (for example, classes), select "Element Family Styles" from the list.
   c. To edit the properties of all elements globally at the project level, select "Project Styles".
   d. To edit the properties of all lines in the project, including association, dependency, and realization lines, select "Line Styles". (This value is only visible if the currently selected element is a line.)
   e. To edit the properties of all elements that are not lines (the so-called "nodes") across the project, select "Node Styles". (This value is only visible if the currently selected element is not a line.)

 © 2019-2025 Altova GmbH

3. Change the value of the required property (for example, "Fill Color").

A more specific style overrides a more generic style. That is, styles applied at individual element level override those applied at element family level. Likewise, styles applied at element family level override those applied at project level.

When a style is overridden, a small red triangle appears in the upper-right corner of the overridden property. Move the cursor over the triangle to display a tooltip with information about style precedence.



Overridden element style

5.2 Diagrams

5.2.1 Creating Diagrams

Diagrams represent visually how modeling elements interact, what is their structure, dependencies, hierarchy, and so on. Diagrams must belong to a package in the project, and therefore must be created under an existing package in the Model Tree window. You can move diagrams from one package to another at any time, by dragging them into a destination package.

To create a new diagram:

1. Right-click a package in the Model Tree window[83].
2. Select **New Diagram | <Diagram Kind>**.

You can also create a new diagram from the Diagram Tree window[87], as follows:

1. Right-click the root node ("Diagrams") in the Diagram Tree window.
2. Select a package where the diagram should belong, and click **OK**.

When the diagram window is active, the toolbars display only modeling elements applicable to the current diagram kind. The diagram kind is displayed in the Properties window after you click an empty area of the diagram. In addition to this, the following icons depict the diagram kind.

Icon	Description
	Activity Diagram
	BPMN 1 (Business Process Modeling Notation) Business Process Diagram
	BPMN 2 Business Process Diagram
	BPMN 2 Choreography Diagram
	BPMN 2 Collaboration Diagram
	Class Diagram
	Communication Diagram
	Component Diagram
	Composite Structure Diagram
	Database Diagram
	Deployment Diagram
	Interaction Overview Diagram
	Object Diagram

© 2019-2025 Altova GmbH

Icon	Description
	Package Diagram
	Profile Diagram
	Protocol State Machine Diagram
	Sequence Diagram
	State Machine Diagram
	SysML diagrams (9 diagram types)
	Timing Diagram
	Use Case Diagram
	XML Schema Diagram

5.2.2 Generating Diagrams

In addition to creating diagrams from scratch, you can also generate certain diagrams automatically from existing modeling elements or from program code. This topic shows you how to generate diagrams from existing modeling elements. For information about how to generate diagrams from source code, see:

- Generating Class Diagrams 399
- Generating Sequence Diagrams from Source Code 366
- Generating Package Diagrams While Importing Code or Binaries 408

To generate diagrams from existing elements, right-click an element (for example, package) in the Model Tree, and then select **Show in new diagram | <option>** from the context menu. Below are some examples:

To create a diagram which shows the contents of an existing package:

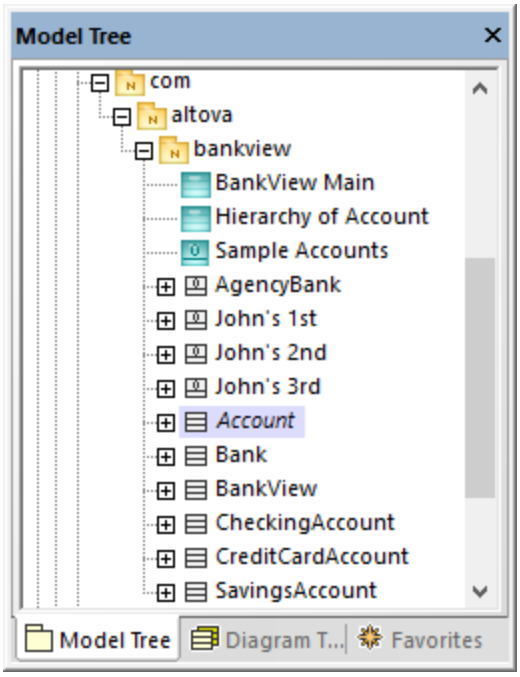
- Right-click a package in the Model Tree window and select **Show in new Diagram | Content** from the context menu.

To create a diagram which shows the dependencies of an existing package:

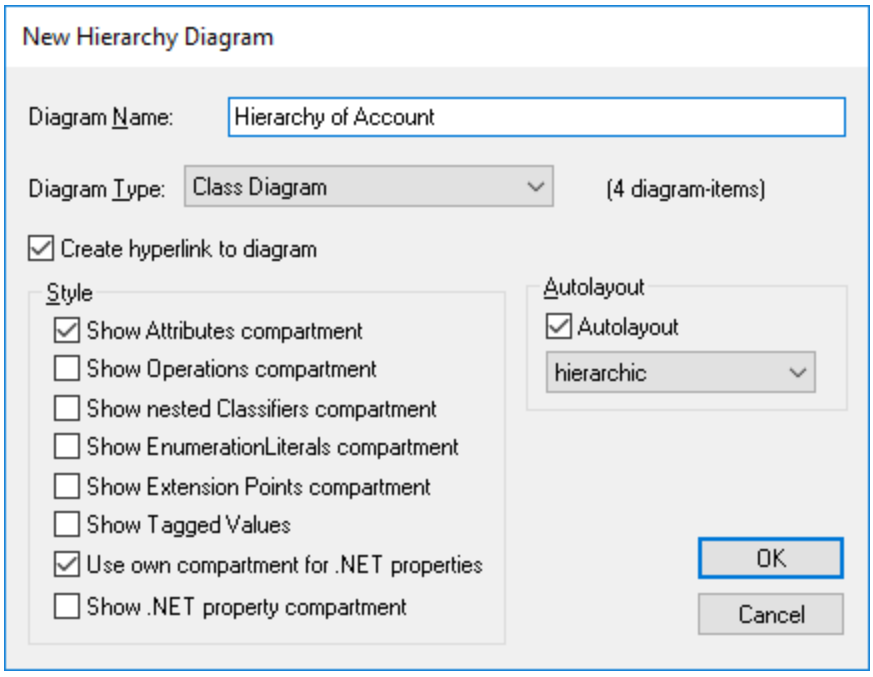
- Right-click a package in the Model Tree window and select **Show in new Diagram | Package dependencies** from the context menu.

To create a diagram which shows the generalization hierarchy of a class:

1. In the Model Tree window, right-click a class which has generalization relationships to or from other classes (for example, class `Account` from the sample project **C:\Users\<username>\Documents\Altova\UModel2025\UModelExamples\Bank_CSharp.ump**).

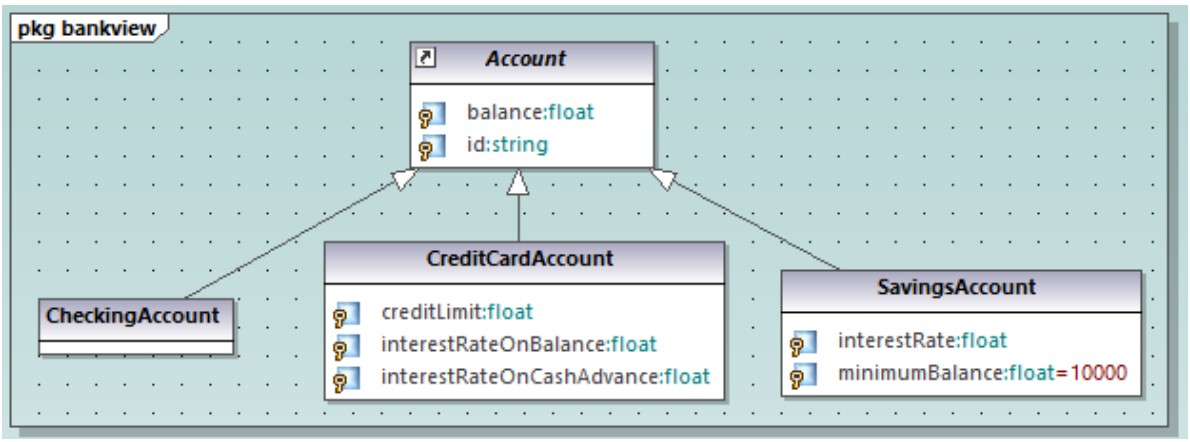


2. Select **Show in new diagram | Generalization hierarchy** from the context menu. A dialog box appears where you can adjust the preferences for the diagram to be created, including the diagram type. Notice the text "N diagram-items", which displays the number of items that are to be added to the diagram. In the example below, the chosen diagram type is "Class Diagram" and there will be four diagram items (classes) on the diagram: the `Account` class and three classes derived from it.



3. Click **OK**. The diagram is generated according to the selected options and opens in the Diagram window, for example:

© 2019-2025 Altova GmbH



5.2.3 Opening Diagrams

If the UModel project contains diagrams, these are displayed in the Diagram Tree window.

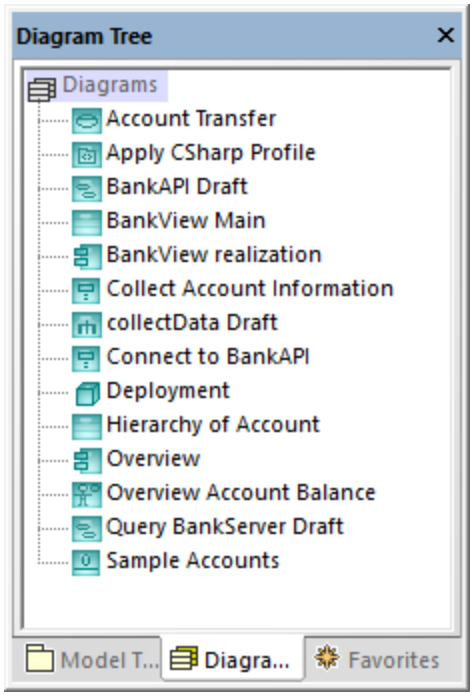
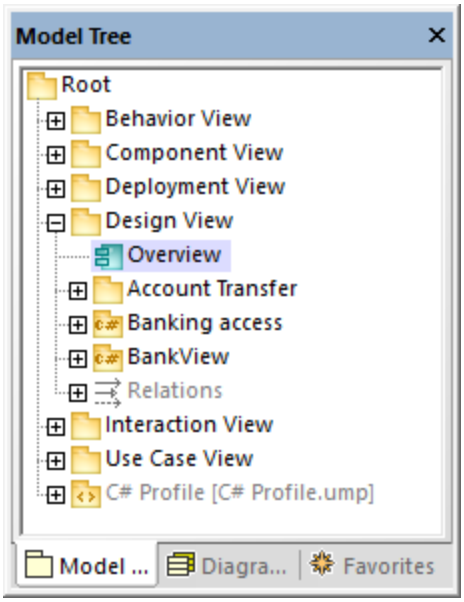


Diagram Tree window

Note: By default, diagrams are grouped by type in the Diagram Tree window. To display only diagrams (without parent groups), right-click inside the window and clear the **Group by diagram type** context menu option.

Diagrams are also displayed in the Model Tree window under the packages where they belong, for example:



To open an existing diagram:

- Double-click the diagram icon in the Model Tree window (or in the Diagram Tree window, or in the Favorites window).
- Right-click the diagram, and select **Open diagram** from the context menu.

5.2.4 Deleting Diagrams

UModel diagrams can be deleted in one of the following ways:

- In the Model Tree window (or Diagram Tree window, or Favorites window), right-click the diagram, and then select **Delete** from the context menu.
- Click the diagram in any of the windows mentioned above, and then press **Delete**.

Deleting a diagram does not remove any elements from the project except the diagram itself. To check if elements are used in any diagrams, right-click the package you would like to inspect, and select **List elements not used in any diagram**, see also Checking Where and If Elements Are Used 116.

For information about deleting elements from a diagram or from a project, see Deleting Elements 113.

5.2.5 Changing the Style of Diagrams

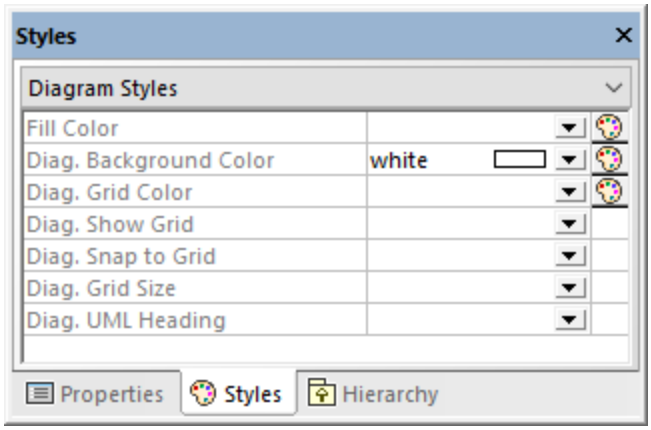
You can change the appearance (style) of a diagram, including the background color, grid visibility, grid size and color, as well as the appearance of the diagram heading. You can either change the style of individual diagrams in the project, or apply the same properties to all diagrams in the project. For information about changing the style of elements inside a diagram, see Changing the Style of Elements 122.

© 2019-2025 Altova GmbH

The size of diagrams is defined by elements and their placement. To enlarge the diagram size, drag an element to one of the diagram edges and the size will adjust accordingly.

To change the appearance of diagrams:

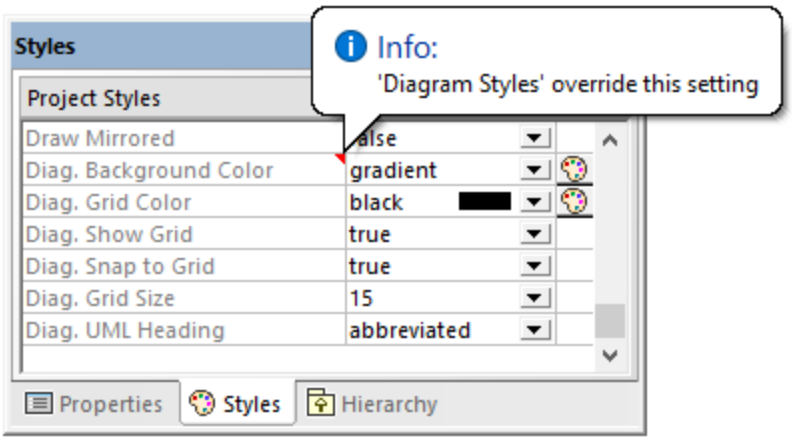
1. Open a diagram (see Opening Diagrams 127).
2. Notice the dropdown list at the top of the Style Window and do one of the following as applicable:
   a. To edit the properties of the current diagram only, select "Diagram Styles" from the list. This value is selected by default if you click anywhere where the diagram background is empty (that is, when you do not click any diagram elements).



   b. To apply changes to all diagrams in the project, select "Project Styles". In this case, scroll down to the end of the Styles window until you find the styles applicable to diagrams (that is, the ones that begin with "Diag.").
3. Change the value of the required property (for example, "Diagram Background Color").

> Styles applied at diagram level override those applied at project level.

When a style is overridden, a small red triangle appears in the upper-right corner of the overridden property. Move the cursor over the triangle to display a tooltip with information about style precedence.



Overridden diagram style

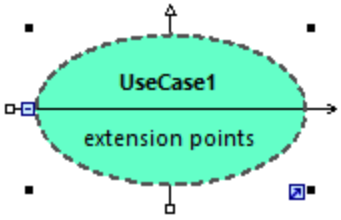
The following diagram-specific properties are available as toolbar buttons. Changing the property in the Styles window will update the state of the toolbar button, and vice versa.

	Show grid	Shows or hides the diagram grid.
	Show diagram heading	Shows or hides the diagram heading.
	Snap to grid	When enabled, this property makes all elements adhere to the grid. When disabled, elements are positioned regardless of the grid pattern.

5.2.6 Aligning and Resizing Modeling Elements

You can change the size of elements on the diagram as follows:

1. Click an element on the diagram. A set of black dots appear at the element's edges.

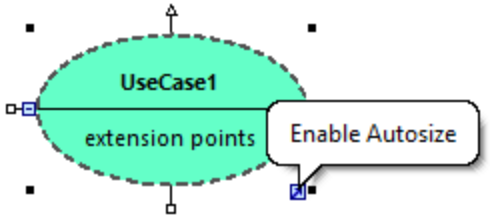


2. Drag any of the black dots into the direction where you want the element to grow.

To reset the element size to its default boundaries, do one of the following:

- Click the **Enable Autosize** icon at the lower-right corner of the element.

© 2019-2025 Altova GmbH



- Right-click an element on the diagram, and select **Autosize** from the context menu.
- Select one or more elements. On the **Layout** menu, click **Autosize**.

When at least two modeling elements are selected on the diagram, they can be aligned in relation to each other (for example, both can be aligned to have the same horizontal or vertical position, or even size). The commands which align or resize elements are available in the **Layout** menu and in the Layout toolbar.

Layout toolbar

When you select several elements, the element that was selected **last** serves as a template for the subsequent align or resize commands. For example, if you select three class elements and run the **Make same width** command, then all three will be made as wide as the last class you selected. The element that was selected last always appears with a dashed border.

The commands specific to element alignment and resizing are as follows:

Icon	Command	Notes
	Align left	
	Align right	
	Align top	
	Align bottom	
	Center vertically	
	Center horizontally	
	Space across	This command is available when three or more elements are selected. It distributes the horizontal space evenly between selected elements.
	Space down	This command is available when three or more elements are selected. It distributes the vertical space evenly between selected elements.
	Line up horizontally	This command repositions all selected elements on the diagram so that they are arranged horizontally one after the other.

Icon	Command	Notes
	Line up vertically	This command repositions all selected elements on the diagram so that they are arranged vertically one after the other.
	Make same width	
	Make same height	
	Make same size	

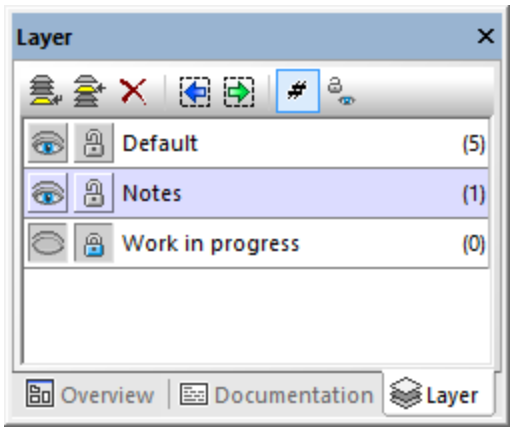
You can also automatically layout all elements in the diagram, as follows:

- On the **Layout** menu, click **Autolayout All** and choose one of the following options: **Force Directed**, **Hierarchic**, or **Block**.

Force Directed	Displays the modeling elements from a centric viewpoint.
Hierarchic	Displays elements according to their hierarchical relationships. For example, a superclass will be placed above any of its derived classes. The hierarchical layout options can be customized from the **Tools | Options** menu, **View** tab, **Autolayout Hierarchic** group.
Block	Displays elements grouped by element size in rectangular fashion.

5.2.7 Adding Layers to Diagrams

By default, a diagram consists of a single layer—this layer stores all the elements visible on the diagram canvas. However, you can optionally add multiple layers to a diagram. With layers, you can make logical groupings of modeling elements within the same diagram and thus separate concerns. For example, you can create, in addition to the default layer, some extra layers that would store notes with some internal information, or unfinished classes. Layers can be viewed and managed from the Layer window.



Layer window

© 2019-2025 Altova GmbH

In the image above, three layers are defined on the diagram. The layer "Notes" is currently selected. The third layer, "Work in progress", is currently locked. The number displayed in the brackets to the right of each layer denotes how many elements each layer has.

Any UML element can be assigned to any layer. By default, new elements are added to the currently active layer, which is highlighted in the Layer window. If all layers are visible, you can create relationships such as association, generalization, etc between elements on different layers.

When printing diagrams or saving them to an image, only elements from the currently visible layers are printed. The maximum number of layers per diagram is 20.

The buttons available in the Layer window have the following purpose:

Icon	Command	Notes
	Append layer	Appends a new layer to the current layer list, and assigns a default name which you can change immediately or through the context menu option "Rename".
	Insert layer	Inserts a new layer above the currently active layer in the layer list.
	Delete layer	Deletes the currently active layer. Before the layer is deleted, a dialog box opens asking where the current layer's items (if any) should be moved (merged).
	Focus previous on active layer	Selects the previous element on the currently active layer. This command is enabled only if the layer contains elements.
	Focus next on active layer	Selects the next element on the currently active layer. This command is enabled only if the layer contains elements.
	Layer item count	Shows or hides the count of elements in each layer.
	Reset all layer states	Sets all layers to visible and unlocked state.

Some of the commands above are also available as context menu items, when you right-click inside the Layer window.

To move elements from one layer to another:

- Right-click the element on the diagram and select the **Layer | <layer name>** command from the context menu. This command is also applicable after you selected multiple elements; in this case, all of them will be moved to the destination layer.
- Alternatively, select one or more elements on the diagram and drag them onto the destination layer in the Layer window.
- To move all elements of a layer into a different one, right-click the layer, and select **Merge To | <layer name>** from the context menu.

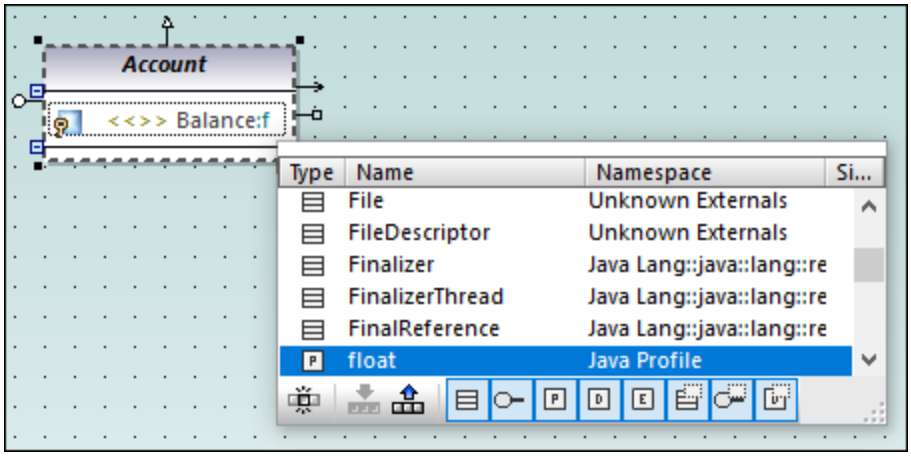
To show, hide, or lock individual layers, or multiple layers at once:

- Right-click the layer in the Layer window, and select the **Show**, **Hide**, or **Lock** command, respectively. The submenu commands **Selected layer** and **Others** let you toggle the command either for the currently selected layer, or for all layers except the one currently selected.
- Alternatively, right-click the layer, and use the **Toggle Visibility** or **Toggle Lock** commands, respectively. This will hide the layer(s) if they were previously shown, or lock them if they were previously unlocked (and vice versa).

5.2.8 Type Autocompletion in Classes

When you add operations and attributes to a class, autocompletion of data types is enabled by default in UModel. This makes it possible to specify the data type of the operation or property directly on the diagram, for example:

1. Right-click a class, and select **New | Operation** from the context menu.
2. Type the name of the operation after the double angle brackets << >>, and then type the colon (:) character.
3. An autocompletion window is automatically opened.



Autocompletion window

The autocompletion window has the following features:

- Clicking a column name sorts the window by that attribute in ascending or descending order.
- The window can resized by dragging the bottom-right corner.
- The window contents can be filtered by clicking the respective filters (categories) at the bottom of the window: Class, Interface, PrimitiveType, DataType, Enumeration, Class Template, Interface Template, DataType Template.

© 2019-2025 Altova GmbH

To enable only one of the filters at a time:

- Click the **Single mode** button . The image above shows the autocompletion window in "multi-mode", that is, all filters are enabled. The single mode button is not enabled.

To select or clear all filters simultaneously:

- Click the **Set All Categories** or **Clear All Categories** buttons, respectively.

To disable autocompletion:

1. On the **Tools** menu, click **Options**, and then click the **Diagram Editing** tab.
2. Clear the **Enable automatic entry helper** check box.

To trigger autocompletion on demand (when it is disabled):

1. Make sure that the cursor is inside an attribute or operation of a class, after the colon (:) character.
2. Press **Ctrl+Space**.

5.2.9 Zooming into/out of Diagrams

To zoom into or out of a diagram, do one of the following:

- Run the menu command **View | Zoom In (Ctrl+Shift+I)** or **View | Zoom out (Ctrl+Shift+O)**.
- Select a predefined percentage value from the Zoom toolbar.

- Hold down the **Ctrl** key while rotating the mouse wheel.

To fit the diagram area to the visible window:

- Run the menu command **View | Fit to window** (or click the **Fit to window** toolbar button).

5.3 Relationships

5.3.1 Creating Relationships

A relationship typically needs two elements, so your diagram must already contain the elements between which you want to add relationships. You can create relationships as follows:

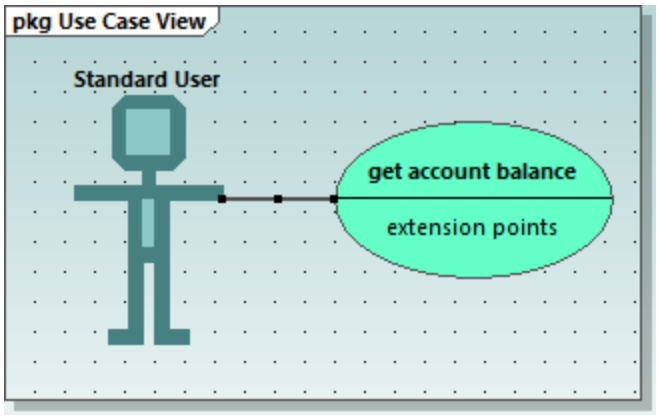
1. By using a toolbar button that depicts the relationship you need (for example, Association).
2. By using handles that appear when you click on any element on the diagram.

Creating relationships using toolbar buttons

When a diagram window is active in UModel's main pane (in focus), the toolbar displays all the elements and relationships supported by that diagram. For example, a Class diagram provides toolbar buttons for all supported relationships, including Association , Collection Association , Aggregation , Composition , Realization , Generalization , and others. Likewise, a Use Case diagram provides toolbar buttons for Associations , Generalizations , as well as Include and Extend relationships.

The instructions below illustrate how to create an association relationship between an actor and a use case. Use the same approach for other relationships you might need.

1. Click an element on the diagram (actor "Standard User", in the image below).
2. Click the toolbar button corresponding to the relationship you need (Association , in this example).
3. Move the mouse over "Standard User" and drag onto a target element ("get account balance" use case). Note that the target element is highlighted in green color and accepts the relationship only when it is meaningful according to UML specifications.



Association in a Use Case diagram

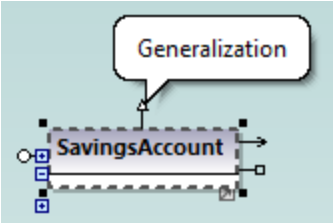
Creating relationships using handles

When you click an element on a diagram, several handles may appears to the left, right, top, or bottom of the element. The handles appear only for elements which support relationships. Each handle corresponds to a relationship kind. For example, class elements have the following handles:

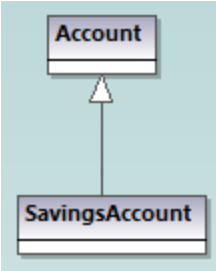
© 2019-2025 Altova GmbH

- InterfaceRealization
- Generalization
- Association
- Collection Association

To view the relationship kind that each handle creates, move the mouse over the handle. For example, in the image below, the selected top handle can be used to create a Generalization relationship.



To create the relationship, click the handle and drag the cursor over a destination element. This creates the corresponding relationship (Generalization, in this case).



Generalization relationship between two classes

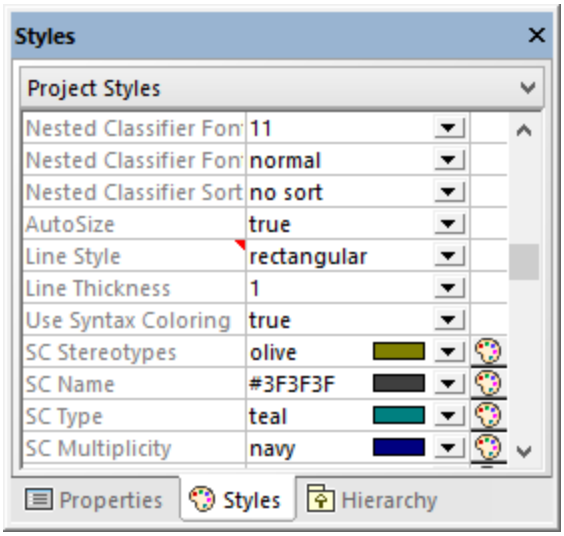
5.3.2 Changing the Style of Lines and Relationships

You can change the thickness, color, and bending style of lines from the Styles window. You can also add text (labels) to relationships, reposition labels, and hide/show labels on the diagram either individually for each relationship or in batch.

Note: In the instructions below, it is important to distinguish between "lines" (any line on the diagram) and "relationships" such as association, generalization, composition, and so on. All relationships are lines, but the opposite is not true. For example, a comment or note link is just a line, not a relationship.

To change line properties:

1. Click a line on the diagram.
2. In the Styles window, set the required property (for example, "Line Thickness").



The values available for the "Line Style" property are also available as commands under the **Layout | Line Style** menu, and as toolbar buttons. If you change this property, the corresponding toolbar button will become enabled, and vice versa.

	Orthogonal line	A line with this style will only bend at straight angles.
	Direct line	A line with this style will make a direct connection between two elements, without any waypoints.
	Custom line	A line with this style can bend at any angle. To move the line, drag any waypoint (small black dots) on the line. To create new waypoints, click in between two existing waypoints, and drag the line. To delete waypoints, drag a waypoint directly on the top of an existing one.

Line styles, just like other element styles, can be set for each individual line, or at a more generic level (project level, for example). The more specific style overrides the generic one. When a style is overridden, this is indicated by a red triangle next to the affected property in the Styles window, see also Changing the Style of Elements 122.

To add label text to a relationship:

- Click a relationship on the diagram, and start typing.

To move the label text:

- Click the label, and the drag it to some other position on the diagram.
- To move the label back to the default position, right-click the relationship, and select **Text Labels | Reposition Text Labels** from the context menu.

© 2019-2025 Altova GmbH

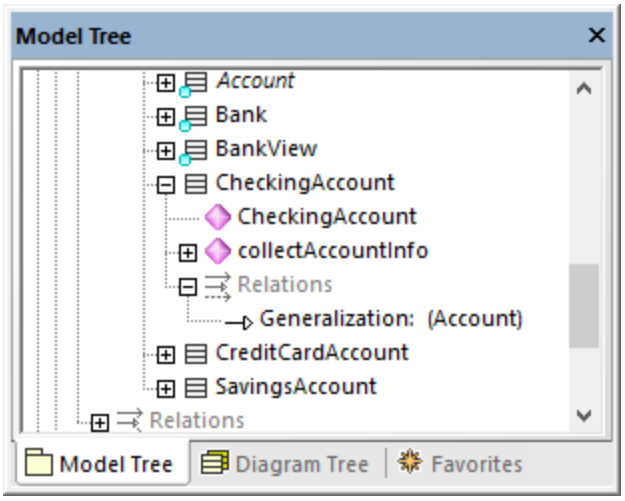
- To reposition multiple labels simultaneously, select one or more relationships on the diagram, and then run the menu command **Layout | Reposition Text Labels**.

To show or hide the label text:

- Right-click the relationship, and select **Text Labels | Show/Hide all Text Labels** from the context menu.

5.3.3 Viewing Element Relationships

By default, the relationships of an element are visible in the Model Tree window under that specific element. For example, the `CheckingAccount` class illustrated below has a Generalization relationship with the `Account` class:



Relationship in the Model Tree window

Note: To hide relationships from the Model Tree window, right-click inside the window and clear the **Show Relations in Tree** option.

To show the relationships of an element on the diagram, right-click the element on the diagram, and select **Show | <relationship kind>** from the context menu.

5.3.4 Associations

An association is a conceptual connection between two elements. You can create association relationships like any other relationship in UModel, see Creating Relationships 136.

When you create an association between two classes, a new attribute is automatically inserted in the originating class. For example, creating an association between `Car` and `Engine` classes adds a property of type Engine to the Car class.

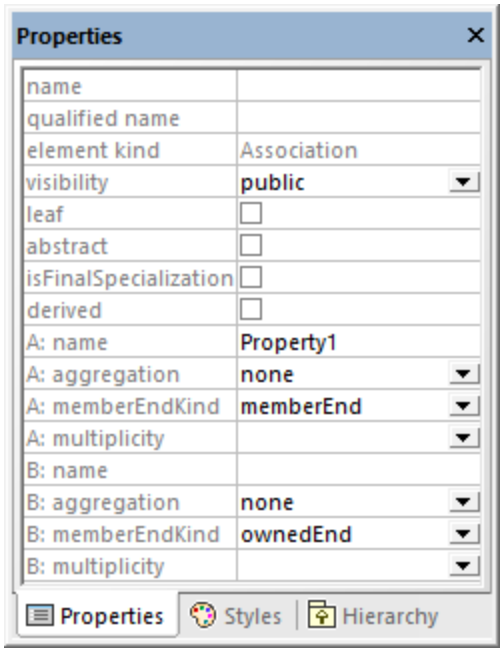
When a class is added to a diagram, its associations are shown automatically on the diagram, provided that the following conditions are met:

- The option **Automatically create Associations** is enabled from **Tools | Options | Diagram Editing** tab.
- The attribute's type is set (in the image above, `Property1` is of type `Engine`)
- The class of the referenced "type" is also present in the current diagram (in the image above, the class `Engine`).

You can also explicitly show the class properties of any class as associations on the diagram. To do this, right-click a class property, and select one of the following commands:

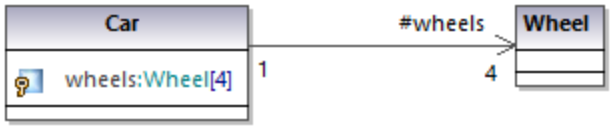
- **Show | <Property> as Association**
- **Show | All Properties as Associations**

When you click an association on the diagram, its properties can be changed, if necessary, from the Properties window.



It is important to note the properties listed below. Modifying these properties changes the appearance of the association on the diagram, or adds various informative text labels. For information about showing or hiding text labels, or changing the appearance of the relationship (such as color or line thickness), see Changing the Style of Lines and Relationships 137.

 © 2019-2025 Altova GmbH

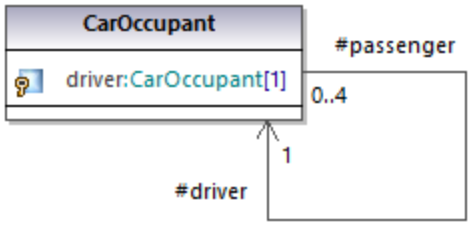
Property	Purpose
A: name	The name of the member on end A of the relationship. In the car example above, it is `Property1`.
A: aggregation	Enables you to change the type of association on end A. Changing this property will also change the representation of the association on the diagram. Valid values: **none** Denotes a normal association **shared** Changes the association into an aggregation **composite** Changes the association into a composition
A: memberEndKind	Attributes participating in a relationship can belong either to a class or to the association. This property specifies who owns this end of the relationship and whether this end of the relationship is navigable. ("Navigable" means that the end has an "arrow" ending). Valid values: **memberEnd** Member on this end belongs to the class. **ownedEnd** Member on this end belongs to the association **navigableOwnedEnd** Member on this end belongs to the association and this end becomes navigable. Setting both A and B ends to **ownedEnd** makes the association bi-directional.
A: multiplicity	Multiplicity specifies the number of objects at this end of the relationship. For example, if a car has four wheels, multiplicity would be 1 on one end and 4 on the other end of the relationship. 

The same set of attributes are available for end B of the relationship.

If enabled, the property **Show Assoc. Ownership** in the Styles window displays ownership dots for the selected relationship. By default, this property is set to **False**. The following is an example of a class where **Show Assoc. Ownership** is set to **True**:

Creating reflexive associations

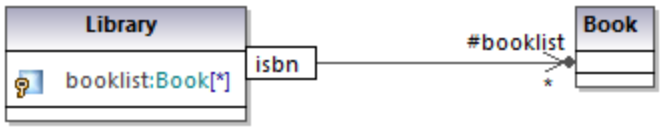
Associations can be created using the same class as both the source and target. This is a so-called "self link", or reflexive association. It may describe, for example, the ability of an object to send a message to itself, for recursive calls. To create a self link, click the association toolbar button, then drag from the element, dropping somewhere else on the same element.



Creating association qualifiers

Associations can be optionally decorated with association qualifiers. Qualifiers are attributes of an association. In the example below, the association qualifier `isbn` specifies that a book can be retrieved from the list of books by this attribute. To add a qualifier:

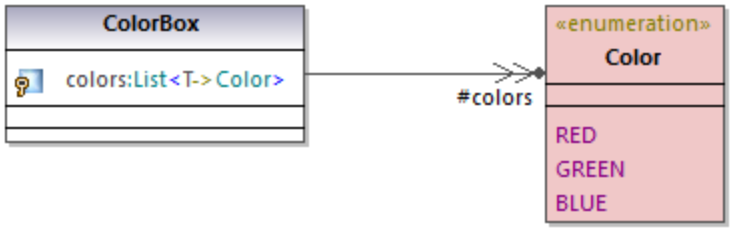
1. Create an association between two classes.
2. Right-click the association and select **New | Qualifier**.



To rename or delete association qualifiers, use the same steps as for all other elements, see Renaming, Moving, and Copying Elements 112 and Deleting Elements 113.

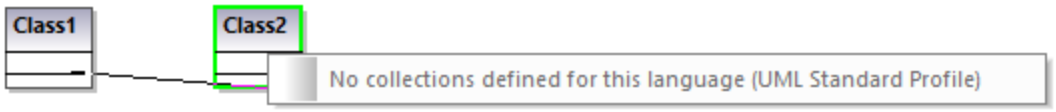
5.3.5 Collection Associations

A collection association relationship is suitable to illustrate that a class property is a collection of some kind. For example, in the diagram below, the property `colors` of the class `ColorBox` is a list of colors. This type is defined in this case as an enumeration; however, it may also be another class or even an interface.



© 2019-2025 Altova GmbH

Before you can create collection associations, the UModel project must contain the collection templates for the project language you want to use (such as Java, C#, or VB.NET). Otherwise, a tooltip with the text "No collections defined for this language" appears when you attempt to create the collection association.

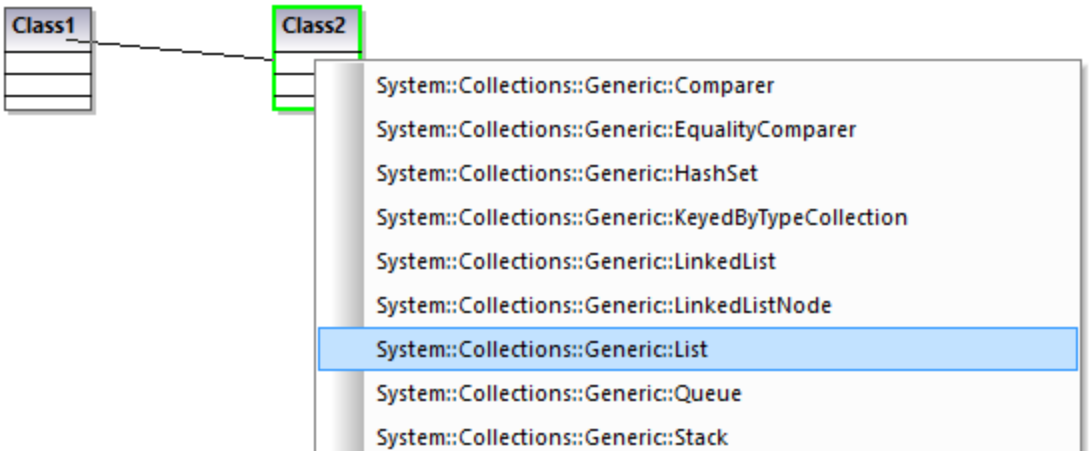


If your project is UML only (without support for a specific code engineering language), you can define collection templates from the menu **Tools | Options | Diagram Editing | Collection Templates | UML** tab.

If your project already contains a language namespace (such as Java, C#, VB.NET), the collection templates are predefined from the profile of that language. Additional templates can be added from the menu **Tools | Options | Diagram Editing | Collection Templates**.

To create a collection association (between two classes, for example):

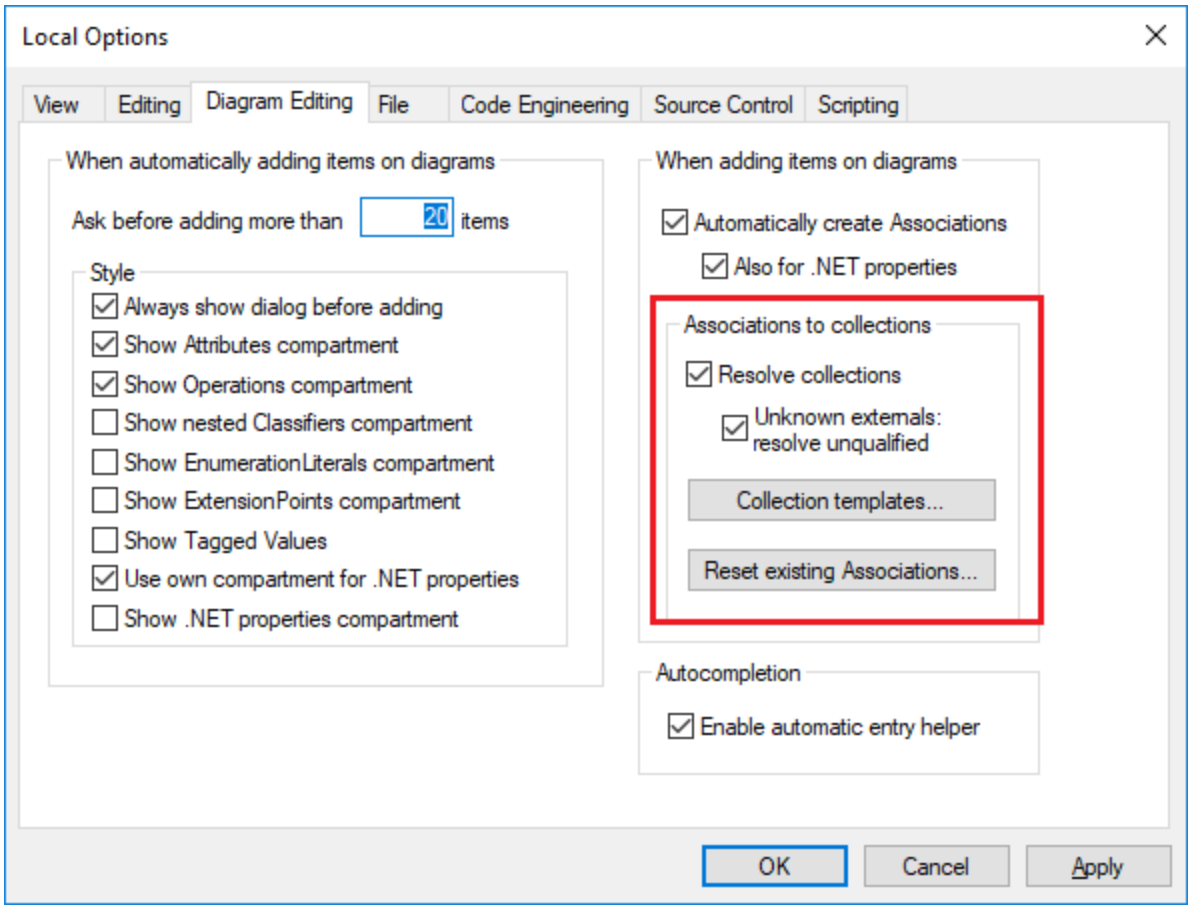
1. Add two classes to the diagram.
2. Click the **Collection Association** toolbar button.
3. Drag from the first class and drop it onto the second class. The collection templates defined for the project appear in the context menu, and you can select the required one.



Collection associations and code engineering

If you import program code into the model, collection associations are created automatically by default, based on predefined collection templates. To enable or disable this option:

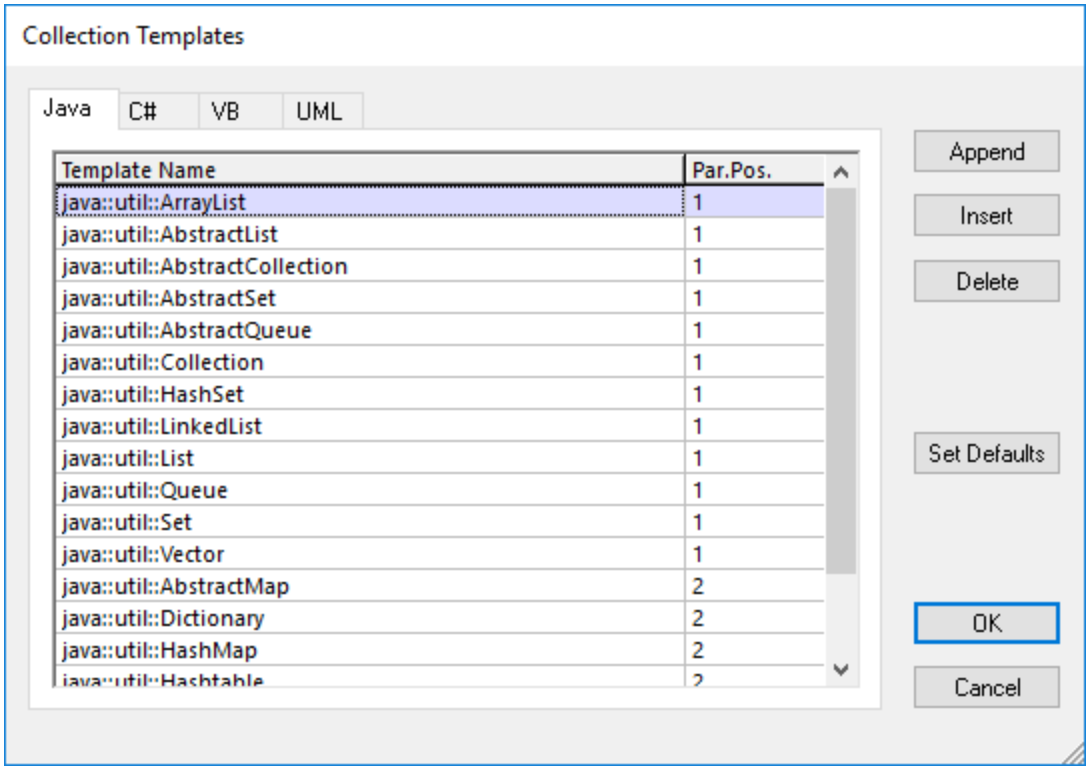
1. On the **Tools** menu, click **Options**.
2. Click the **Diagram Editing** tab.
3. Select or clear, as necessary, the check box **Resolve collections**.



The collection associations are resolved by default based on a list of built-in collection templates. To view or modify the built-in collection templates, click **Collection Templates**.

To insert custom collection types, use the **Append**, **Insert**, or **Delete** buttons available in the dialog box below. The column **Par.Pos.** denotes the position of the parameter which contains the value type of the collection.

© 2019-2025 Altova GmbH



Collection Templates dialog box

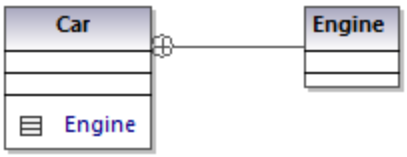
To reset the collection templates to their default values, click **Set default**.

5.3.6 Containment

A containment line is used to show, for example, parent-child relationships between two classes or two packages.

To illustrate containment between two classes:

1. Click the **Containment** toolbar button (in a class or package diagram).
2. Drag from the class that is to be "contained", and drop on the container class.



Note that the contained class, `Engine` in this case, is now visible in a compartment of `Car`. This also places the contained class in the same namespace as the container class.

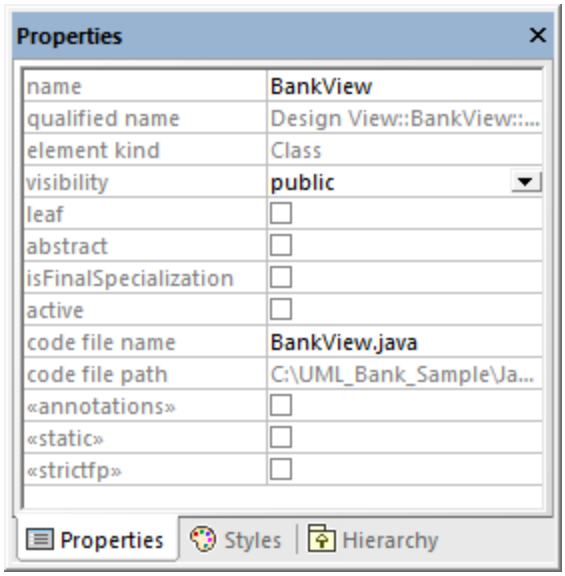
5.4 Stereotypes and Tagged Values

A stereotype is an extension mechanism; it is intended as a flexible way to extend an existing UML element and capture some aspect of it that standard UML doesn't. Stereotypes applied to an element signify that that element has some special use. The UModel built-in profiles (C#, Java, VB.NET, and so on) contain all the stereotypes required to model projects in the respective languages. However, you can also create your own profiles (and their respective stereotypes), see Creating and Applying Custom Profiles 412.

When you import source code or binaries into the model, UModel applies stereotypes to elements automatically, based on the structure of the original code. For example, if annotations modifiers exist in the imported Java source code, the corresponding elements in the model get the «annotations» stereotype. For information about how various language constructs map to UModel elements and become stereotypes in the model, see UModel Element Mappings 224.

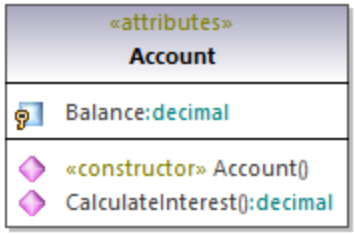
You can also apply stereotypes to elements manually, while modeling them. For example, you can apply the «attributes» stereotype to a C# class, which would indicate that the class must be decorated with attributes in generated code. To specify the attribute values in the generated code, you can add so-called "tagged values" in UModel, as shown in Applying Stereotypes 148. Stereotypes are also used extensively in XML schema modeling, to model elements such as simple types, complex types, facets, and so on. Likewise, stereotypes are used in database modeling, to model elements such as tables, columns, indices, and so on, see Designing Database Objects 495.

Across the UModel graphical interface, stereotypes are displayed enclosed within guillemets (for example, «static»). All stereotypes included into the built-in UModel profiles appear in the Properties window when you click an element. For example, clicking a Java class in the Model Tree would display in the Properties window only class stereotypes applicable to the Java profile (in this example, «annotations», «static», «strictfp»).



In class diagrams, stereotypes are visible above the name of the class. For example, the class below has the «attributes» stereotype applied to it.

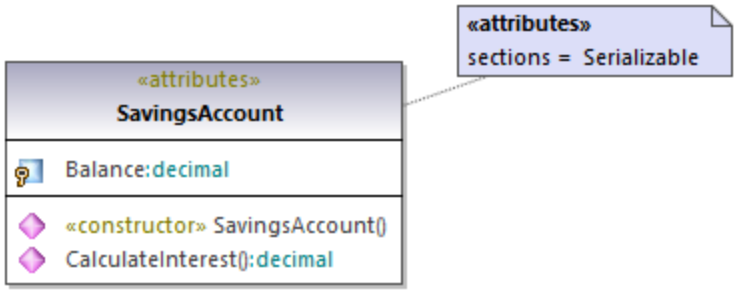
© 2019-2025 Altova GmbH



In case of methods or properties, stereotypes are displayed inline, like the «constructor» stereotype applied to the **Account()** method in the class above.

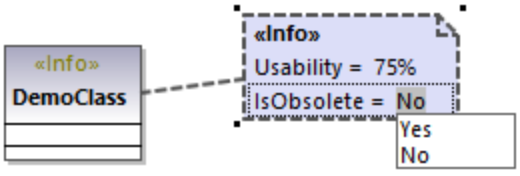
5.4.1 Tagged Values

Stereotypes may have attributes (tagged values) associated with them. Tagged values are name-value pairs that provide extra information related to the stereotype where they belong. For example, the class illustrated below has the stereotype «attributes» applied to it. Notice that the «attributes» stereotype has tagged values associated with it: a key (name) called "sections" and a value called "Serializable".



Tagged values

A stereotype may have multiple pairs of tagged values. Also, a value can be selected from a set of enumeration values.



You can change how tagged values are displayed on the diagram, or hide them altogether, see Showing or Hiding Tagged Values 150. For information about changing a stereotype's tagged values, see Applying Stereotypes 148. For an example that illustrates how to create stereotypes with tagged values, see Example: Creating and Applying Stereotypes 416.

5.4.2 Applying Stereotypes

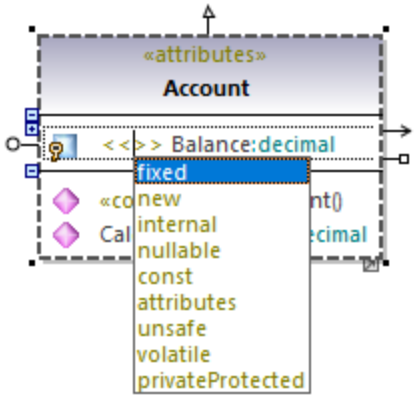
By applying a stereotype to an element, you indicate that the element has some specific use. In case of code languages supported in UModel (such as C#, VB.NET, Java), you typically apply stereotypes in order to comply with the grammar of that language. For example, a Java class may have the «static» stereotype applied to it.

> Before you can apply stereotypes, the corresponding profile must be applied to your package(s) first. This is done automatically by UModel if you right-click a package and select the **Code Engineering | Set as {language} namespace root** command. For more information, see Applying UModel Profiles 160.
>
> If you created custom profiles, these must be applied manually to the package, see Creating and Applying Custom Profiles 412.

To apply a stereotype to an element:

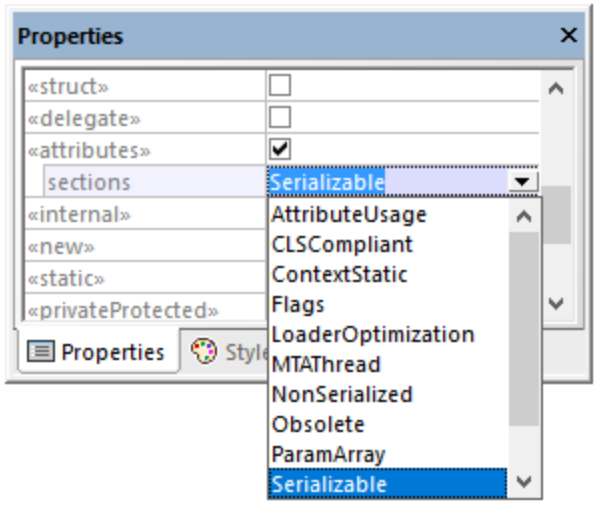
1. Click the element in the Model Tree window. If the element can be extended by any stereotypes, they appear as properties in the Properties window, enclosed within guillemets ("«" and "»").
2. Select the stereotype's check box in the Properties window (for example, «static»).

You can also apply stereotypes while designing elements inside a class diagram. To do this, click a property of a class and start typing text inside the "<< and ">>" characters.



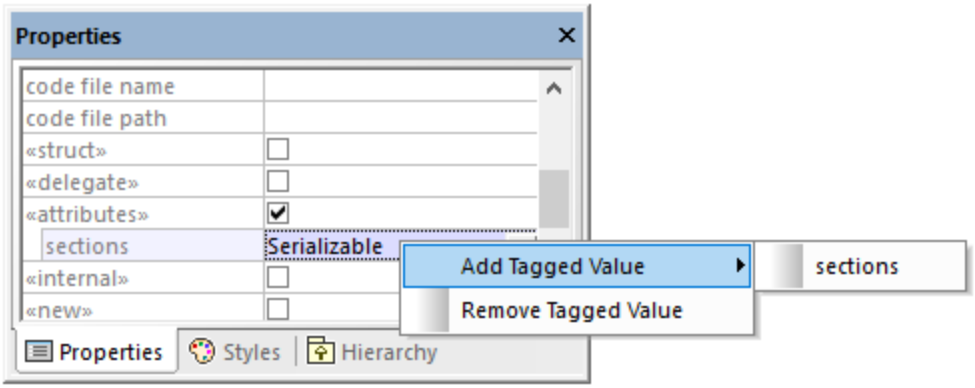
Some stereotypes are associated with a list of name-value pairs referred in UML as "tagged values". To apply a stereotype with tagged values to an element, select the stereotype's check box in the Properties window (in this example, «attributes»). This adds an indented entry where you can select the required value from a predefined list.

© 2019-2025 Altova GmbH

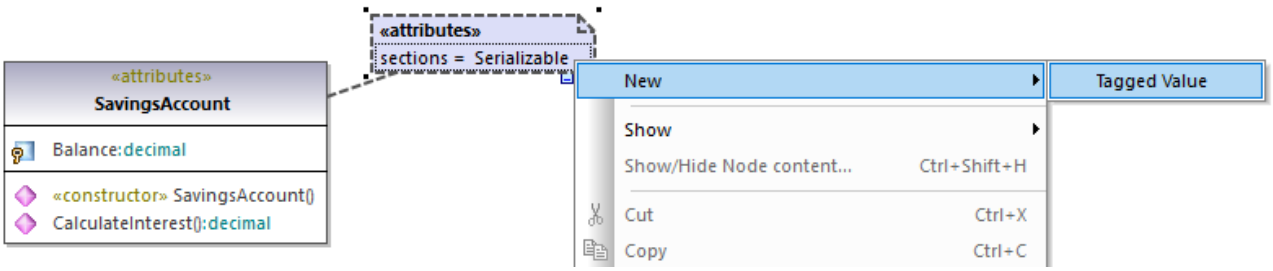


Tagged values

You can also add multiple values to the same key. To do this, right-click the idented entry, and select **Add Tagged Value | <name>** from the context menu.

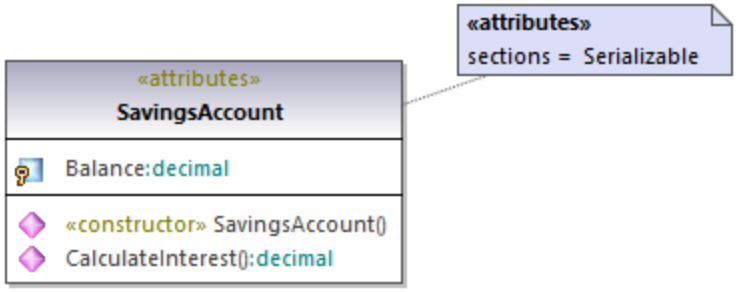


Alternatively, you can add tagged values directly from the diagram, by right-clicking a value, and selecting **New | Tagged Value** from the context menu.



5.4.3 Showing or Hiding Tagged Values

When an element has tagged values, you can view all the respective tagged values either in a standalone box, or inline, as a compartment. You can also hide tagged values completely. To choose how tagged values should be displayed, right-click the element on the diagram, and select **Tagged Values | <display option>**. For example, to display all tagged values outside of the class, right-click the class on the diagram, and select **Tagged Values | all**. To hide all tagged values of a class, right-click the class on the diagram, and select **Tagged Values | none**.

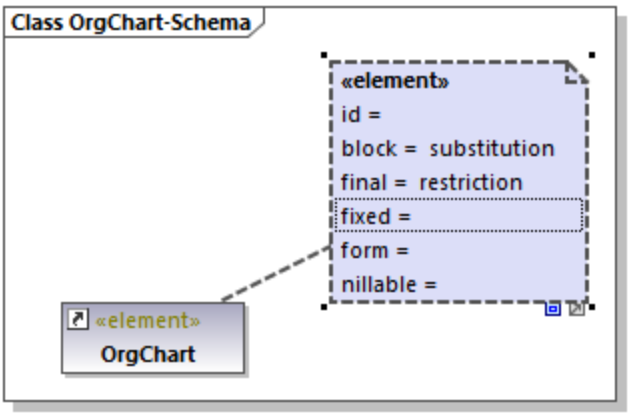


Tagged values displayed outside a class

Toggle compact mode

When some values in a tagged values box are empty, you can hide only the empty values, as follows:

1. Select a tagged values box on the diagram (one that has both empty and non-empty values).



2. Click the **Toggle compact mode** handle in the bottom-right corner of the box.

When the handle is in expanded state, the empty values are shown as well. When the handle is in collapsed state, the empty values are hidden.

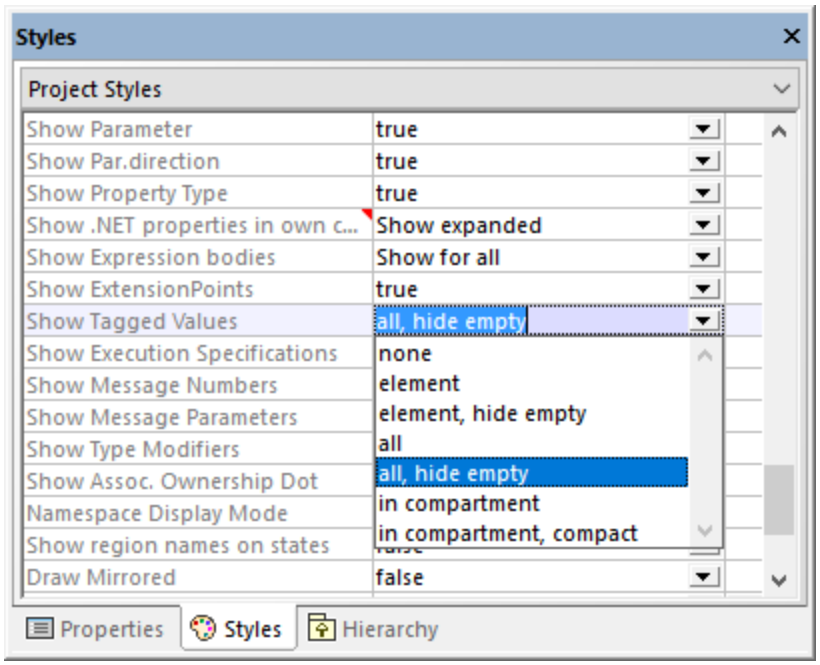
Changing the display of tagged values globally

You can change the display of tag values either individually for each element as shown above, or globally at project level.

© 2019-2025 Altova GmbH

To change tag values at project level:

1. Select **Project Styles** from the list at the top of the Styles Window[90].
2. Scroll down until to the **Show Tagged Values** property and select the required option from the list (for example, **all, hide empty**).

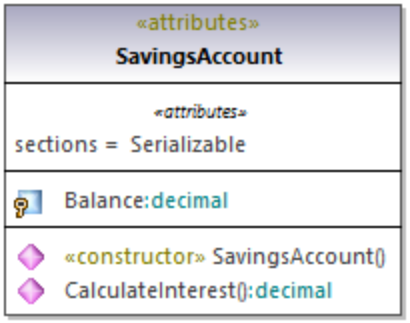


For information about changing styles at various levels, see Changing the Style of Elements[122].

Possible display options

The possible options for controlling the display of tagged values are listed in the table below. These options are similar when you change tagged values globally or for individual elements.

None	Hides all tagged values.
All	Displays the tagged values of an element (for example, a class) as well as those of elements owned by the class, such as attributes and operations.
All, hide empty	Displays only those tagged values where a value exists.
Element	Displays the tagged values of an element (for example, a class) but not those of owned attributes, operations, and so on.
Element, hide empty	Displays only those tagged values of an element where a value exists.

In compartment	Displays the tagged values in a separate compartment. For example, the class illustrated below has an *«attributes»* compartment that contains tagged values. 
In compartment, hide empty	Displays only those tagged values where a value exists, in a compartment.
In compartment, compact	Same as above.

© 2019-2025 Altova GmbH

6 Projects and Code Engineering

This chapter provides information about creating UModel modeling projects (either new, or by importing data from source code or binaries). It also describes various operations applicable to code engineering with UModel, namely:

- Forward engineering (generating code from a UModel project)
- Reverse engineering (importing source code into a UModel project)
- Roundtrip engineering (that is, synchronizing the model and code in either direction, as and when necessary)

The menu commands applicable to code engineering are available in the **Project** menu. For example, the menu command **Project | Import Source Project** enables you to import C#, or VB.NET Visual Studio solutions, or Java code, and generate UModel diagrams based on it. When no project solution is available, use the menu command **Project | Import Source Directory**, see Importing Source Code (Reverse Engineering) 192. Java, C#, and VB.NET binaries can also be imported, provided that a few basic prerequisites are met, see Importing Java, C# and VB.NET Binaries 204.

The code engineering operations above are applicable not only to programming languages but also to databases and XML Schema. For example, you could use the menu command **Project | Import XML Schema File** to reverse engineer an existing XML schema and automatically generate a class diagram based on it.

For the list of mappings between UModel elements and elements in each supported language profile (including databases and XML Schema), see UModel Element Mappings 224. For database connectivity instructions and operations applicable to databases, see UModel and Databases 486.

6.1 Managing UModel Projects

A UModel project acts as a container for UML modeling elements, diagrams, and various project-related settings that you may define. UModel projects are saved as files with .ump (UModel Project File) extension.

UModel does not force you to follow any predetermined modeling sequence. You can add any type of model element: UML diagram, package, actor etc., to the project in any sequence (and in any position). All model elements can be inserted, renamed, and deleted in the Model Tree window itself, you are not even forced to create them as part of a diagram.

6.1.1 Creating, Opening, and Saving Projects

When you start UModel for the first time, a new project is open automatically. On subsequent runs, UModel will open the most recent project you worked with.

Note: UModel includes several example projects that you can explore in order to learn the modeling basics and the graphical user interface. These can be found at the following path: **C:\Users\<username>\Documents\Altova\UModel2025\UModelExamples**.

To create a new project:

- On the **File** menu, click **New** (or click the **New** toolbar button).

A new project with the default name **NewProject1** is created. Also, the following packages are automatically added to the project and visible in the Model Tree window.

- Root
- Component View

These two packages have special use and are the only ones that cannot be renamed, or deleted, as explained in the tutorial, see Forward Engineering (from Model to Code) 64.

Once the project is created, you can add modeling elements to it, such as UML packages and diagrams, see Creating Elements 109 and Creating Diagrams 124.

To add a new package:

1. Right-click the package under which you want the new package to appear (either Root or Component View in a new project).
2. Select **New Element | Package** from the context menu.

Be aware that packages, as well as other modeling elements, can also be added from UML diagrams, in which case they will appear in the Model Tree window automatically.

To add a new diagram:

- Right-click a package in the Model Tree, and select **New Diagram**.

© 2019-2025 Altova GmbH

To add elements to a diagram:

- Do one of the following:
  - Right-click the diagram, and select **New Element | <Element Kind>** from the context menu.
  - Drag the desired element from the toolbar.

For a worked example of how to create a project and generate program code from it, see Forward Engineering (from Model to Code) 64.

To open an existing project:

- On the **File** menu, click **Open,** and browse for the .ump project file.

Note: By default, UModel registers any changes made externally to the .ump project file or included file(s), and displays a dialog box asking you to reload the project. This functionality can be disabled from the **Tools | Options | File** tab.

To save a project:

- On the File menu, click **Save** (or **Save as**).

All project relevant data is stored in the UModel project file, which has the extension *.**ump** (UModel Project File).

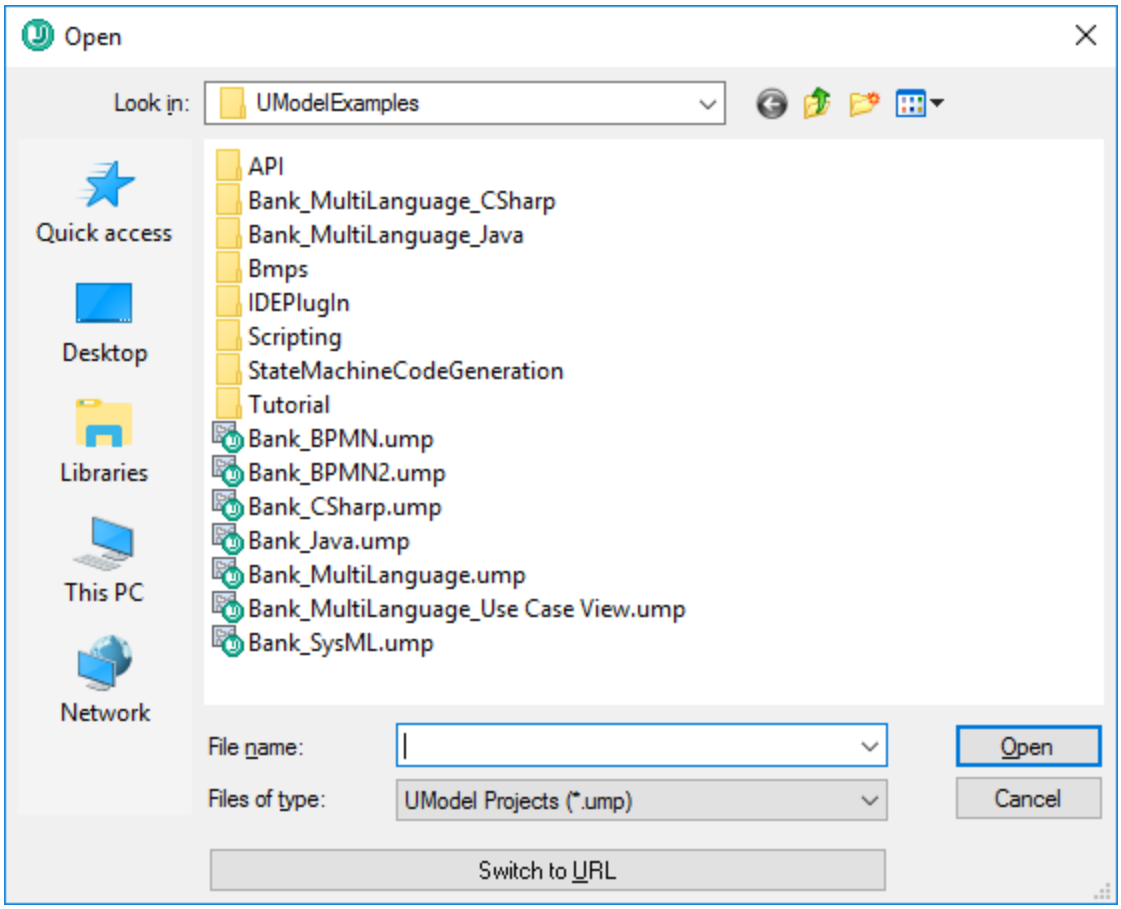
Note: The *.ump file is an XML file format which can be optionally "prettified" on saving. Pretty-printing can be enabled from the **Tools | Options | File** tab.

6.1.2 Opening Projects from a URL

In addition to opening local project files, you can also open files from a URL. The supported protocols are HTTP, HTTPS, and FTP. Note that files loaded from URLs cannot be saved back to their original location (in other words, access to the file is read-only), unless they are checked out from a Microsoft® SharePoint® Server, as shown below.

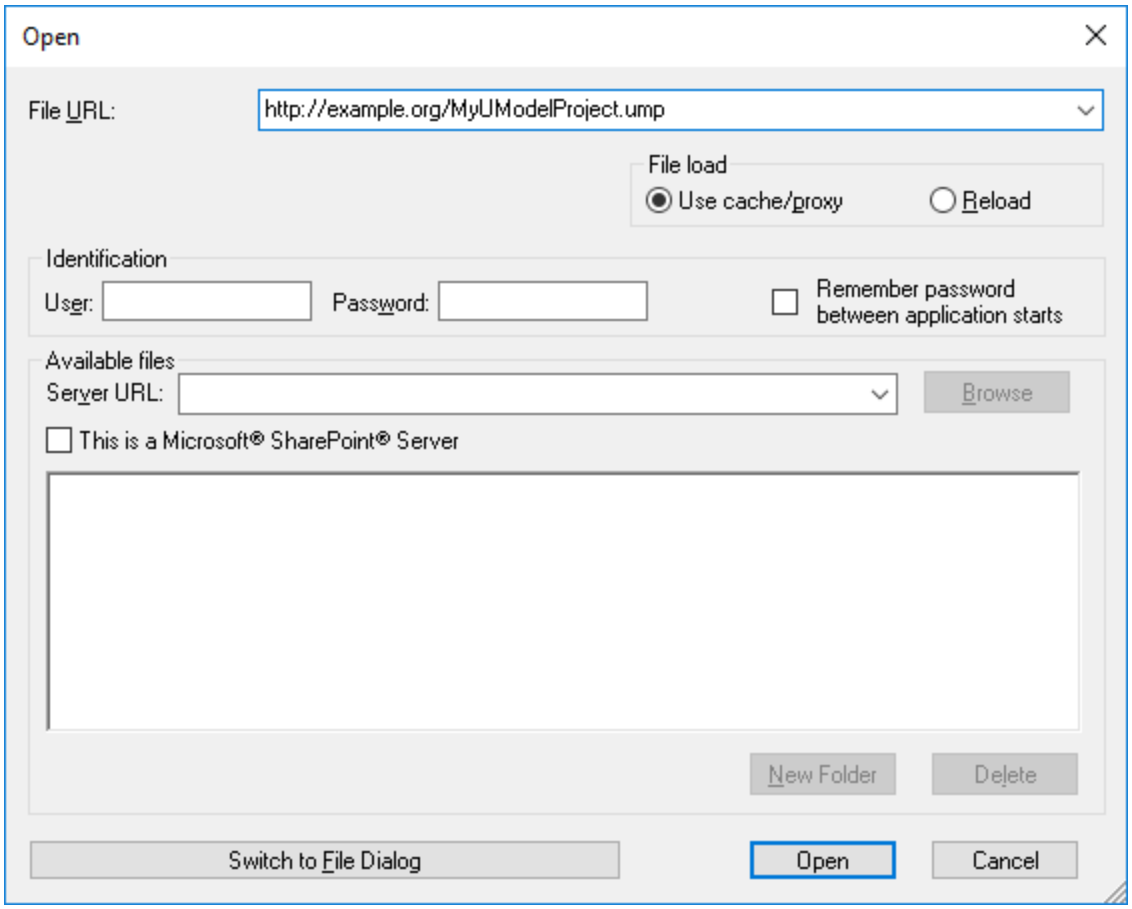
To open a file from a URL:

1. On the **Open** dialog box, click **Switch to URL**.



2. Enter the URL of the file in the **File URL** text box, and click **Open**.

© 2019-2025 Altova GmbH



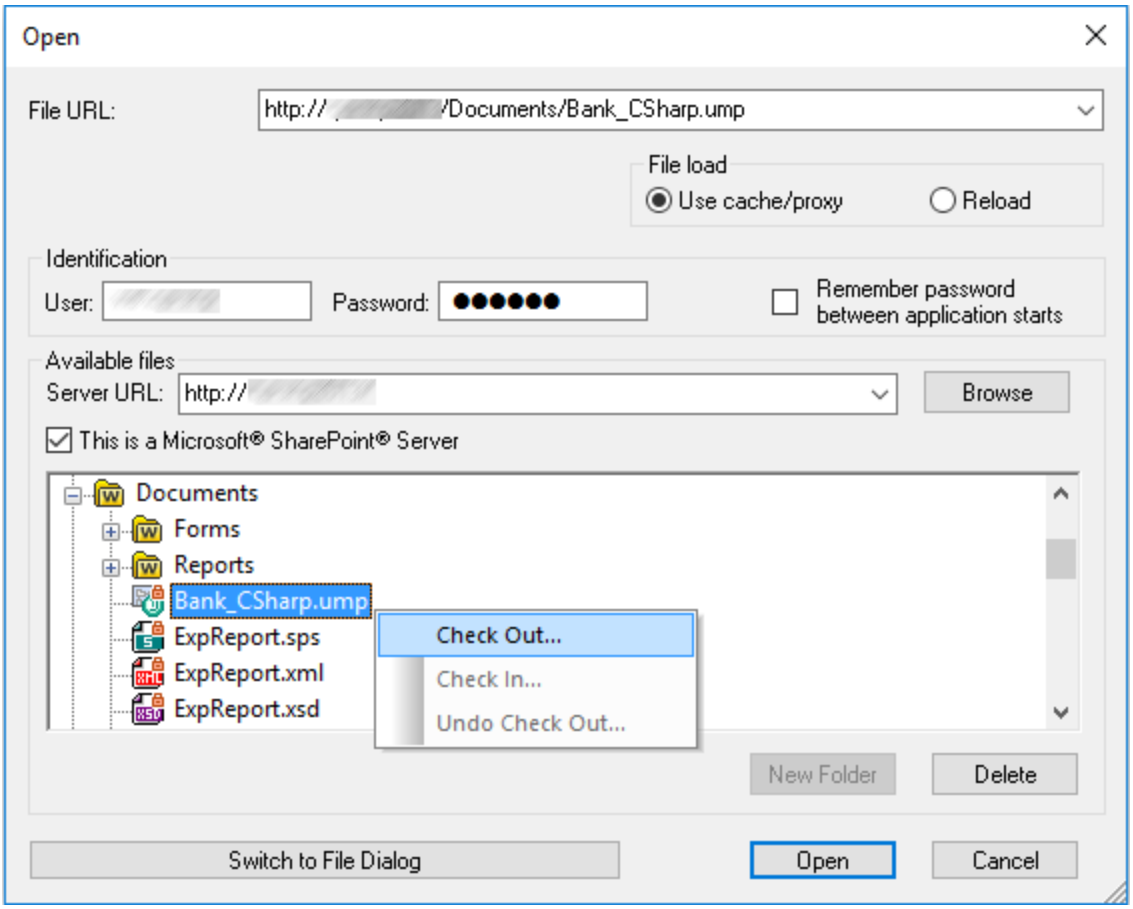
If the server requires password authentication, you will be prompted to enter the user name and password. If you want the user name and password to be remembered next time you start UModel, enter them in the Open dialog box and select the **Remember password between application starts** check box.

If the file you are loading is not likely to change, select the **Use cache/proxy** option to cache data and speed up loading the file. Otherwise, if you want the file to be reloaded each time when you open UModel, select **Reload**.

For servers with Web Distributed Authoring and Versioning (WebDAV) support, you can browse files after entering the server URL in the **Server URL** text box and clicking **Browse**.

Note: The **Browse** function is only available on servers which support WebDAV and on Microsoft SharePoint Servers.

If the server is a Microsoft® SharePoint® Server, select the **This is a Microsoft® SharePoint® Server** check box. Doing so displays the check-in or check-out state of the file in the preview area.



The state of the file can be one of the following:

	Checked in. Available for check-out.
	Checked out by another user. Not available for check-out.
	Checked out locally. Can be edited and checked-in.

To be able to modify the file in UModel, right-click the file and select **Check Out**. When a file is checked out from Microsoft® SharePoint®, saving the file in UModel sends the changes back to the server. To check in the file back to the server, right-click the file in the dialog box above, and select **Check In** from the context menu (alternatively, log on to the server and perform this operation directly from the browser). To discard the changes made to the file since it was checked out, right-click the file, and select **Undo Check Out** (or perform this operation from the browser).

Note the following:

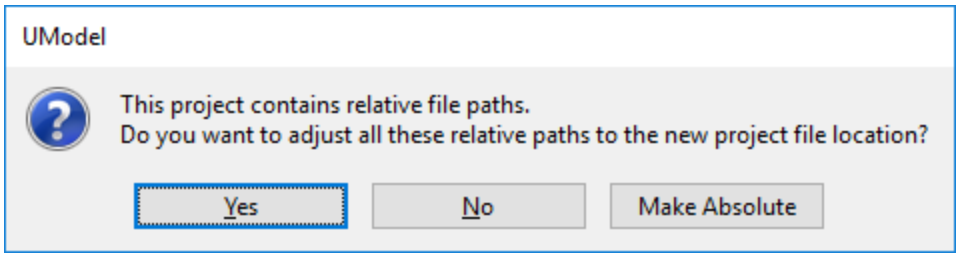
- When a file is already checked out by another user, it is not available for check out.
- If you check out a file in one Altova application, you cannot check it out in another Altova application. The file is considered to be already checked out to you.

© 2019-2025 Altova GmbH

6.1.3 Moving Projects to a New Directory

UModel projects and generated code can be easily moved to a different directory (or a different computer) and be resynchronized there. There are two ways to do this:

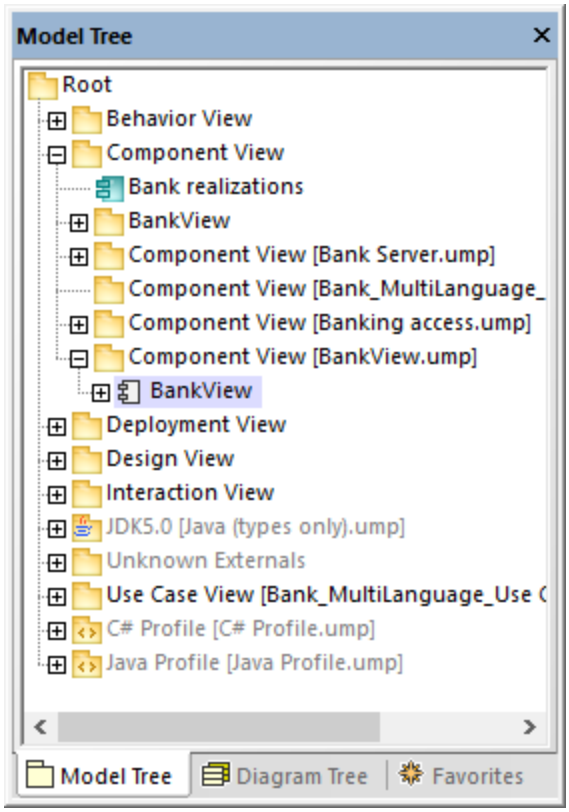
- Select the menu option **File | Save As...**, and click **Yes** when prompted to adjust the file paths to the new project location.



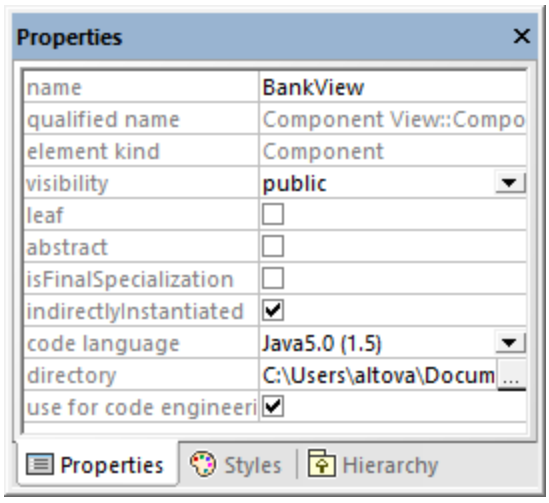
- Copy the UModel project (*.ump) to a new location, and then adjust the code generation paths for each component involved in code generation.

For an example of the second approach, open the following sample project: **C:\Users\<username>\Documents\Altova\UModel2025\UModelExamplesBank_Multilanguage.ump**.

1. Locate the `BankView` component in the Model Tree.



2. In the Properties window, locate the **directory** property and update it to the new path.



3. Re-synchronize the model and code.

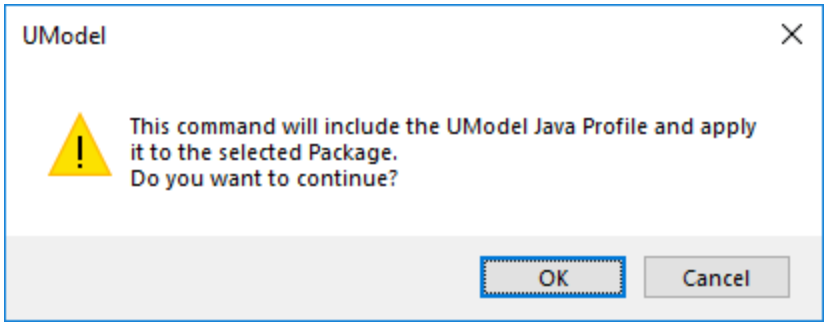
6.1.4 Applying UModel Profiles

By default, whenever you start a new modeling project in UModel, the project is unaware of the business application or code engineering language that you are going to need. Therefore, to tailor your UML project to a domain or language, you must *apply a profile* to it.

One must distinguish between two types of profiles:

- Profiles built into UModel (these include C#, VB.NET, Java, BPMN 1.0, BPMN 2.0, SysML, and so on).
- Custom profiles that you can create to extend UML to your specific domain or needs.

You can add any of the built-in profiles to your project by selecting the menu command **Project | Include Subproject**. In addition, UModel prompts you to apply a built-in profile whenever you take an action that requires that specific profile. For example, when you right-click some new package and select the **Code engineering | Set as Java Namespace Root** context menu option, you are prompted to apply the Java profile to it.



To view the full list of UModel built-in profiles or add them to your model manually, select the menu command **Project | Include Subproject**. See also Including Subprojects 164.

 © 2019-2025 Altova GmbH

For instructions about creating custom profiles in order to extend or adapt UML, see Creating and Applying Custom Profiles 412.

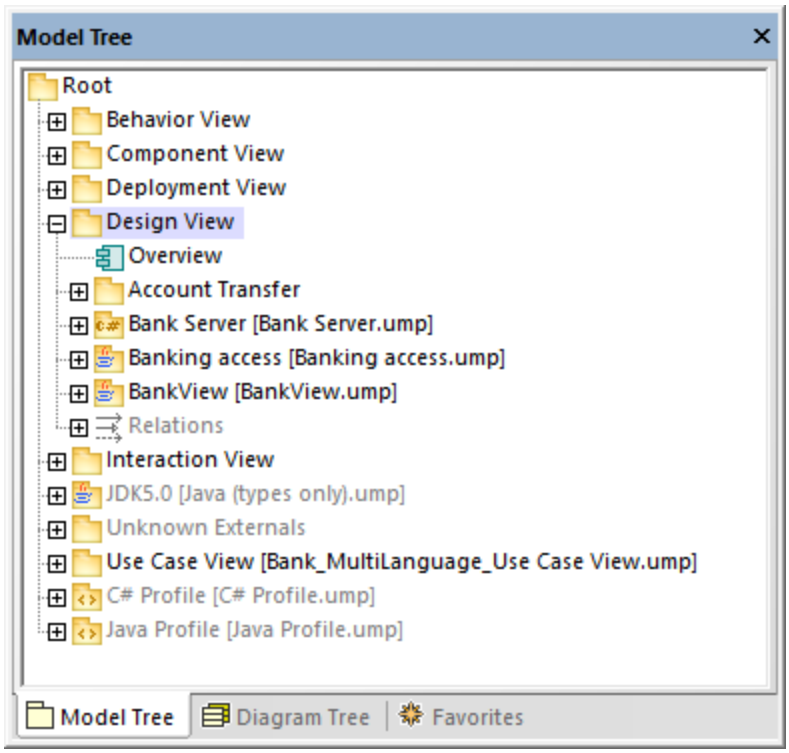
6.1.5 Splitting UModel Projects

You can split UModel projects into multiple subprojects and thus allow several developers to simultaneously edit different parts of a single project. Subprojects are like standard UModel project files and have the same *.ump extension. Each individual subproject can be added to a source control system. The top-level project is called the main project.

You can create a subproject from nearly any package in the main project. You can choose whether the subproject should be editable from within the main project, or be read-only. In the latter case, the subproject is editable only if you open it as a standalone project.

Subprojects can be structured in any way that you wish, in a flat or hierarchical structure, or a combination of both. This makes it theoretically possible to split off every package of a main project into subproject files.

In the Model Tree Window 83, subprojects appear with the respective .ump file name displayed to the right, enclosed within square brackets. For example, the project illustrated below includes several subprojects (this is the **Bank_MultiLanguage.ump** from the **C:\Users\<username>\Documents\Altova\UModel2025\UModelExamples** directory).

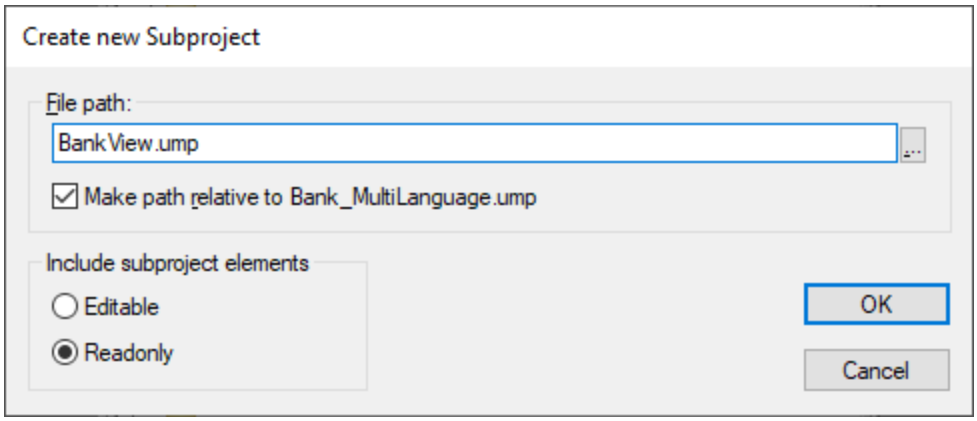


During the code-engineering process, all subordinate components of a subproject are considered. There is no difference between a single project file or one that consists of multiple editable subprojects. This also applies to UML diagrams—they can also be edited at the main, or subproject, level.

Note: You can also share packages and UML diagrams they might contain, between different projects. For more information, see Sharing Packages and Diagrams 166.

Creating subprojects

To create a subproject, right-click a package, and select the command **Subproject | Create new Subproject** from the context menu.



Next, click **Browse** and select the directory where the subproject should be saved.

Select **Editable** to be able to edit the subproject from the main project. (Selecting Read-only makes it uneditable in the main project.)

Note: You can change the file path of the subproject at any time by right clicking the subproject and selecting **Subproject | Edit File Path**.

Opening and editing subprojects

You can open a subproject as a standalone UModel project, directly from the main project. For this to be successful, there should not be any unresolved references to other elements. UModel automatically performs checks when creating a subproject from the "main" project, and whenever a file is saved.

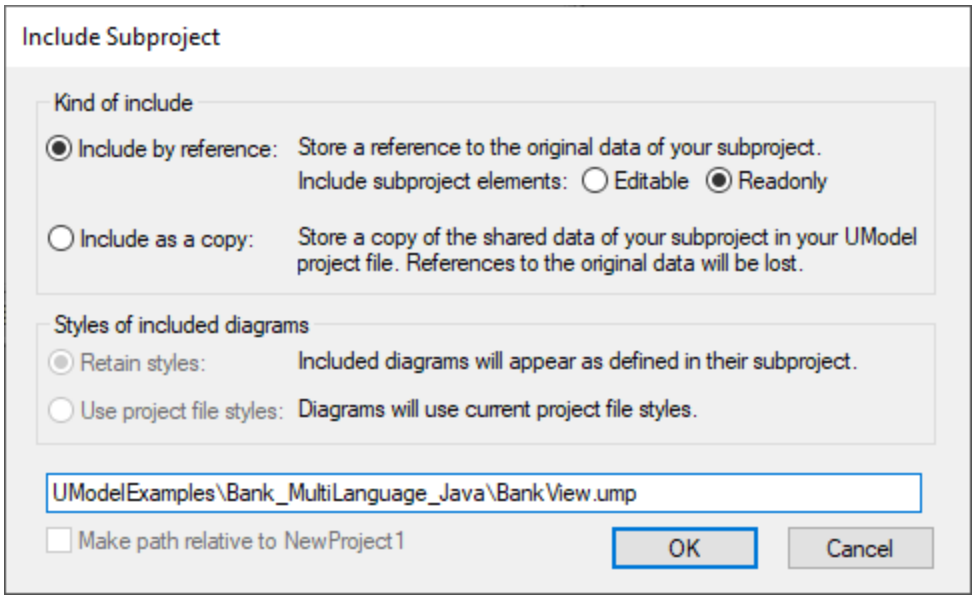
To open a subproject as a standalone UModel project, right-click the subproject package in the main project and select **Subproject | Open as Project**. This starts another instance of UModel and opens the subproject as a "main" project. Any unresolved references are shown in the Messages window.

Reusing subprojects

Subprojects that have been split off from a main project can be used in any other main project(s).

1. Open a project and select the menu command **Project | Include Subproject**.
2. Click the Browse button and select the *.ump file that you want to include.

© 2019-2025 Altova GmbH



3. Choose how the file is to be included; by reference or as copy.

Saving projects

When you save the main project file, all editable subproject files are also saved. You should therefore not create/add data (components) outside of the shared/subproject structure, if the subproject is defined as "editable" in a main project file. If data exists outside of the subproject structure, a warning message will be displayed in the Messages window.

Saving subproject files

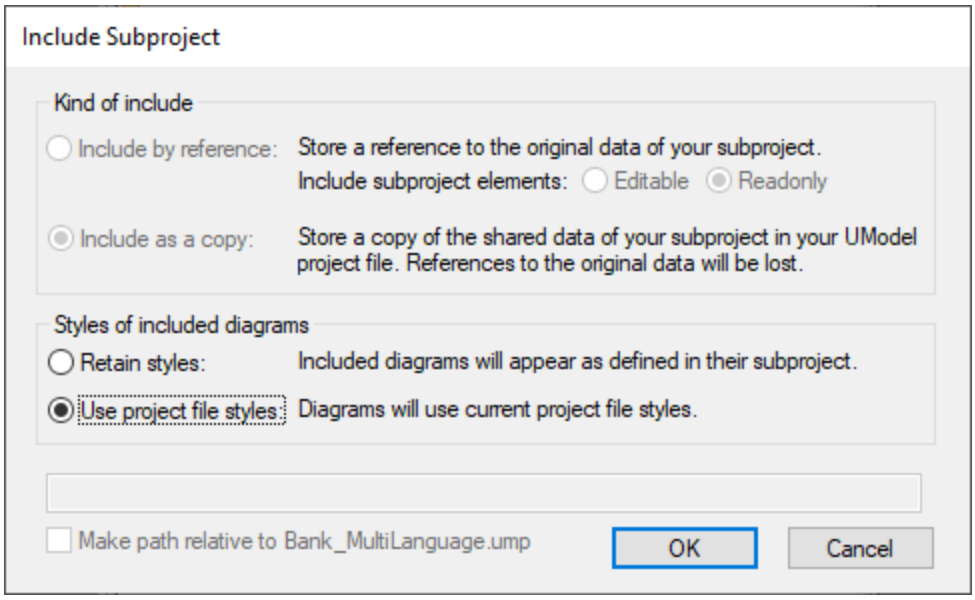
When saving subprojects (from the main project level), all references to sibling, as well as child subprojects, are considered and saved. For example, if two sibling subprojects, "sub1" and "sub2", exist and "sub1" uses elements from "sub2", then "sub1" is saved in such a way that it automatically saves references to "sub2" as well.

If "sub1" was opened as a "main" project, then it is considered as a self contained project and can be edited without any reference to the actual main project.

Reintegrating subprojects into the main project

You can copy previously defined subprojects back into the main project again. If the subproject does not contain any diagrams then the reintegration will be immediate. If diagrams exist, a dialog box will open.

1. Right-click the subproject and select **Subproject | Include as Copy**. This opens the "Include Subproject" dialog box, which allows you to define the diagrams styles you want to use when including the subproject.



2. Select the style option that you want to use, and then click **OK**.

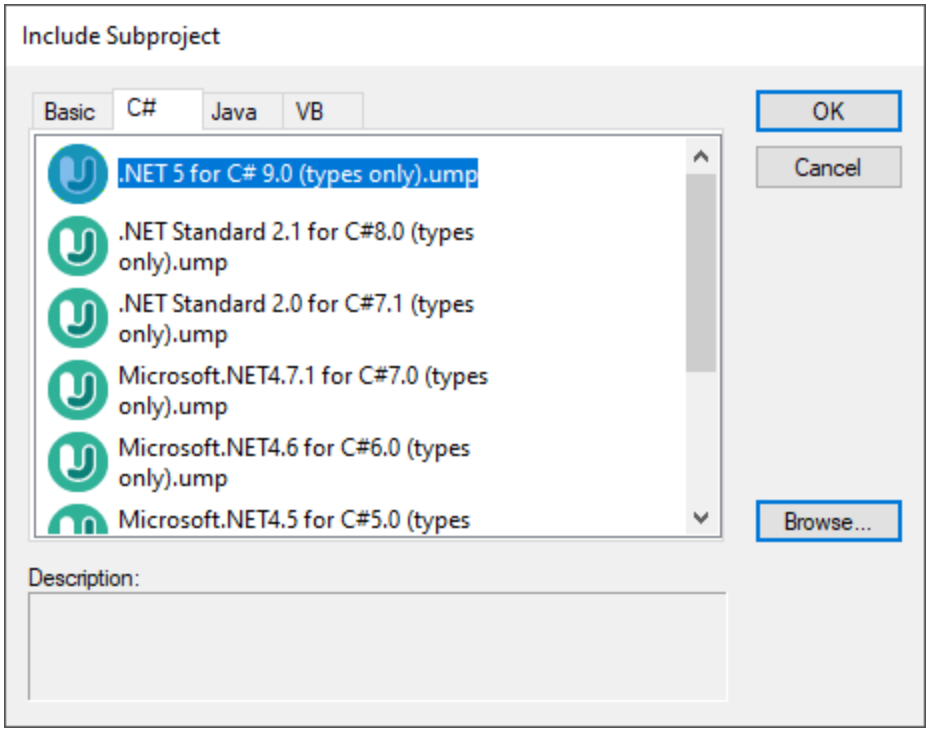
6.1.6 Including Subprojects

When you want to generate code from a model, or import source code into a model, a profile project applicable to that specific language (for example, C#, Java, VB.NET) must be included in your UModel project.

To include a UModel project as a subproject of another UModel project, select the menu command **Project | Include Subproject**. As illustrated below, several .ump subprojects (language profiles required for code engineering) are available on the **Basic** tab. In addition, several .ump subprojects containing C#, Java, and VB.NET types, organized by version, are available in tabs with the same name.

In order for all types to be recognized correctly during code engineering, make sure to include both the language profile (for example, the **C# profile**) and the types project of the corresponding language version (for example, **.NET 5 for C# 9.0**). Otherwise, an "Unknown Externals" package will be created in the project which will include all unrecognized types.

© 2019-2025 Altova GmbH

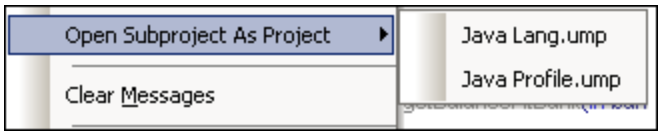


Include Subproject dialog box

The tabs and UModel projects (.ump files) available on the "Include Subproject" dialog box are configurable. Namely, UModel reads this information from the following path relative to the "Program Files" folder on your operating system: **\Altova\UModel2025\UModelInclude**. Note that the project files available on the **Basic** tab exist directly under the **UModelInclude** folder, while projects in each of the Java, VB, and C# tabs exist as subfolders of the **UModelInclude** folder.

To view all currently imported projects:

- Select the menu option **Project | Open Subproject Individually**. The context menu displays the currently included subprojects.



To create a custom tab on the "Include Subproject" dialog box:

- Navigate to the **\Altova\UModel2025\UModelInclude** folder (relative to your "Program Files"), and create your custom folder in it, for example **\UModelInclude\myfolder**. The name you give to the folder determines the name of the tab on the "Include Subproject" dialog box.
- Copy to your custom folder any .ump files that you want to make available on the corresponding tab.

To create descriptive text for each UModel project file:

- Create a text file using the same name as the *.ump file and place in the same folder. For example, the **MyModel.ump** file requires a descriptive file called **MyModel.txt.** Please make sure that the encoding of this text file is UTF-8.

To remove an included project:

1. Click the included package in the Model Tree view and press the **Delete** key.
2. When prompted, click OK to delete the included file from the project.

To delete or remove a project from the "Include Subproject" dialog box:

- Delete or remove the (MyModel).ump file from the respective folder.

6.1.7 Sharing Packages and Diagrams

You can share packages (and UML diagrams they might contain) between different UModel projects. Packages can be included in other UModel projects by reference, or as a copy.

Also note that subproject files can be split off a main, or subproject, file at any time. The subproject files can be included as editable or read-only from the main project; each package is shared and saved as a subproject file. Subprojects can be added to a source control system, see Teamwork support for UModel projects 161.

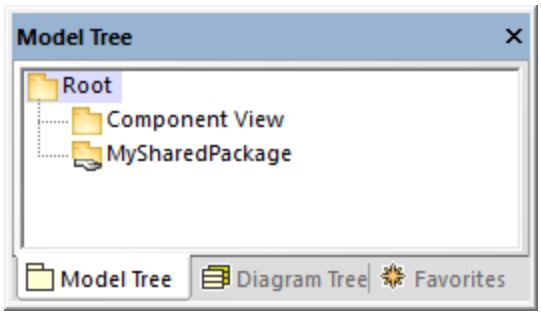
Notes

- In order to be shareable, a package must not contain links to external elements (elements outside of the shared scope).
- When creating UModel project files, do not use one project file as a "template/copy" for another project file into which you intend to share a package. This will cause conflicts due to the fact that every element should be globally unique (see uuid 569) and this will not be the case, as two projects will have elements that have identical uuids.

To share a package between projects:

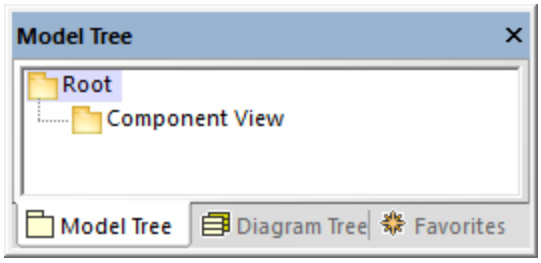
- Right-click a package in the Model Tree window and select **Subproject | Share package**. A "shared" icon appears below the shared package in the Model Tree. This package can now be included in any other UModel project.

© 2019-2025 Altova GmbH

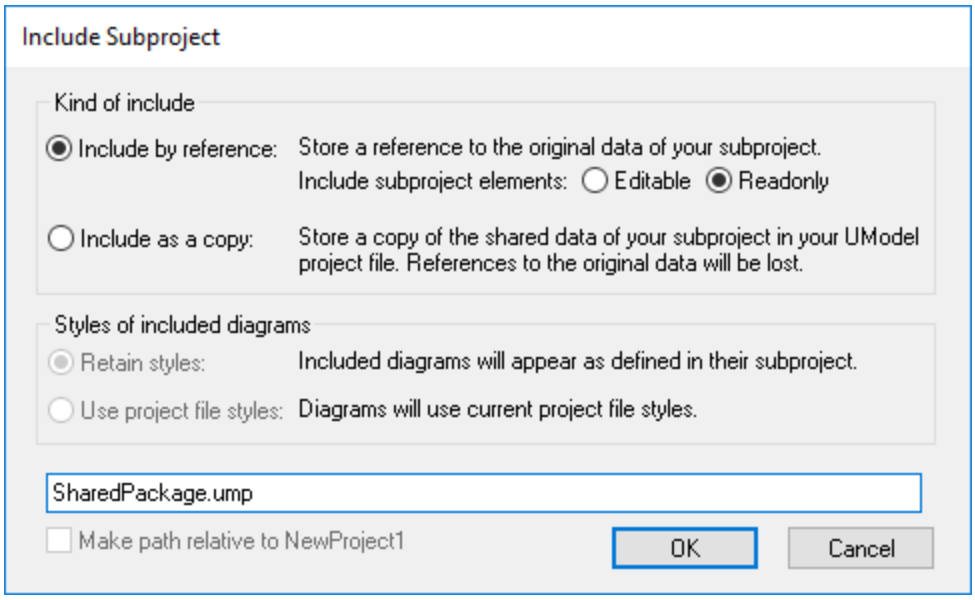


To include/import a shared folder in a project:

1. Open the project which should contain the shared package (an empty project in this example).

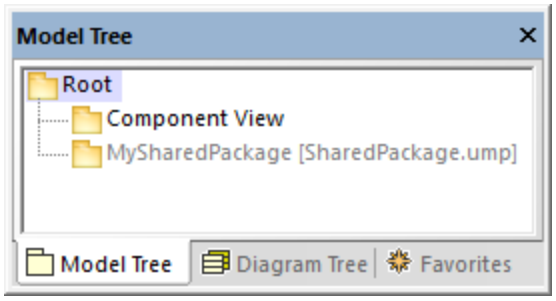


2. Select the menu item **Project | Include Subproject...**
3. Click **Browse**, select the project that contains the shared package, and click **Open**. The "Include Subproject" dialog box allows you to choose between including the package/project by reference, or as a copy.



4. Select the required option ("Include by reference", in this example) and click OK.

The "Deployment View" package is now visible in the new package. The packages' source project is displayed in parenthesis (**SharedPackage.ump**, in this example).

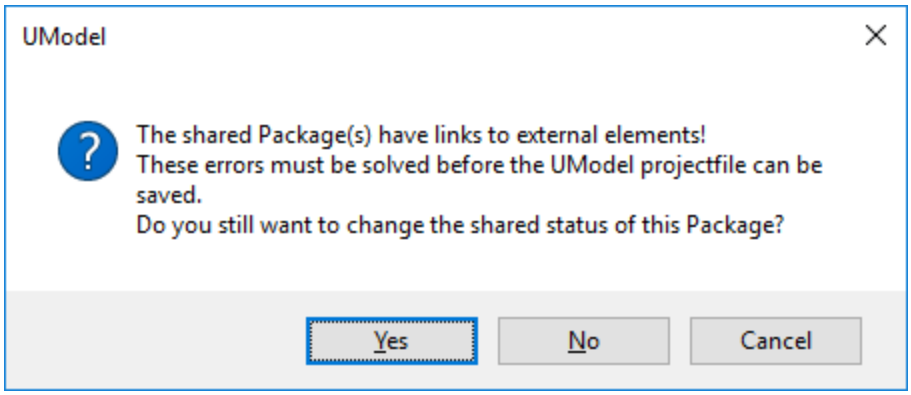


Notes:

- When you include a source project which contains subprojects, all subprojects of the source project will also be included into the target project.
- Shared folders that have been included by reference can be changed to "Include by copy" at any time, by right-clicking the folder and selecting **Subproject | Include as a Copy**.

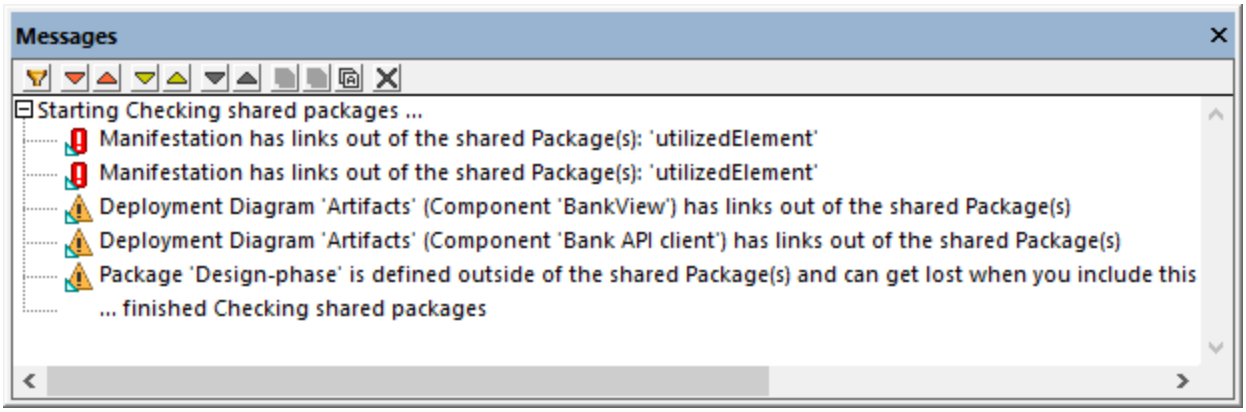
Resolving links to external elements

Attempting to share a package which has links to external elements causes a warning dialog box to appear. For example, the following message appears if you attempt to share the "Deployment View" package of the sample project **C:\Users\<username>\Documents\Altova\UModel2025\UModelExamples\Tutorial\BankView-start.ump**.



Click **Yes** to share the package despite of the errors; otherwise, click **No**. The Messages window provides information about each of the external links.

© 2019-2025 Altova GmbH



Click an entry in the Messages window to display the relevant element in the Model Tree window.

6.1.8 Tips for Enhancing Performance

Some modeling projects can become quite large, in which case there are a few ways you can enhance the modeling performance:

- Make sure that you are using the latest driver for your specific graphics card (resolve this before addressing the following tips)
- Disable syntax coloring (from the Styles window, set the property **Use Syntax Coloring** to **false**).
- Disable "gradient" as a background color for diagrams, use a solid color (from the Styles window, set the property **Diagram background color** to a solid color, for example, white).
- Deactivate automatic completion (go to **Tools | Options | Diagram Editing** and clear the check box **Enable automatic entry helper**).

6.2 Generating Program Code

After you design the model of your application in UModel (for example, one or more class diagrams), you might want to quickly generate a prototype project which includes all defined interfaces, classes, operations, and so on, in your language of choice. UModel enables you to generate C#, VB.NET, or Java program code from a model, based on UML elements found in your UModel project (such as interfaces, classes, operations, and so on). This process is also known as "forward engineering". The generated code will create all objects exactly as they were defined in the model, so that you can proceed to their actual implementation.

Code generation is also applicable to XML schemas and databases*. For example, you could design an XML schema or a database with UModel and then generate the corresponding file (or SQL script, in case of databases) from the model. To achieve this, consult the mapping tables to find out which schema or database elements map to UModel elements, see UModel Element Mappings 224.

** Engineering databases requires UModel Enterprise or Professional editions.*

Prerequisites

In order for code generation to be possible, the UModel project must meet the following minimum requirements:

- One of the packages in your project must be designated as namespace root. The namespace root can be a C#, Java, VB.NET, XSD, or Database namespace. This package must contain all classes and interfaces from which code is to be generated. For more information, see Setting a Package as Namespace Root 170.
- A code engineering component must be added to the project. This component must be realized by all the classes or interfaces from which code is to be generated. For more information, see Adding a Code Engineering Component 171.
- In case of databases, a connection to the target database must be created first, using the menu option **Project | Import SQL database**. Once the connection is established, you can design or modify the database structure in the model and commit the changes to the database through a SQL script. For more information, see UModel and Databases 486..

In addition to this, it is recommended that you include one of the built-in UModel subprojects corresponding to the language (or the language version) you want to use, see Including Subprojects 164. For example, if your application must target a specific version of C#, Java, or VB.NET, this would enable you to use the corresponding data types while designing your UML classes, interfaces, and so on.

For a worked example of how to create a project from scratch and generate code from it, see Example: Generate Java Code 182.

6.2.1 Setting a Package as Namespace Root

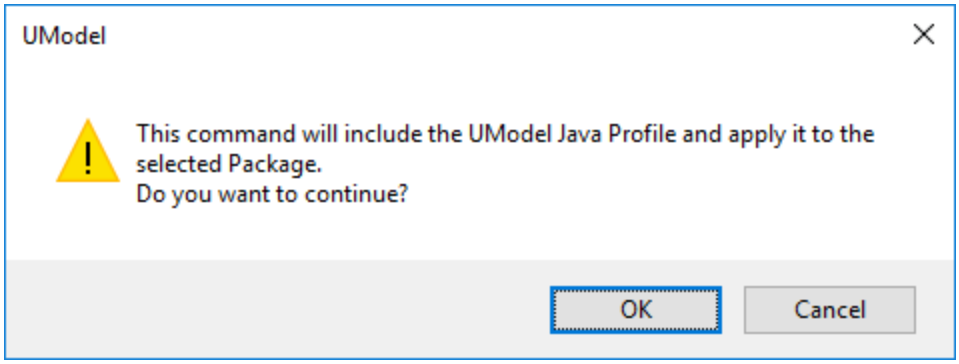
In order to generate program code from your UModel project, a package in your model must be designated as namespace root.

© 2019-2025 Altova GmbH

To set a package as namespace root:

- Right-click a package in the Model Tree Window[83] and select **Code Engineering | Set as <...> Namespace Root** from the context menu, where **<...>** is one of the following: C#, Java, VB.NET, XSD, Database.

When you set a package as namespace root, UModel informs you that the UML profile of the corresponding language will also be added to the project and applied to the selected package. Click OK to confirm when prompted by a dialog box such as the one below.



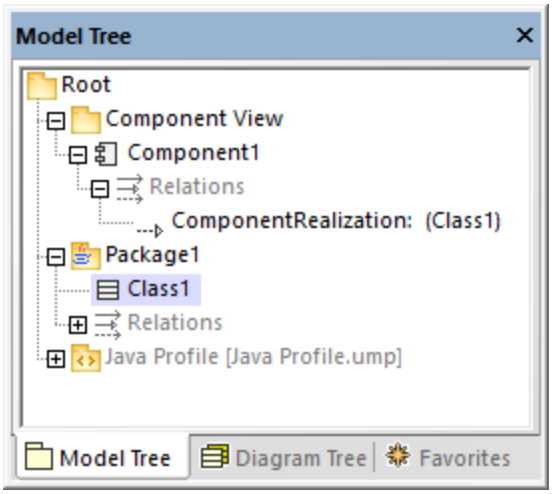
6.2.2 Adding a Code Engineering Component

In order to generate program code, your UModel project must contain a code engineering component that specifies all the code generation details (for example, which classes from the project should be included in code generation, and what should be the target generation directory). As illustrated in the instructions below, the component must meet the following criteria for successful code generation:

- The component must have a physical location (directory) assigned to it. Code will be generated in this directory.
- The classes or interfaces that take part in code engineering must be realized by the component.
- The component must have the property **use for code engineering** enabled.

To add a component which realizes the desired classes or interfaces:

1. Right-click a package in the Model Tree and select **New Element | Component** from the context menu. This adds a new Component to the model.
2. In the model tree, click the class or interface that must be realized by the component, and then drag and drop the cursor onto the component (in this example, `Class1` from `Package1` was dragged onto `Component1`). This automatically creates a `ComponentRealization` relation in the Model Tree.



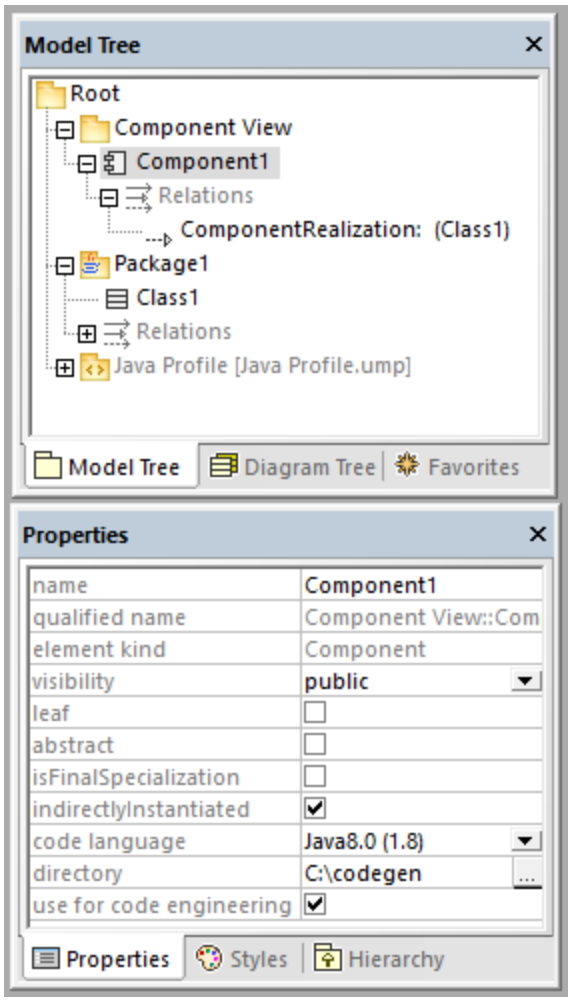
There is also an alternative approach to do this, by creating a Component diagram and then drawing a `ComponentRealization` relation between the component and the classes or interfaces. For more information, see Component Diagrams 53.

To prepare a component for code engineering:

1. Select the component in the Model Tree (it is assumed that this component is already realized by at least one class or interface, as explained above).
2. In the Properties window, locate the **directory** property and set it to the path where you want to generate code.
3. In the Properties window, select the check box **use for code engineering**.

For example, in the image below, the component **Component1** from package **Component View** is configured to generate Java 8.0 code into the directory **C:\codegen**:

© 2019-2025 Altova GmbH



6.2.3 Checking Project Syntax

It is important to check the syntax of the project before generating code from the model. This will inform you of any problems which prevent code from being generated. Project syntax can be checked from the menu command **Project | Check Project Syntax** (alternatively, press **F11**). A syntax check will also be performed automatically before code is updated from the model. The results (errors, warnings, and information messages) are reported in the Messages window.

When a syntax check is performed, the project file is checked on multiple levels as detailed in the tables below. Note the following:

- For information about solving common syntax errors, see the Code generation prerequisites 170.
- For components, the checks below are performed only if the **use for code engineering** property is enabled for the component in the Properties window.
- For classes, interfaces, and enumerations, the checks below are performed only if the class, interface, enumeration is contained in a code language namespace. In other words, it must be under a package which has been defined as namespace root.

- Constraints on model elements are not checked, as they are not part of the code generation process, see Constraining Elements 117.

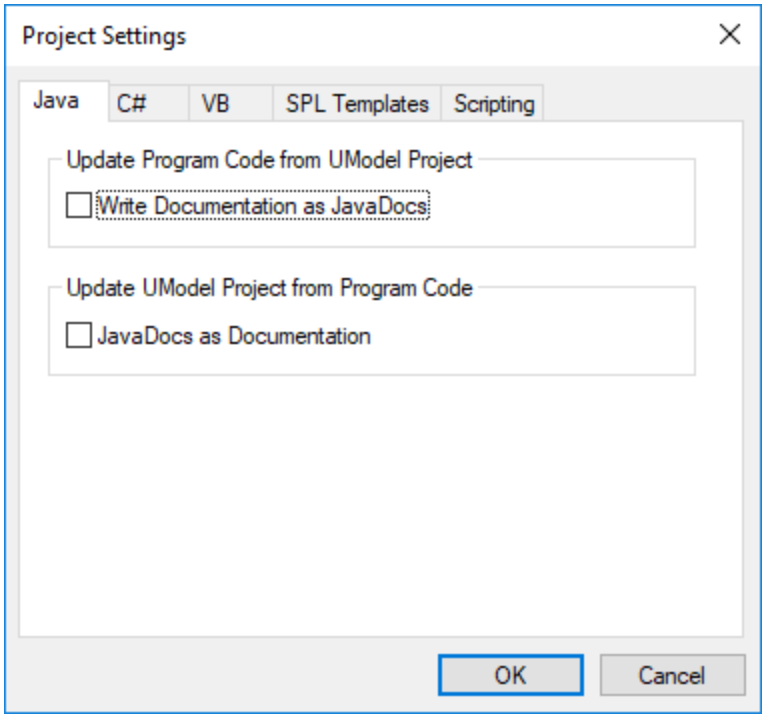
Level	Checks if...	Error severity if check fails
Project	...at least one namespace root package exists.	Error
Component	...project file or directory is set.	Error
	...this component has a `ComponentRealization` relation with at least one class or interface.	Error
Class	...code file name is set. **Note:** This check is not applicable for nested classes.	*Error* if the option **Generate missing code file names** is not set in **Tools | Options | Code Engineering** tab. *Warning* if the option is set.
	...type for operation parameter is set.	Error
	...type for properties is set.	Error
	...operation return type is set.	Error
	...duplicate operations (names + parameter types) exist.	Error
	...a `ComponentRealization` relation exists to a component. **Note:** This check is not applicable for nested classes.	Warning
	...name is valid (no forbidden characters, name is not a keyword)	Error
	...multiple inheritance occurs	Error
Class operation	...name is valid (no forbidden characters, name is not a keyword)	Error
	...a return parameter exists.	Error
Class operation parameter	...name is valid (no forbidden characters, name is not a keyword)	Error
	...type is valid	Error
Interface	...code file name is set.	*Error* if the option **Generate missing code file names** is not set in **Tools | Options | Code Engineering** tab. *Warning* if the option is set.
	...interface is contained in a code language namespace.	Error
	...type for properties are set.	Error

© 2019-2025 Altova GmbH

Level	Checks if...	Error severity if check fails
	...type for operation parameters are set	Error
	...operation return type is set	Error
	...duplicate operations (names + parameter types)	Error
	...interfaces are involved in a ComponentRealization	Warning
	...name is valid (no forbidden characters, name is not a keyword)	Error
Interface operation	...name is valid (no forbidden characters, name is not a keyword)	Error
Interface operation parameter	...name is valid (no forbidden characters, name is not a keyword)	Error
Interface properties	...name is valid (no forbidden characters, name is not a keyword)	Error
Package	...name is valid (no forbidden characters, name is not a keyword) **Note:** This check is applicable if the package is inside a namespace root package and has the <<namespace>> stereotype applied to it from the Properties window.	Error
Enumeration	...a `ComponentRealization` relation exists to a component.	Warning

6.2.4 Code Generation Options

When generating program code into a UModel project, you may want to set or change the options listed below. These options are available when you run the menu command **Project | Project Settings** and are saved together with the project.



The options are grouped into tabs as follows.

Tab	Options
Java	Select the check box **Write Documentation as JavaDocs** to convert the documentation of UModel elements to equivalent JavaDocs-style documentation in generated code.
C#	Select the check box **Write Documentation as DocComments** to convert the documentation of UModel elements to comments in generated C# code.
VB	Select the check box **Write Documentation as DocComments** to convert the documentation of UModel elements to comments in generated VB.NET code.
SPL Templates	If you want to force UModel to read SPL templates from a custom path other than the default one, the custom path must be entered here. See also SPL Templates 190.
Scripting	Options in this tab are only applicable if you developed UModel scripting projects to handle various events or customize the behaviour of your UModel projects. For more information, see Scripting Editor 707.

In addition to the settings above, there are a few other settings which affect code generation. To access them, run the menu command **Tools | Options**, and then click the **Code Engineering** tab. The settings applicable to generating code from a model are grouped under **Update Program Code from UModel Project**. Note that these settings are local (they will only affect the current installation of UModel and will not be saved with the project).

© 2019-2025 Altova GmbH

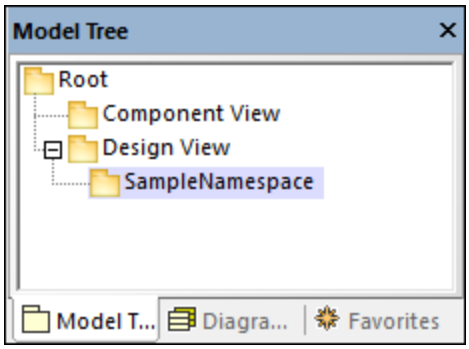
6.2.5 Example: Generate C# Code

This example shows you how to generate C# code with UModel. You will first create a sample C# namespace that contains a couple of classes, configure the project for code generation, and then generate the actual code.

In this example, the target platform is *.NET Standard 2.0 for C# 7.1*. This is possible thanks to a profile built into UModel that defines all the types of *.NET Standard 2.0 for C# 7.1*. UModel also includes built-in profiles for specific .NET Framework versions. For details, see Including Subprojects 164.

Create a new project and its structure

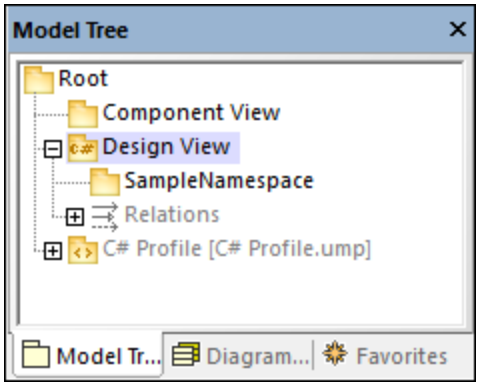
The first step is to create an empty project that has two default packages (`Root` and `Component View`): Click **New** in the **File** menu or in the toolbar. Next, right-click the `Root` package and create a few more packages, as illustrated below. If you are new to the UModel graphical user interface, see the UModel Tutorial 18 and How to Model 108 sections to get started.



In this example, the `Design View` package acts as a container for the design part of your model (e.g., classes and class diagrams), while the `SampleNamespace` package acts as a namespace for all classes that are to be created. In general, you can organize your packages differently.

Code engineering

The next step is to set C# for our package. Right-click the `Design View` package and select **Code Engineering | Set as C# Namespace Root** from the context menu. UModel will inform you that the C# profile will be applied to the package. Click **OK**. The C# profile built into UModel has just been included in the project (*see screenshot below*).

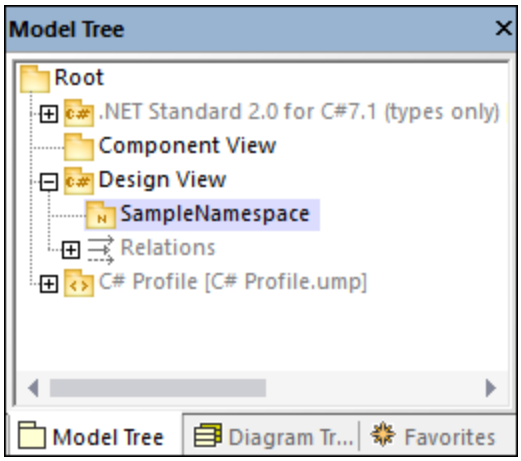


Set SampleNamespace as namespace
Next, click the `SampleNamespace` package and select the `<<namespace>>` check box in the **Properties** window. This applies the `namespace` stereotype to the package, and its icon changes to . You can now create classes under this namespace.

Include a subproject
So far, the model includes the C# profile, which contains the data types applicable to C#. However, the model does not yet include the types specific to .NET Standard 2.0 (these are available in a separate UModel profile). To add this profile to the project, do the following:

1. Go to the **Project** menu and select **Include Subproject**.
2. Switch to the **C#** tab and select *.NET Standard 2.0 for C# 7.1 (types only)*.
3. Select *Include by reference* in the **Include Subproject** dialog and click **OK**.

The additional profile has been added to the project (*see below*).



Create C# classes

The next step is to create classes, which you can do directly in the **Model Tree** pane or from a class diagram. For this example, we have chosen the second option. Follow the steps below:

1. Open the **Diagram Tree** pane.
2. Right-click **Class Diagrams** and select **New Diagram | Class Diagram**.

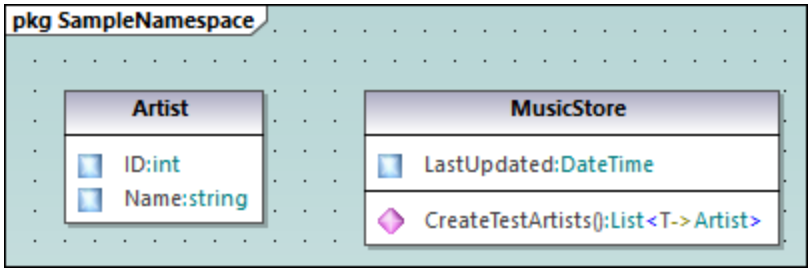
This example assumes that all your classes must be generated under the `SampleNamespace` namespace. Therefore, when prompted to select an owner for the diagram, select the `SampleNamespace` package. If you choose a different package, any elements that you add to the diagram will belong to the same package as the diagram (which may or may not be the intended goal).

Create classes and their structure
Next, create classes, types, and other elements required in your model. For our example, you can create a simple diagram that contains an `Artist` class and a `MusicStore` class (*see screenshot below*). Follow the instructions below:

1. Right-click inside the *pkg SampleNamespace* window and select **New | Class**.
2. Name this class `Artist`.

© 2019-2025 Altova GmbH

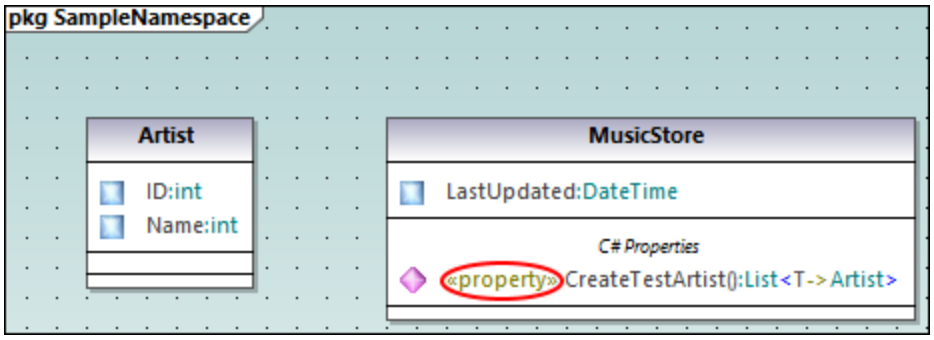
3. Right-click inside the `Artist` box and create two properties: `ID` of type `int` and `Name` of type `string`.
4. Create the second class called `MusicStore`.
5. Create a property called `LastUpdated` of type `DateTime`.
6. Create an operation and type its name and definition as shown below.

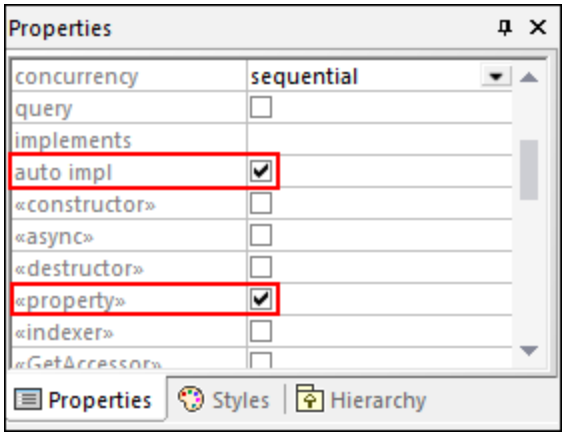


For more information about designing classes and their members, see the Class Diagrams[31] and How to Model[108] sections.

About auto-implemented C# properties

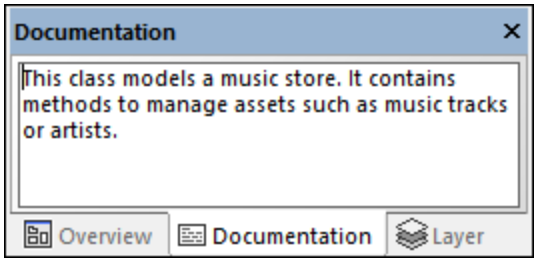
In UModel, you can see whether C# properties have been auto-implemented. The auto-implementation option becomes available after the `property` check box has been selected (for `CreateTestArtist()` in our example) in the **Properties** window (*see screenshots below*).





Add documentation (optional)

Optionally, click the `MusicStore` class in the diagram and add some documentation by typing the text in the Documentation window[94] (*see screenshot below*). This lets you generate code comments for this class.

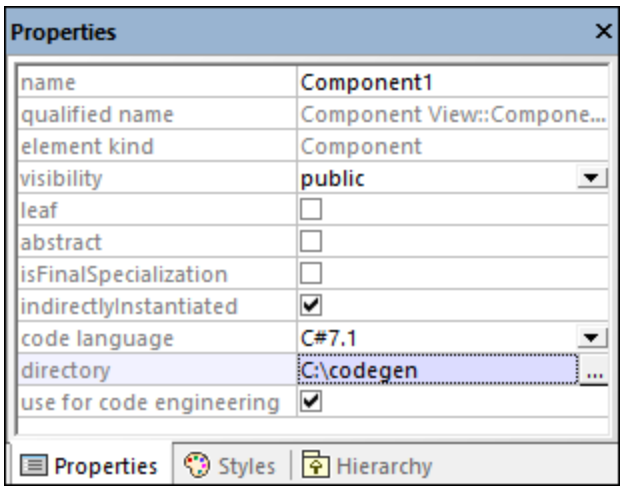


Configure the project for code engineering

In the next step, we need to define code engineering settings. Take the steps below:

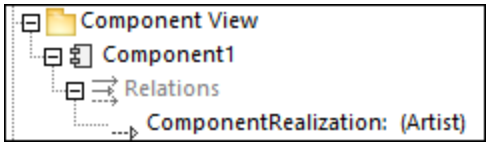
1. Save the project to a directory.
2. Then right-click the `Component View` package in the **Model Tree** pane and add a new **Component** (that is, a software component) to it.
3. Click the new software component and set the following properties in the **Properties** window (*see screenshot below*):

   - Set the code language of the component to C# 7.1, for example.
   - Select the code generation directory (`C:\codegen` in our example).
   - Select the *use for code engineering* check box.



Create a ComponentRealization relationship

Next, create a `ComponentRealization` relationship between the classes from which C# code must be generated. This can be done as follows: In the **Model Tree** pane, click the class to be realized by the component (`Artist` in this example), then drag and drop it into the code engineering component (`Component1`) (*see screenshot below*). Take the same step for the `MusicStore` class.



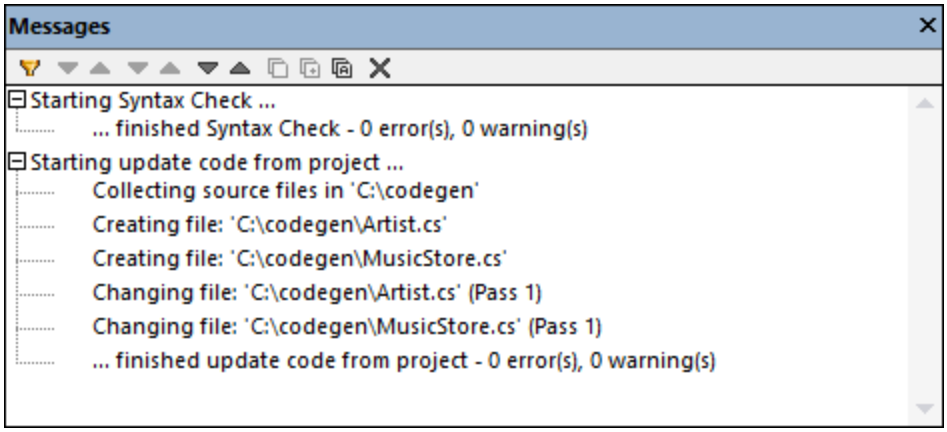
© 2019-2025 Altova GmbH

Note: In case you forget to create a `ComponentRealization` relationship for a class, UModel still generates the corresponding code file, even though warnings will be issued in the **Messages** window. This setting is configurable from **Tools | Options | Code Engineering** tab (the *Generate missing ComponentRealizations* check box).

Generate C# code

The final step is to generate the actual C# code. Take the steps below:

1. Go to the **Project** menu and click **Merge Program Code from UModel Project**. A dialog box appears where you can adjust whether changes in code should be merged with those in the code or overwrite them (if applicable). For the scope of this example, you can select **Overwrite** since a new project is getting generated.
2. To include the class documentation as comments in the generated code, click **Project | Project Settings** and select the **Write Documentation as DocComments** check box. For more information, see Code Generation Options 175.
3. Click **OK**. The **Messages** window displays the code engineering result (*see below*).



If you have added any documentation to the `MusicStore` class, notice that it appears as code comments in the generated code:

```
using System;
using System.Collections.Generic;
namespace SampleNamespace
{
   /// This class models a music store. It contains methods to manage assets such as
music tracks or artists.
   public class MusicStore
   {
      public DateTime LastUpdated;
      public List<Artist> CreateTestArtists()
      {
         // TODO add implementation
      }
   }

}
```

6.2.6 Example: Generate Java Code

This example illustrates how to create a new UModel project and generate program code from it (a process known as "forward engineering"). For the sake of simplicity, the project will be very simple, consisting of only one class. You will also learn how to prepare the project for code generation and check that the project uses the correct syntax. After generating program code, you will modify it outside UModel, by adding a new method to the class. Finally, you will learn how to merge the code changes back into the original UModel project (a process known as "reverse engineering").

The code generation language used in this tutorial is Java; however, similar instructions are applicable for other code generation languages.

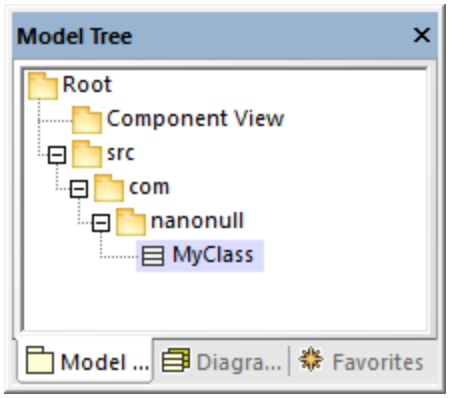
Creating a new UModel project

You can create a new UModel project as follows:

- On the **File** menu, click **New**. (Alternatively, press **Ctrl+N**, or click the New toolbar button.)

At this stage, the project contains only the default "Root" and "Component View" packages. These two packages cannot be deleted or renamed. "Root" is the top grouping level for all other packages and elements in the project. "Component View" is required for code engineering; it typically stores one or more UML components that will be realized by the classes or interfaces of your project; however, we didn't create any classes yet. Therefore, let's first design the structure of our program, as follows:

1. Right-click the "Root" package in the Model Tree window and select **New Element | Package** from the context menu. Rename the new package to "src".
2. Right-click "src" and select **New Element | Package** from the context menu. Rename the new package to "com"
3. Right-click "com" and select **New Element | Package** from the context menu. Rename the new package to "nanonull".
4. Right-click "nanonull" and select **New Element | Class** from the context menu. Rename the new class to "MyClass".



Preparing the project for code generation

To generate code from a UModel model, the following requirements must be met:

© 2019-2025 Altova GmbH

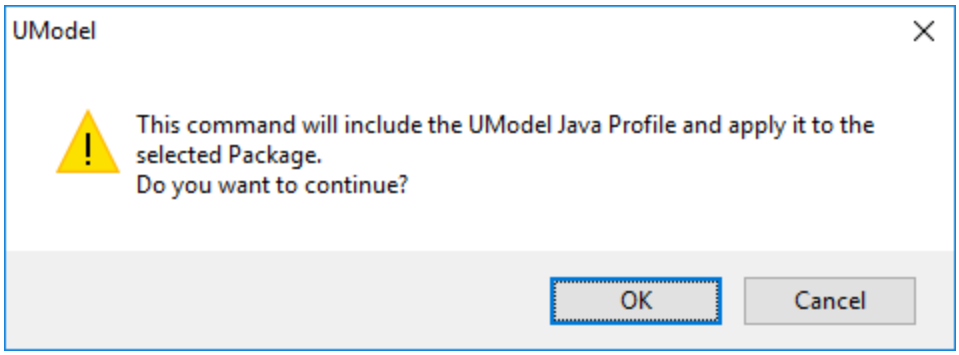
- A Java, C#, or VB.NET namespace root package must be defined.
- A component must exist which is realized by all classes or interfaces for which code must be generated.
- The component must have a physical location (directory) assigned to it. Code will be generated in this directory.
- The component must have the property **use for code engineering** enabled.

All of these requirements are explained in more detail below. Note that you can always check if the project meets all code generation requirements, by validating it:

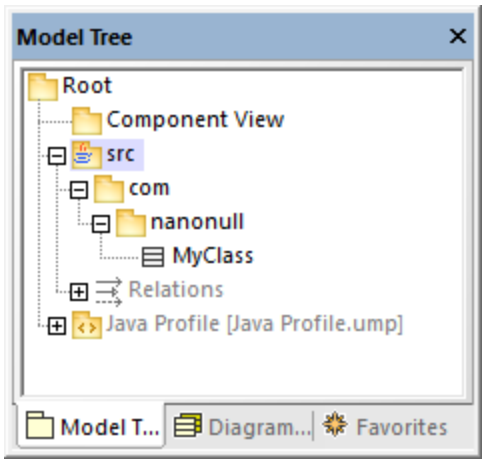
- On the **Project** menu, click **Check Project Syntax**. (Alternatively, press **F11**.)

If you validate the project at this stage, the Messages window displays a validation error ("*No Namespace Root found! Please use the context menu in the Model Tree to define a Package as Namespace Root*"). To resolve this, let's assign the package "src" to be the namespace root:

- Right-click the "src" package and select **Code Engineering | Set As Java Namespace Root** from the context menu.
- When prompted that the UModel Java Profile will be included, click **OK**.



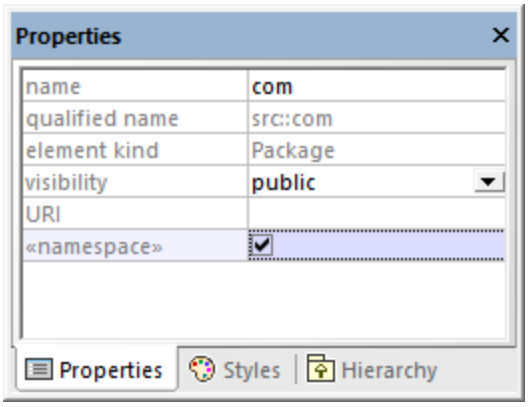
Notice the package icon has now changed to , which signifies that this package is a Java namespace root. Additionally, a Java Profile has been added to the project.



The actual namespace can be defined as follows:

1. Select the package "com" in the **Model Tree** window.

2. In the **Properties** window, enable the **<<namespace>>** property.

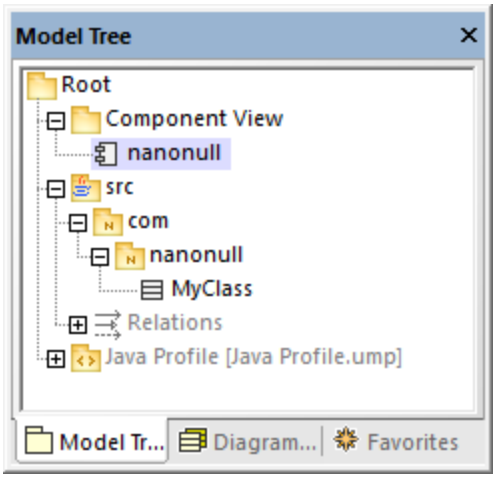


3. Repeat the step above for the "nanonull" package.

Notice that the icon of both "com" and "nanonull" packages has now changed to , which indicates these are now namespaces.

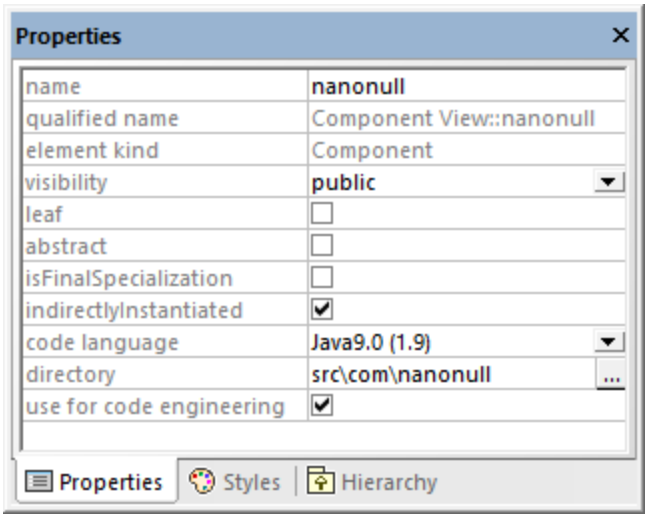
Another requirement for code generation is that a component must be realized by at least a class or an interface. In UML, a component is a piece of the system. In UModel, the component lets you specify the code generation directory and other settings; otherwise, code generation would not be possible. If you validate the project at this stage, a warning message is displayed in the **Messages** window: "*MyClass has no ComponentRealization to a Component - no code will be generated*". To solve this, a component must be added to the project, as follows:

1. Right-click "Component View" in the Model Tree window, and select **New Element | Component** from the context menu.
2. Rename the new Component to "nanonull".



3. In the **Properties** window, change the **directory** property to a directory where code should be generated (in this example, "src\com\nanonull"). Notice that the property **use for code engineering** is enabled, which is another prerequisite for code generation.

© 2019-2025 Altova GmbH



4. Save the UModel project to a directory and give it a descriptive name (in this example, **C:\UModelDemo\Tutorial.ump**).

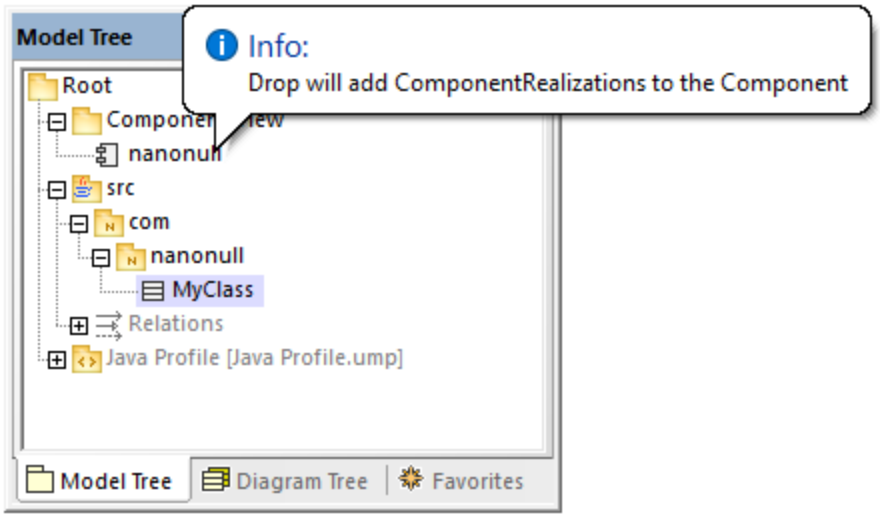
Note: The code generation path can be absolute or relative to the .ump project. If it is relative as in this example, a path such as **src\com\nanonull** would create all the directories in the same directory where the UModel project was saved.

We have deliberately chosen to generate code to a path which includes the namespace name; otherwise, warnings would occur. By default, UModel displays project validation warnings if the component is configured to generate Java code to a directory which does not have the same name as the namespace name. In this example, the component "nanonull" has the path "C:\UModelDemo\src\com\nanonull", so no validation warnings will occur. If you want to enforce a similar check for C# or VB.NET, or if you want to disable the namespace validation check for Java, do the following:

1. On the **Tools** menu, click **Options**.
2. Click the **Code Engineering** tab.
3. Select the relevant check box under **Use namespace for code file path**.

The component realization relationship can be created as follows:

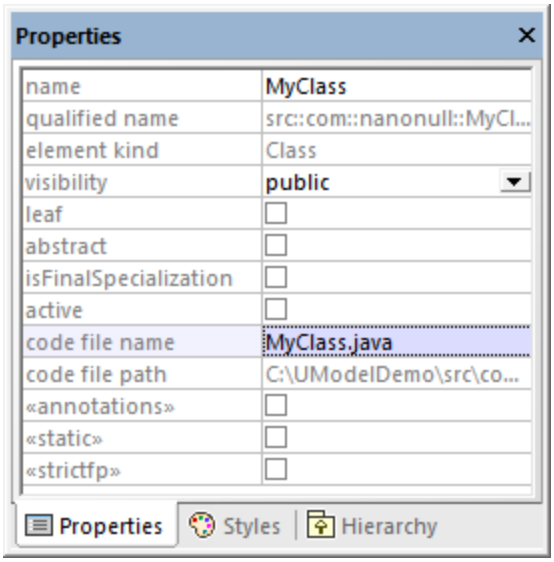
- In the **Model Tree** window, drag from the `MyClass` created previously and drop onto component `nanonull`.



The component is now realized by the project's only class `MyClass`. Note that the approach above is just one of the ways to create the component realization. Another way is to create it from a component diagram, as illustrated in the tutorial section Component Diagrams 53.

Next, it is recommended that the classes or interfaces which take part in code generation have a file name. Otherwise, UModel will generate the corresponding file with a default file name and the **Messages** window will display a warning ("*code file name not set - a default name will be generated*"). To remove this warning:

1. Select the class `MyClass` in the **Model Tree** window.
2. In the **Properties** window, change the property **code file name** to the desired file name (in this example, `MyClass.java`).



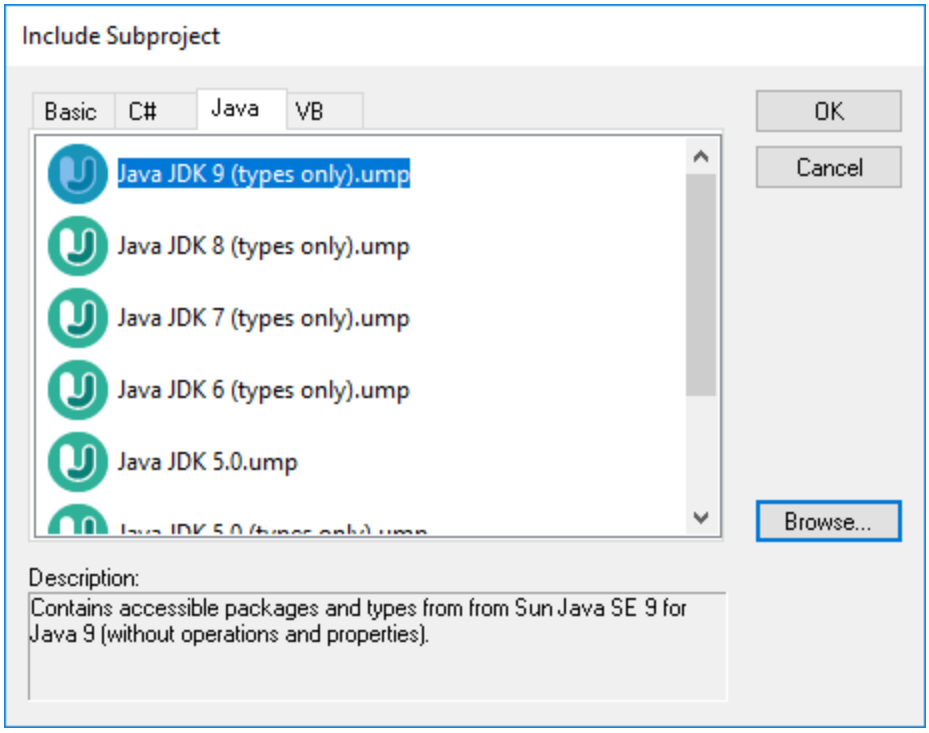
Including the JDK types

Although this step is optional, it is recommended that you include the Java Development Kit (JDK) language types, as a subproject of your current UModel project. Otherwise, the JDK types will not be available when you

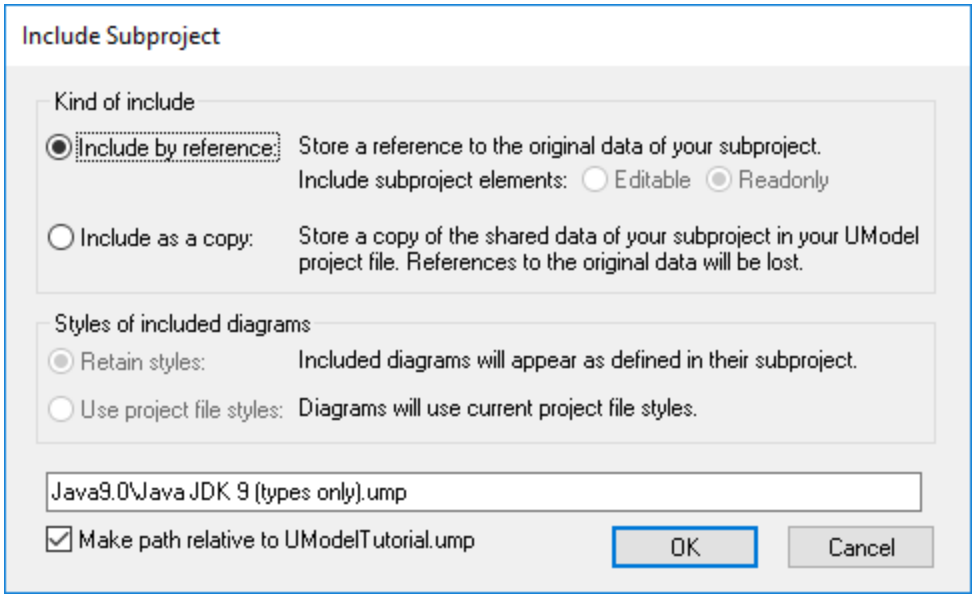
© 2019-2025 Altova GmbH

create the classes or interfaces. This can be done as follows (the instructions are similar for C# and VB.NET):

1. On the **Project** menu, click **Include Subproject**.
2. Click the **Java** tab and select the **Java JDK 9 (types only)** project.



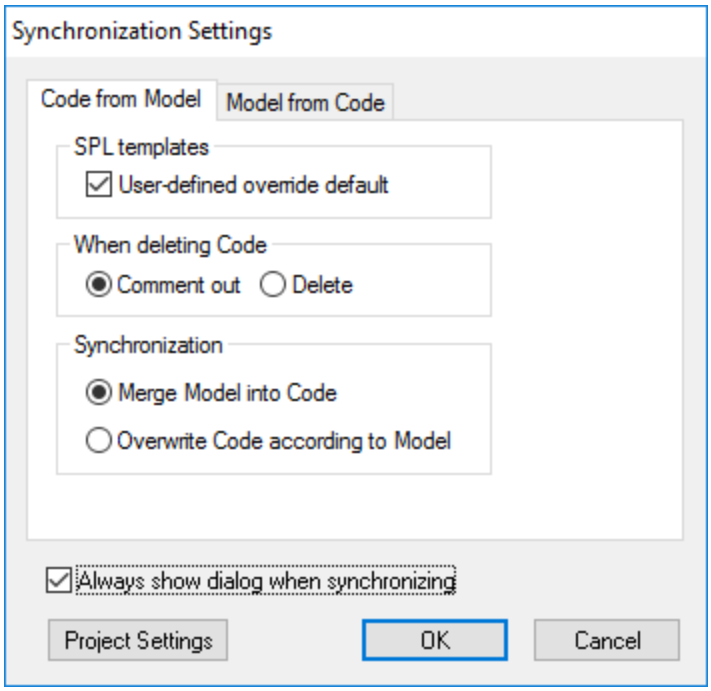
3. When prompted to include by reference or as a copy, select **Include by reference**.



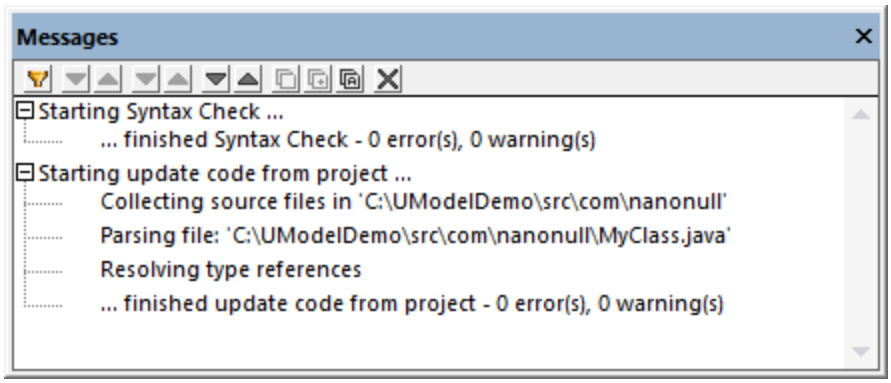
Generating code

Now that all prerequisites have been met, code can be generated as follows:

1. On the **Project** menu, click **Merge Program Code from UModel Project**. (Alternatively, press **F12**.) Note that this command will be called **Overwrite Program Code from UModel Project** if the **Overwrite Code according to Model** option was selected previously on the "Synchronization Settings" dialog box illustrated below.



2. Leave the default synchronization settings as is, and click **OK**. A project syntax check takes place automatically, and the **Messages** window informs you of the result:



Modifying code outside of UModel

Generating program code is just the first step to developing your software application or system. In a real life scenario, the code would go through many modifications before it becomes a full-featured program. For the scope of this example, open the generated file **MyClass.java** in a text editor and add a new method to the class, as shown below. The **MyClass.java** file should look as follows:

```
package com.nanonull;
```


© 2019-2025 Altova GmbH

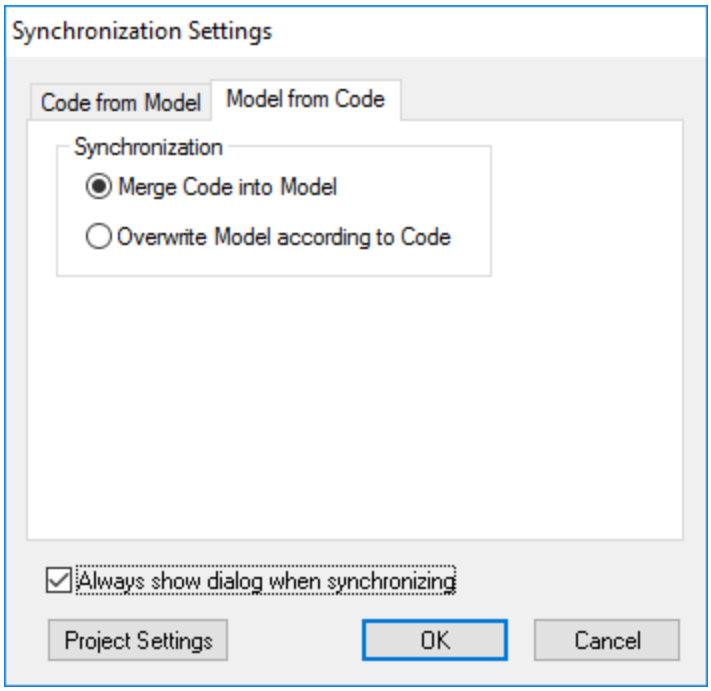
```
public class MyClass{
   public float sum(float num1, float num2){
    return num1 + num2;
  }
}
```

MyClass.java

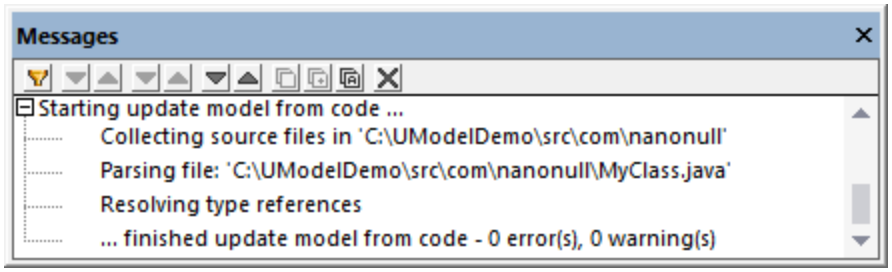
Merging code changes back into the model

You can now merge the code changes back into the model, as follows:

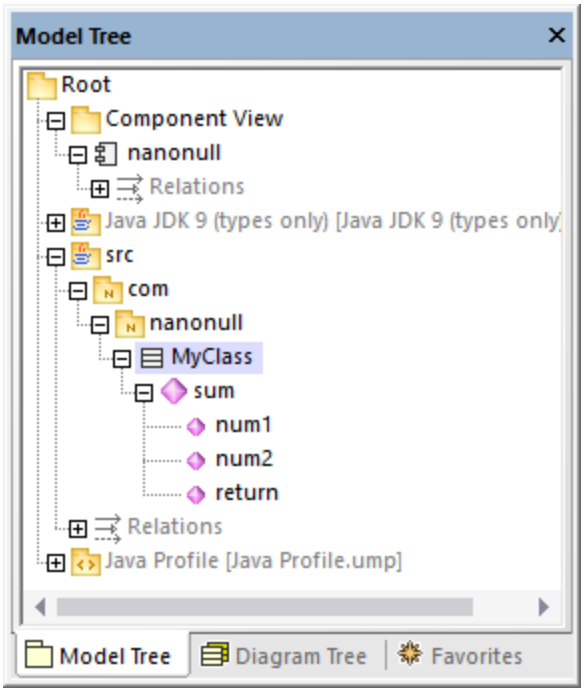
1. On the **Project** menu, click **Merge UModel Project from Program Code** (Alternatively, press **Ctrl + F12**).



2. Leave the default synchronization settings as is, and click OK. A code syntax check takes place automatically, and the **Messages** window informs you of the result:



The operation `sum` (which has been reverse engineered from code) is now visible in the Model Tree window.



6.2.7 SPL Templates

When generating C#, Java, or VB.NET code, as well as XSD schemas, UModel uses a templating language called SPL (Spy Programming Language). The SPL templates dictate the syntax of the generated code files. It is possible to customize the SPL templates, for example, in order to slightly change the syntax of the generated code. Editing SPL templates is meaningful only for languages supported by UModel. If you want to create completely new SPL templates for other languages, it would be possible to generate new code but it would not be possible to update existing code (since the language syntax would be unknown to UModel).

The default SPL templates are available in the **UModelSPL** directory relative to the program installation directory.

> Do not modify the existing default SPL templates, since these directly affect the default code generation. Should you need to customize code generation, create custom templates instead, as shown below.

SPL templates are only used when new code is generated (that is, when new classes, operations etc have been added to the model, and then code generation takes place). Any existing code is not affected by the SPL templates.

For an introduction to SPL, see SPL Reference 1274.

To modify the provided SPL templates:

1. Locate the provided SPL templates in the UModel installation directory ("Program Files"), for example: **...\UModel2025\UModelSPL\Java\Default**.

 © 2019-2025 Altova GmbH

2. Copy the SPL files you want to modify into the **parent** directory. For example, if you want to modify the appearance of a Java class in generated code, copy the **Class.spl** file from **...\UModel2025\UModelSPL\Java\Default** to **...\UModel2025\UModelSPL\Java**.
3. Make the changes to the .spl file(s) and save them.

To use the custom SPL templates:

1. Select the menu option **Project | Synchronization settings**.
2. Select the **User-defined override default** check box in the SPL templates group.

6.3 Importing Source Code

Existing Java, C#, and VB.NET program code can be imported into UModel (a process also known as "reverse engineering"). The following project types can be imported into UModel:

- Java projects (Eclipse .project files, NetBeans project.xml files, and JBuilder .jpx files)
- C# and VB.NET projects (Visual Studio .sln, .csproj, .csdprj, .vbproj, .vbp as well as Borland .bdsproj project files)

In addition to importing source code from a source project, it is also possible to import code from a source directory. Importing from a source directory works in a similar way, and is particularly useful when your code doesn't use any of the project types listed above. For an example of importing a source directory, see Reverse Engineering (from Code to Model) 73.

It is possible to import source code either into a new, empty UModel project or into an existing UModel project. During the import, you can specify whether the imported elements should overwrite those in the model (if any), or be merged into the model. Optionally, Class and Package diagrams can be generated automatically as you import code.

The import wizard includes various import options specific to each platform (Java, .NET). For example, if the imported Java/C#/VB.NET code contains comments, these can be optionally converted to UModel documentation. For a complete list of options, see Code Import Options 194 .

Once your C#, VB.NET, or Java code has been imported into UModel, it is possible to modify the model (for example, add new classes, or rename properties and operations), and optionally synchronize it back with the original code, thus achieving full round-trip engineering, see Synchronizing the Model and Source Code 217.

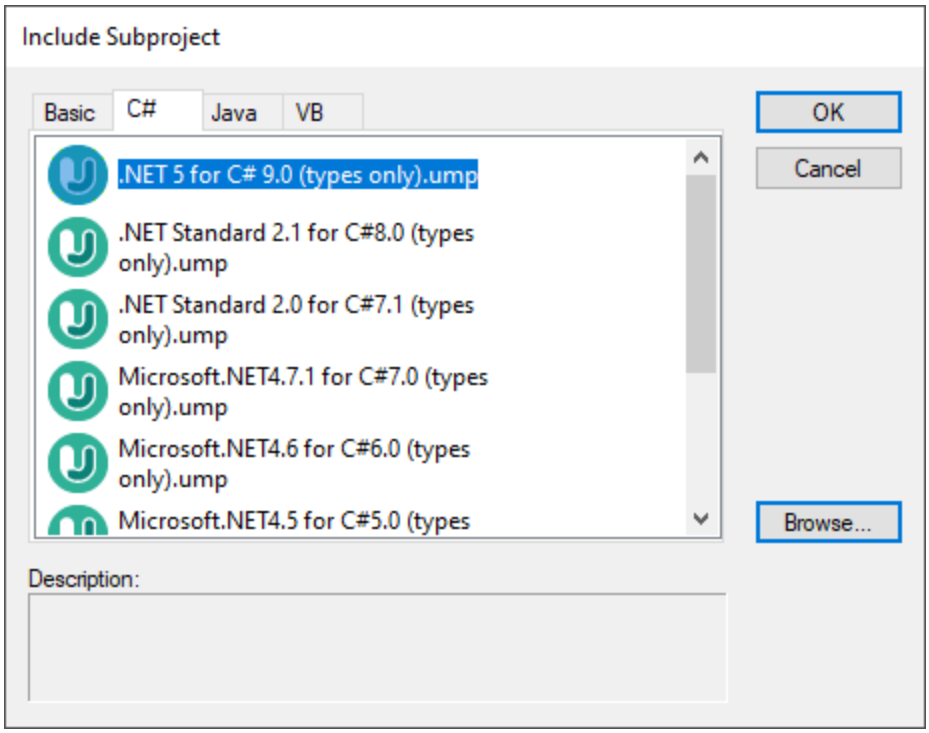
Prerequisites

UModel includes several built-in sub-projects that were created specifically for code engineering and which include the data types applicable to each supported language and platform. Before attempting to import source code into a UModel project, it is recommended to include the built-in UModel subproject applicable to the corresponding programming language and platform, see Including Subprojects 164. Otherwise, certain data types will not be recognized and will be placed after import into a separate package called "Unknown externals".

To include a subproject with the required language data types:

1. On the **Project** menu, click **Include Subproject**.
2. Click the tab applicable to the source language and platform (for example, Java 8.0, C# 6.0, VB 9.0), and then click OK.

© 2019-2025 Altova GmbH



Note the following:

- When you include a data type subproject for a particular language, UModel also automatically adds the profile of that language to your project. The profile subproject (.ump) contains only the most basic types and is different from the data type subproject (also .ump) which contains more extensive type definitions.
- If you perform the import without including a data type subproject, the import operation will take place nonetheless, and UModel will also automatically include the profile of that language to the project. However, any unknown types will be placed into the "Unknown externals" package. To solve this, make sure to include the data types subproject for the required language and platform, as explained above.

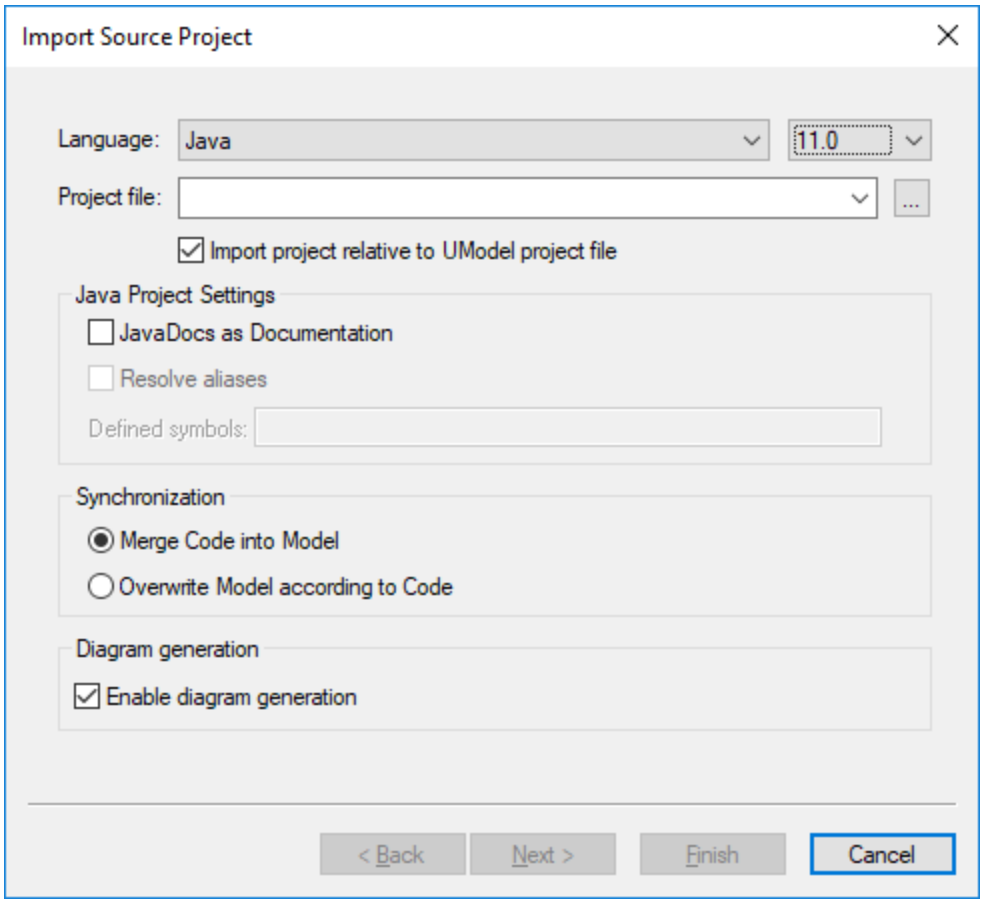
Importing source code from a project

1. On the **Project** menu, click **Import Source Project**. (Alternatively, if you would like to import code from an existing directory, select **Import Source Directory**.)
2. Select the language version of the source project (for example, Java 8.0, C# 6.0).
3. Click **Browse** ... and select the source project file.
4. Set or change the required import options, see also Code Import Options 194 (note that these options depend on the language selected in step 2).
5. Click **Finish** to complete the wizard.

For a step-by-step example, see Example: Import a C# Project 196.

6.3.1 Code Import Options

When importing program code into a UModel project, you may need to set or change the options listed below. These options are available on the dialog box which appears when you run the menu command **Project | Import Source Project** or **Project | Import Source Directory**.



Import Source Project dialog box

Most of the options on the dialog box above can also be changed at any time later, see Code Synchronization Settings 221.

The following options are applicable to all project types, regardless of the language or platform:

Option	Description
Import project relative to UModel project file	By default, this option is selected, which means that a relative path dependency will be established between the UModel project and the imported source code project. After source code is imported, a UML component is generated automatically in the UModel project (it is available in the Model Tree, as a child of "Component View"). This component realizes

 © 2019-2025 Altova GmbH

Option	Description
	the interfaces or classes to be engineered; it also specifies the code engineering options, including the path to the source project or directory. This will be a relative path if **Import project relative to UModel project file** is selected; otherwise, it will be an absolute path.
Merge Code into Model / Overwrite Model according to Code	If **Merge...** is selected, potential name conflicts (such as package or class names) will be resolved by appending a number to the element that is being imported. If **Overwrite...** is selected, and if there are name conflicts, the imported element will take precedence over (overwrite) the one existing in the project.
Enable diagram generation	Optionally, select this check box if you want to generate Class and Package diagrams from the imported classes. When this check box is selected, the import wizard includes additional steps which enable you to customize the look of the generated diagrams.

The following options are applicable only to C# and VB.NET projects:

Option	Description
DocComments as Documentation	Select this check box to convert comments found in the C# code into UModel element documentation (see also Documentation 94).

Resolve aliases	This check box is enabled by default. If your C# or VB.NET code contains namespace or class aliases like in the code listing below, it is recommended to keep this check box selected. Otherwise, associations and dependencies involving aliased classes and namespaces in your code may not be detected automatically by UModel during the import (and thus would not be present in the model).
	
	`using Q = System.Collections.Generic.Queue<String>;`
	`Q myQueue;`
	
	Example of an alias in C# code
	
	During the source code import, any potentially conflicting aliases are added to the "Unknown externals" package of the UModel project if their use is unclear.
	
	When you update the code back from the model (round-trip engineering), aliases will be retained as they exist in the generated code.
	
	The **Resolve aliases** option can be changed at any time later, see Code Synchronization Settings 221. If you enable this option after (not before) the import operation, UModel prompts you to update the project from the code again, since the option also has consequences for forward engineering.

Option	Description

Defined symbols	If your C# or VB.NET code includes symbols that are defined through preprocessor directives such as `#if`, `#endif`, you can instruct UModel to take them into account while reverse engineering code.
	
	`#if DEBUG`
	`  static void DisplayMessage()`
	`  {`
	`  Console.WriteLine("Please wait...");`
	`  }`
	`#endif`
	
	Example of a conditional compilation symbol in C# code
	
	For example, if you reverse engineer the code above, the method `DisplayMessage()` will only be imported into the model if you specified the `DEBUG` symbol.
	
	To specify conditional compilation symbols, enter them in the "Defined symbols" text box, delimited by a semicolon.
	
	During the reverse engineering process, UModel outputs all symbols used in the source code in the Messages window.

The following option is applicable only to Java projects:

Option	Description
JavaDocs as Documentation	Select this check box to convert JavaDocs-style comments found in the code into UModel element documentation (see also Documentation 94). **Note:** Only comments applicable for Java classes, interfaces, operations, and properties are converted.

6.3.2 Example: Import a C# Project

This example illustrates how to import into UModel a sample C# solution created with Visual Studio. The source solution is available as a .zip archive at the following path: **C:\Users\<username>\Documents\Altova\UModel2025\UModelExamples\Tutorial\Anagram_CSharp.zip**. It is not necessary to compile the solution with Visual Studio before importing it; however, make sure to unzip the **Anagram_CSharp.zip** archive to a folder of your choice before proceeding to the steps below.

Our goal in this example is to reverse engineer the C# solution and create a UModel project from it. As we import code, we will opt to generate class and package diagrams automatically.

Step 1: Create a new project

It is possible to import source code either into existing or new UModel projects. For the scope of this example, we will be importing code into a new UModel project.

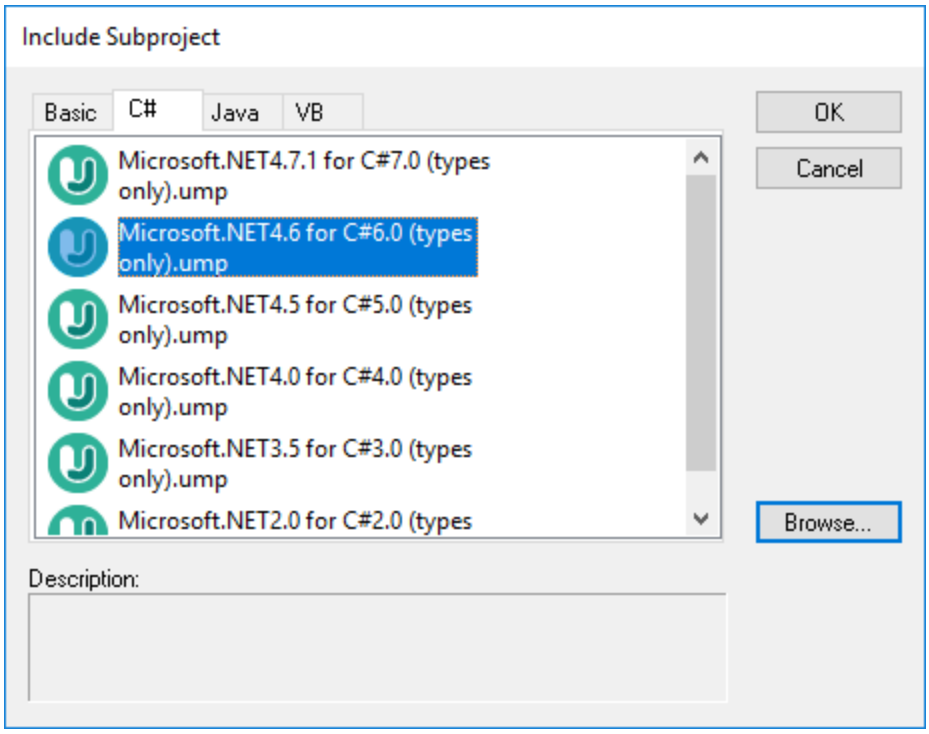
© 2019-2025 Altova GmbH

- On the **File** menu, click **New** (Alternatively, press **Ctrl + N** or click the **New** toolbar button).

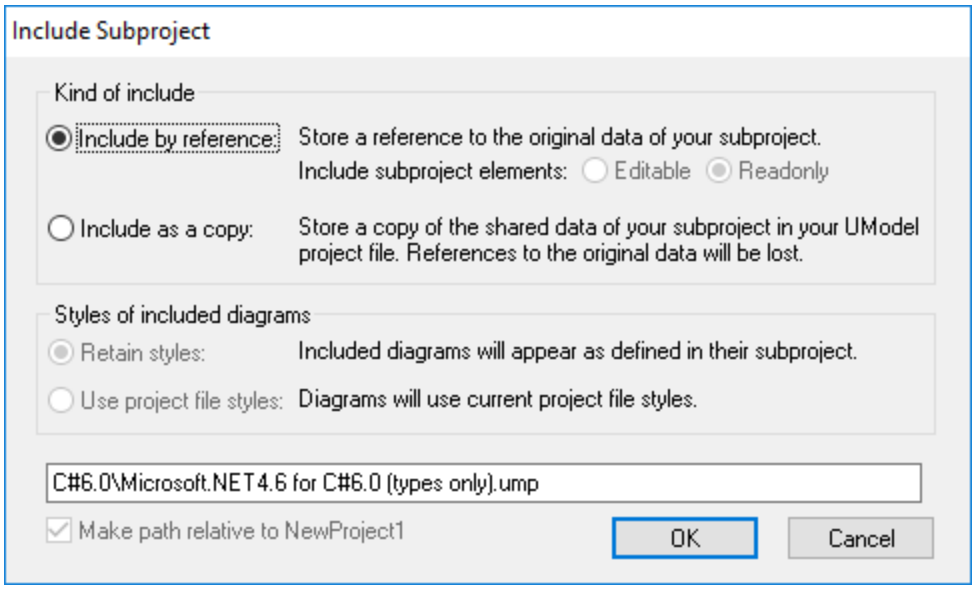
Step 2: Include the C# language types

The source project was written in C# with Visual Studio 2015, so we will include a built-in UModel project that contains the C# 6.0 language types (since the C# language version corresponding to Visual Studio 2015 is 6.0). Earlier versions of C# are also likely to work with our C# example solution.

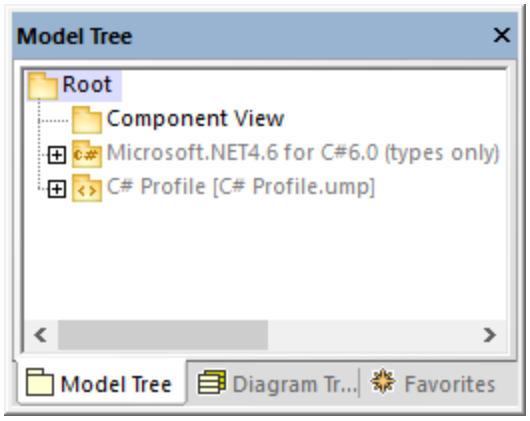
1. On the **Project** menu, click **Include Subproject**.
2. Click the **C#** tab.



3. Select the project **Microsoft .NET 4.6 for C# 6.0 (types only).ump**, and click **OK**.
4. When prompted to select the kind of include (by reference or as a copy), leave the default option as is.



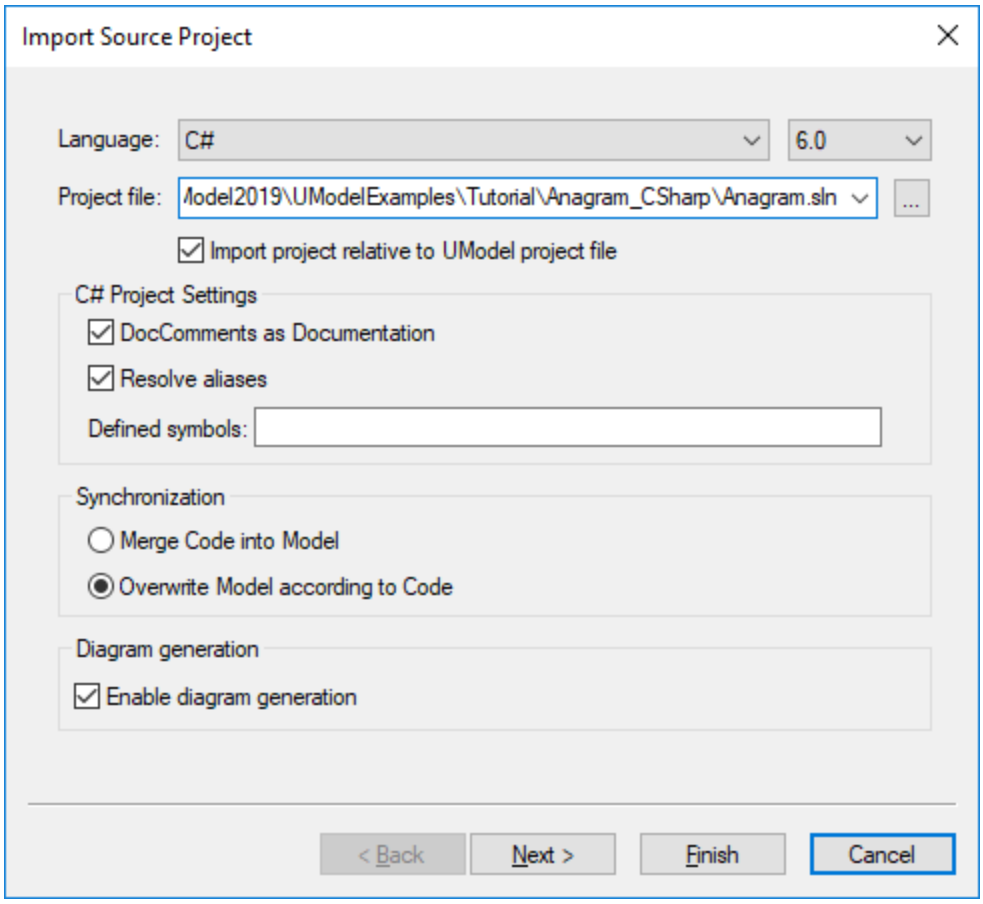
As a result, both the C# language types and the C# language profile are included and visible in the Model Tree:



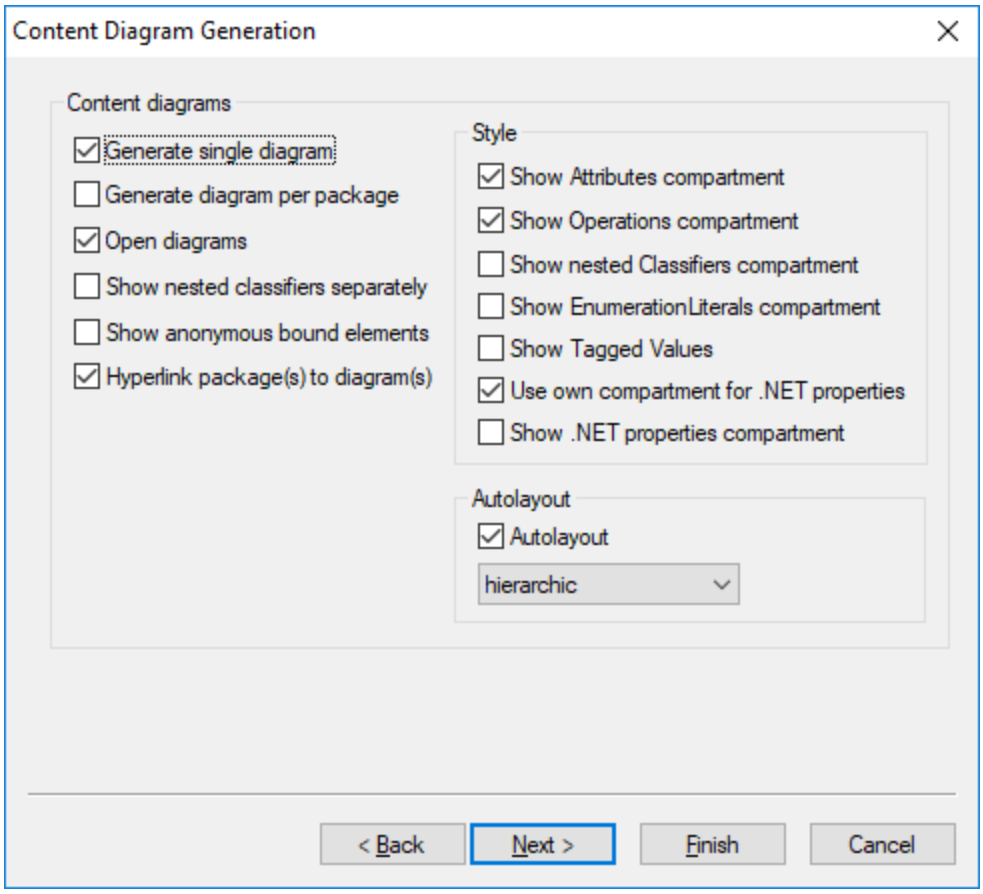
Step 3: Import the C# solution

1. On the **Project** menu, click **Import Source Project**.

© 2019-2025 Altova GmbH

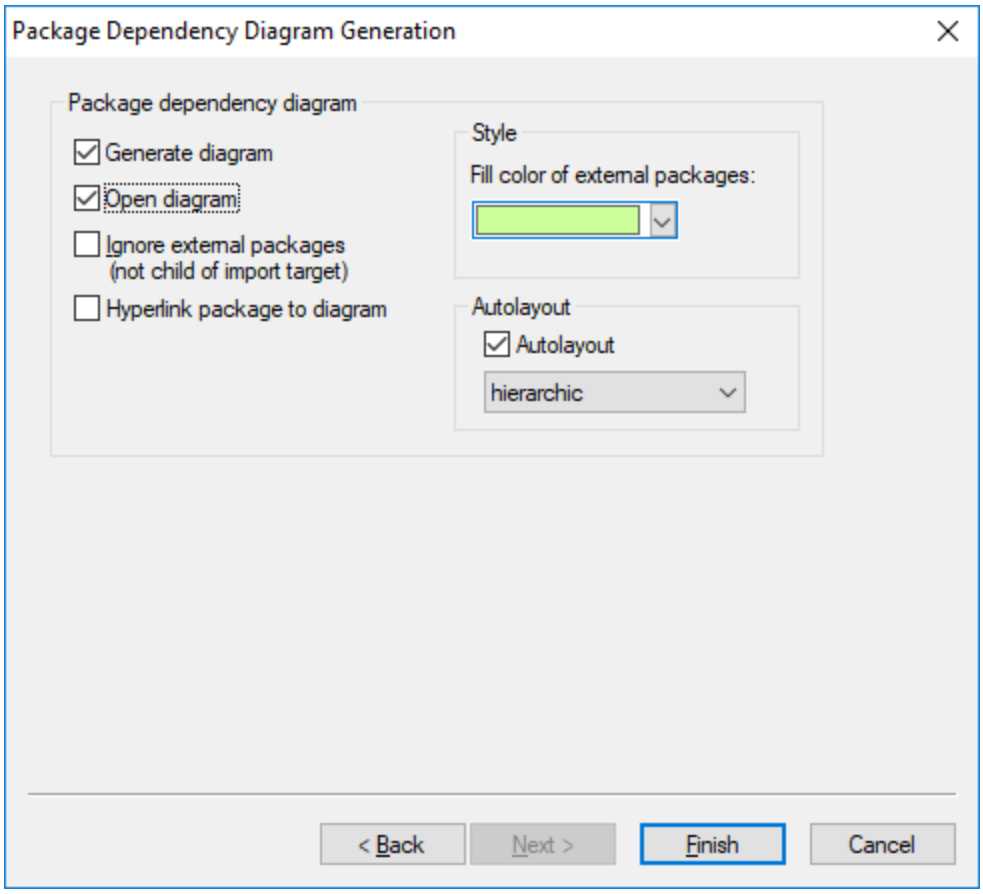


2. Select **C# 6.0** as language.
3. Click **Browse** ... next to **Project file** and browse for the solution .sln file.
4. Select the **DocComments as Documentation** check box (this will import the code comments found on operations or properties into the model).
5. Since we are importing code into a new UModel project, select the option **Overwrite Model according to Code** (the other option **Merge Code into Model** is preferable when you import into an existing project).
6. Click **Next**.
7. Select the diagram generation options as shown below, and click **Next**. (These options are applicable to Class diagrams generated automatically on code import.)

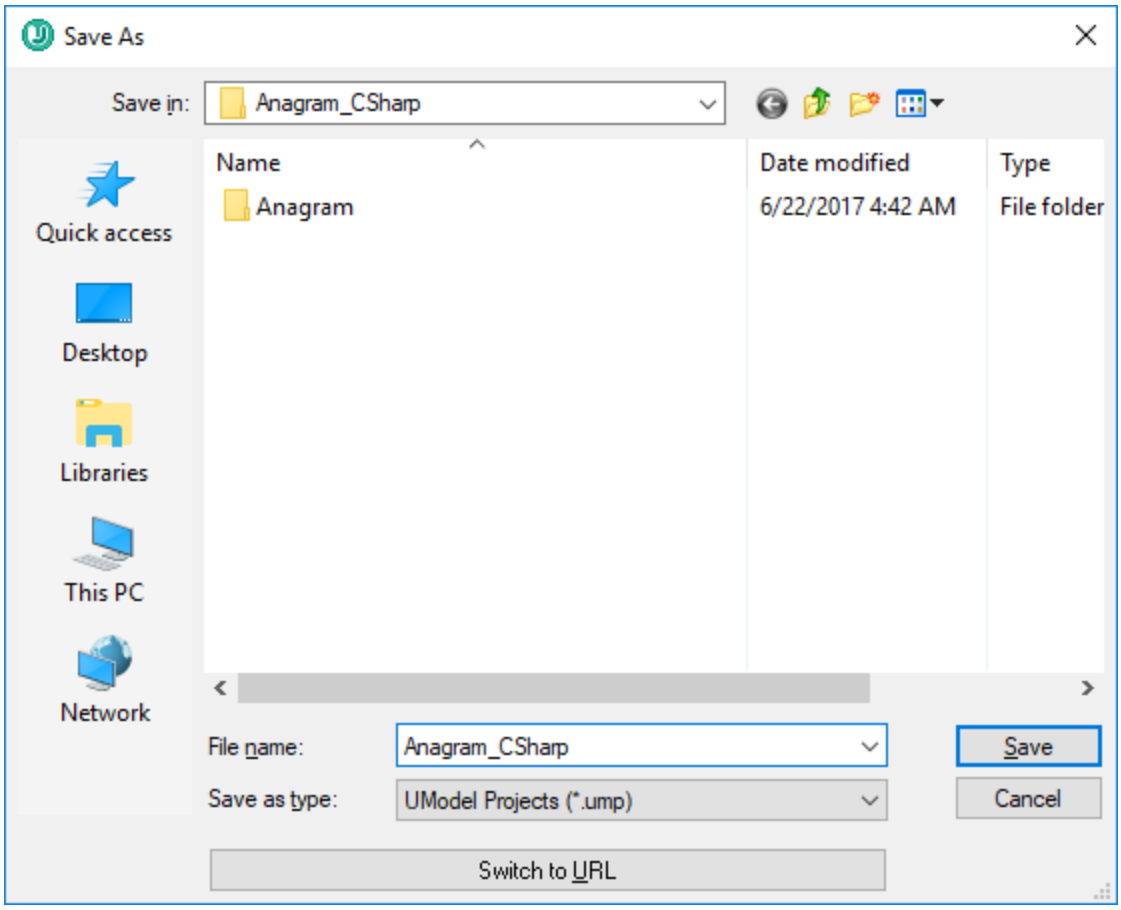


8. Select the diagram generation options as shown below, and click **Finish**. (These options are applicable to Package diagrams generated automatically on code import.)

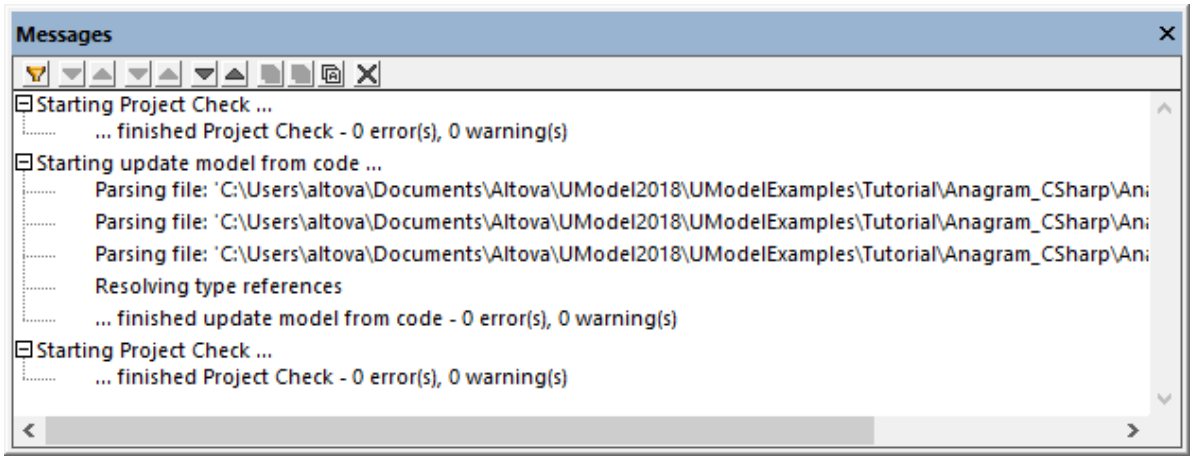
© 2019-2025 Altova GmbH



9. Enter a name and select a destination folder for the new UModel project, and click **Save** (by default, this dialog box displays the same folder as the solution you are importing).

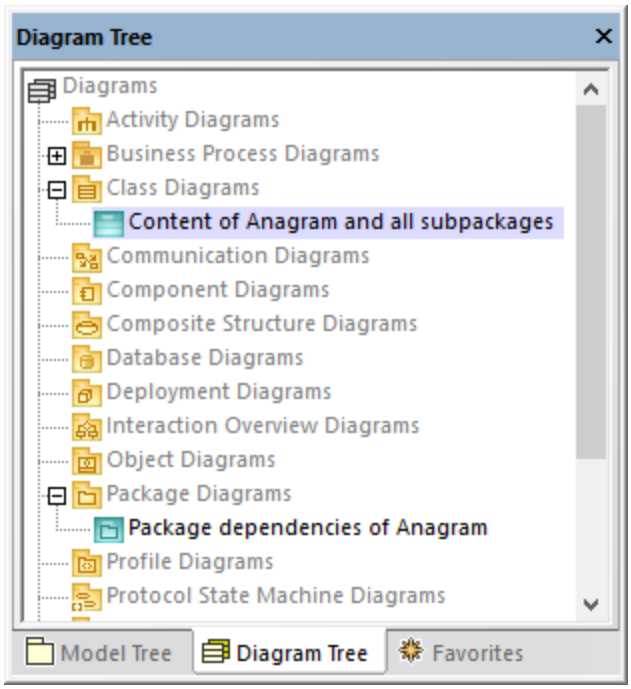


The progress of the reverse engineering operation is shown in the Messages window.

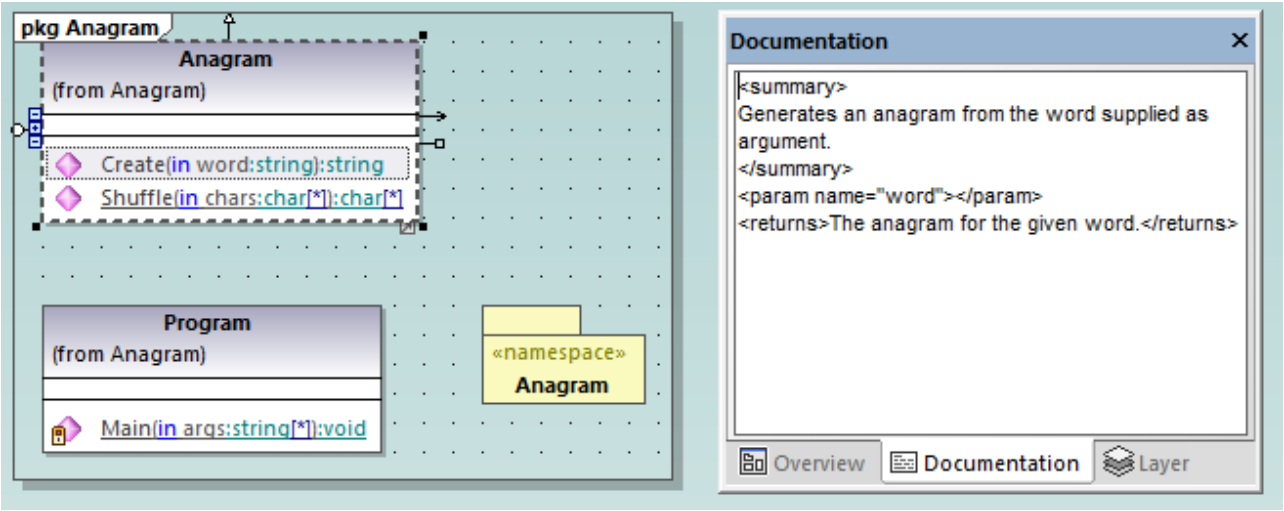


Also, when code import completes, all generated diagrams are opened automatically since this option was selected before code generation. All generated diagrams are available in the Diagram Tree:

© 2019-2025 Altova GmbH



Since we opted to generate documentation from the source code, the imported documentation is visible in the **Documentation** window if you click, for example, the `Create` operation of the `Anagram` class:



Note: The documentation is added only if the option **DocComments as Documentation** was selected while importing the C# solution (see "Step 3: Import the C# Solution" above).

6.4 Importing Java, C# and VB.NET Binaries

UModel supports the import of C# , Java and VB.NET binaries. This is extremely useful when working with binaries from a third party, or if the original source code has become unavailable. Note the following:

- To import Java binary files, a supported version 13 of the Java Runtime Environment (JRE) or Development Kit (JDK) must be installed. Type import is supported for Java .class files or .jar class archives adhering to the Java Virtual Machine Specification. This includes Java Virtual Machines such as OpenJDK, SapMachine, Liberica JDK, and others, see Adding Custom Java Runtimes 205.
- To import C# or VB.NET binary files, .NET Framework, .NET Core, .NET 5, or .NET 6 must be installed, as applicable. For best results, select the **any (use disassembler) option** on the import dialog box. After import, any unrecognized types will be placed in the "Unknown externals" package. To prevent (or decrease the number of) unknown externals, apply the UModel profile specific to the version of your code engineering language (for example, ".NET 5 for C# 9.0") *before the import*. See also Applying UModel Profiles 160.
- The import of obfuscated binaries is not supported.

The table below lists the available approaches for importing binary types into a UModel project.

C#, VB.NET	Java
Import assembly file (.dll, .exe)	Import class file archive (.jar, .zip)
Import assembly from Global Assembly Cache (GAC)	Import class file (.class) from a package root folder
Import assembly from Visual Studio .NET References	Import class archives from class path
	Import class archives from Java runtime (only for Java versions up to and including Java 8)

You can import binary files by running the **Project | Import Binary Types** menu command. Optionally, you can have UModel generate class and package diagrams from the imported types. For examples, see Example: Import .NET GAC Assemblies 209 and Example: Import Java .class Files 211.

In addition, you can import binary files from the command line (see UModel Command Line Interface 101) and programmatically using the UModel API (see Importing Binary Types Programmatically 773).

When importing binary files into a UModel project, you can specify various import options, including:

- You can import any referencing types, in addition to the types defined in the binary file. In addition, you can restrict importing referencing types to specific Java packages and .NET namespaces.
- You can skip type members while importing. For example, you can import classes and interfaces without their properties and methods.
- You can import types according to their accessibility modifiers (such as private or public). For example, you can import only public classes and skip private, protected, and internal classes.

For reference to all options, see Import Binary Options 205.

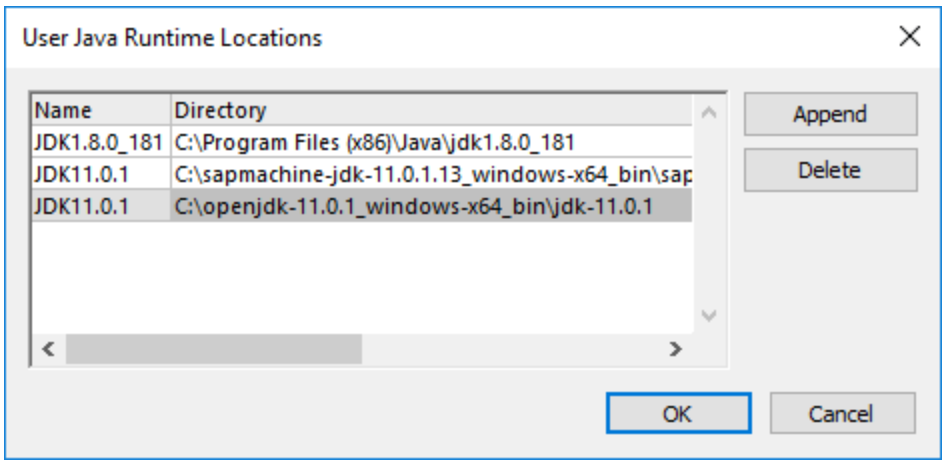
© 2019-2025 Altova GmbH

6.4.1 Adding Custom Java Runtimes

By default, UModel detects JDKs and JREs if they are *installed* on the local machine. Consequently, these appear in the list of Java runtimes when you start the binary import wizard. This is the case for JDKs and JREs released by Oracle, which come with an installer and register themselves in the system when installed. However, other Java Virtual Machine distributions that do not have an installer must be added manually into UModel. The latter include Oracle OpenJDK, SapMachine, and others.

To add custom Java runtimes to UModel:

1. On the **Project** menu, click **Import Binary Types**.
2. Select **Java** as language.
3. Expand the **Runtime** drop-down list, and click **Edit user Java runtime locations**.
4. Click **Append** and browse for the directory of the respective JDK.



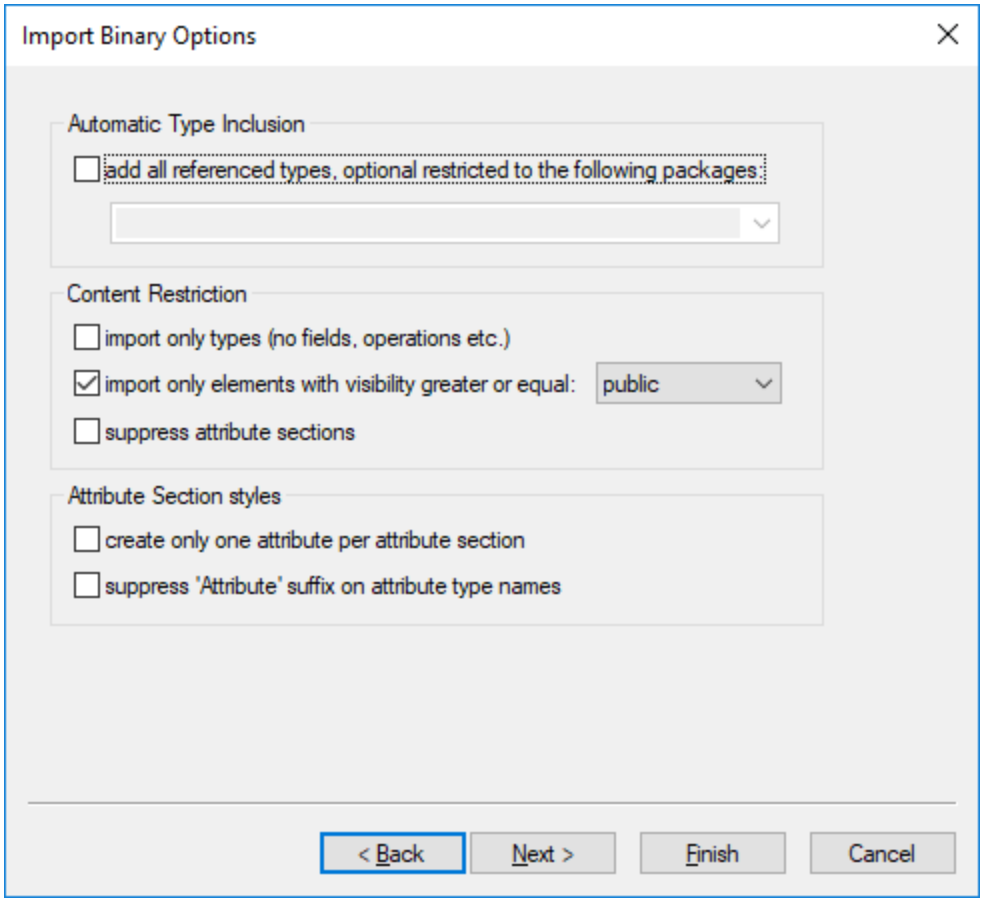
5. Click **OK**.

The selected runtime now appears in the **Runtime** list, and you can select it whenever you need to import binary files targeting that runtime.

Note that these settings affect only the import of binary files. For information about adding a Java Virtual Machine path to be used for JDBC connectivity and Java code generation and import, see Java Virtual Machine Settings 694.

6.4.2 Import Binary Options

When you run the menu command **Project | Import Binary Types**, one of the wizard steps prompts you to specify the binary import options. The options you can set are described below. Note that the dialog box options may be slightly different, depending on whether you are importing .NET or Java binaries.



Import Binary Options dialog box

Automatic type inclusion

.NET or Java binaries may reference various external assemblies or packages. Select the option **add all referenced types...** if you would like to import all types referenced by the types included in the binary file.

To import referenced types only for specific Java packages or .NET namespaces, enter those packages or namespaces in the adjacent text box. To separate multiple packages or namespaces, use the comma, semi-colon, or space characters.

For example, let's assume that the source .NET .dll file references types from `System.Reflection` and `System.Data` namespaces. If you would like to import types from the `System.Reflection` namespace but not from the `System.Data` namespace, select the option **add all referenced types, optionally restricted to the following packages** and enter "System.Reflection" in the text box.

Content restriction

Select the option **import only types** to skip members such as fields, operations, properties, and so on.

Select the option **import only elements with visibility greater than or equal to** to import types and type members according to their visibility. The table below lists visibility of types, beginning with types with least

© 2019-2025 Altova GmbH

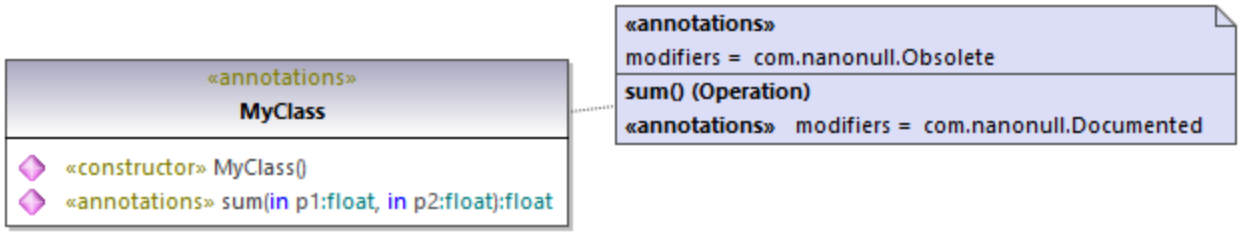
visibility. For example, selecting "private" will import all types, whereas selecting "public" will import only public types and type members.

Note: If the check box is not selected, all types will be imported, regardless of their visibility.

.NET	Java
private	private
internal	package (default visibility when no explicit modifier exists)
protected	protected
public	public

The option **suppress attribute sections** is applicable for .NET binaries. By default, UModel imports the C# or VB.NET attributes detected in the binary. Select the **suppress attribute sections** option if you don't want to import attributes. Otherwise, members that were decorated with attributes in the original source code will have the `<<attributes>>` stereotype applied to them after you import the binary into the model. If attributes are imported, you can display them on the diagram as tagged values, by right-clicking the class on the diagram and selecting **Tagged Values | All** from the context menu. For more information, see Stereotypes and Tagged Values 146.

The option **suppress annotation modifiers** is applicable for Java binaries. By default, UModel imports Java annotations detected in the binary, provided that their retention policy was defined as `RUNTIME` (not `CLASS` or `SOURCE`). If you don't want to import annotations, select the **suppress annotation modifiers** option. If annotations are imported, members that had annotations in the original source have the `<<annotations>>` stereotype, and annotations appear as tagged values, as illustrated below.



Attribute section styles

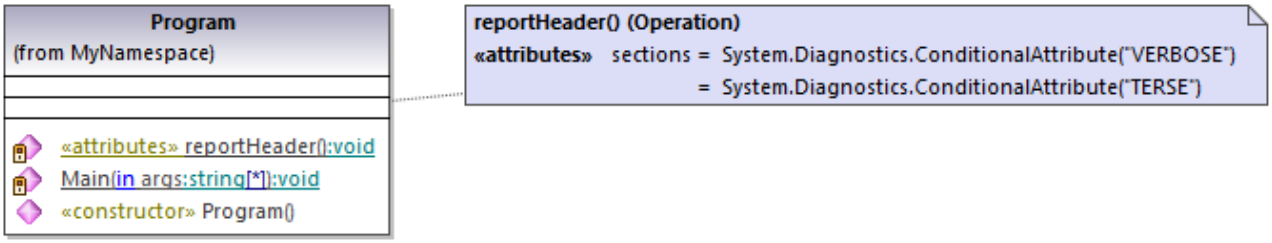
These options are applicable to .NET binaries only. As previously mentioned, if types or type members in the original source code were decorated with attributes, these are imported as tagged values in UModel.

The option **create only one attribute per attribute section** is best illustrated by an example. Let's assume that the original C# source code defined a method with two attributes:

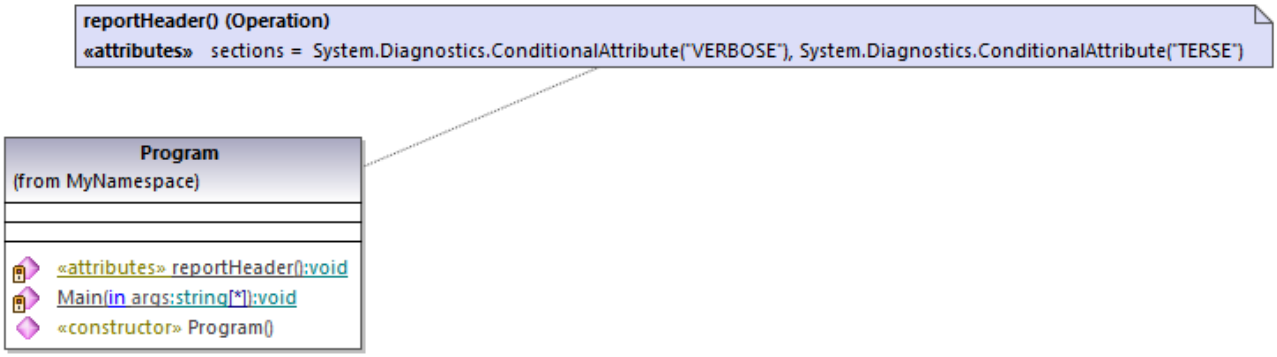
```
using System;
using System.Diagnostics;

namespace MyNamespace
```

```
{
    class Program
    {
        [Conditional("VERBOSE"), Conditional("TERSE")]
        static void reportHeader()
        {
            Console.WriteLine("This is the header");
        }

        static void Main(string[] args)
        {
            reportHeader();
        }
    }
}
```

If the option **create only one attribute per attribute section** is enabled upon importing from the binary file, then each attribute would appear on a separate line inside the "Tagged Values" element :



Otherwise, attributes would appear as comma-separated:



Finally, the option **suppress 'Attribute' suffix on attribute type names** removes the 'Attribute' suffix of an attribute type. For example, if this option is selected, an attribute type defined in the original code as `System.Xml.Serialization.XmlTypeAttribute` would be imported as `System.Xml.Serialization.XmlType`.

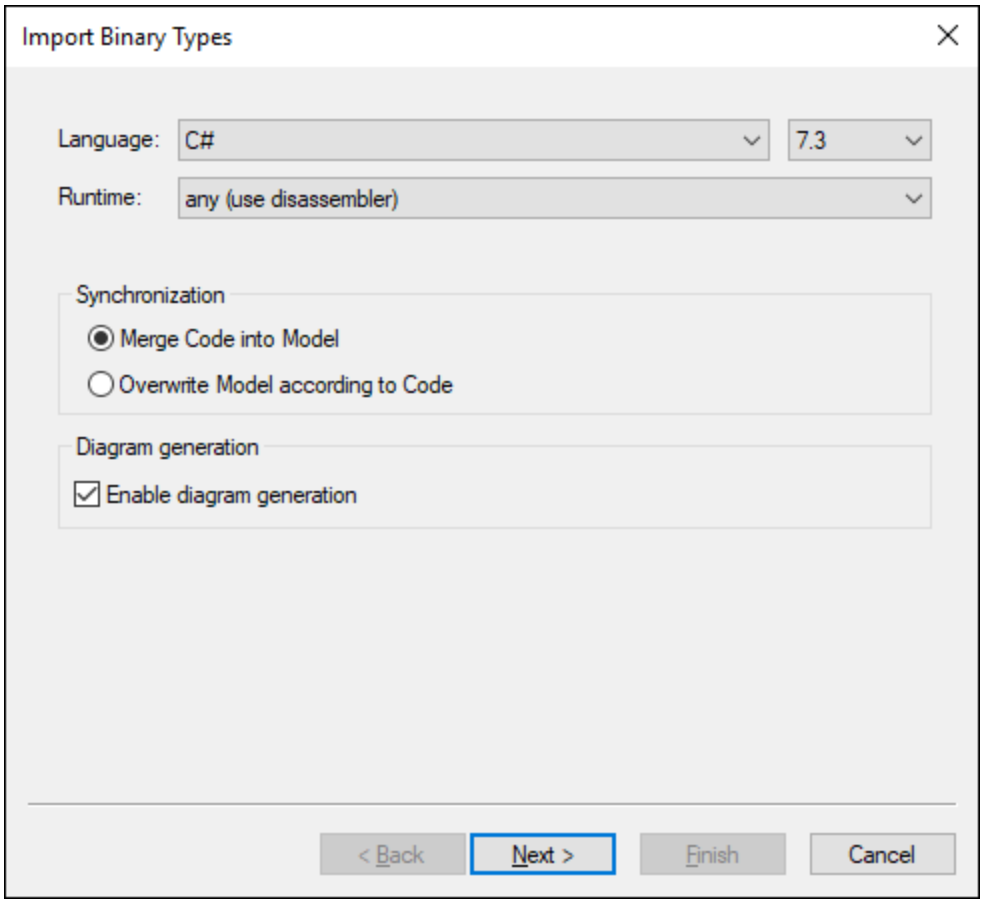
© 2019-2025 Altova GmbH

6.4.3 Example: Import .NET Assemblies

This example shows you how to import binary types from the .NET Global Assembly Cache (GAC) into a UModel C# project. The instructions are similar if you want to import binary types from a standalone `.dll` or `.exe` file. To find out out how to import Java `.class` files, see the next topic 211.

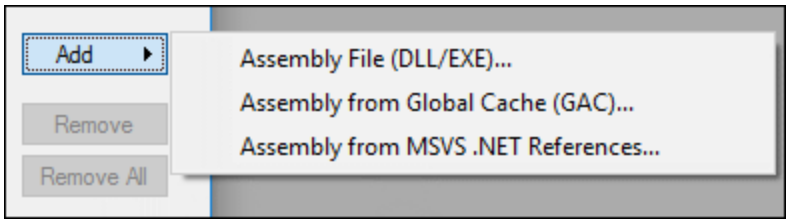
To import binary files from the .NET Global Assembly Cache:

1. Go the **Project** menu and click **Import Binary Types** (*see screenshot below*).

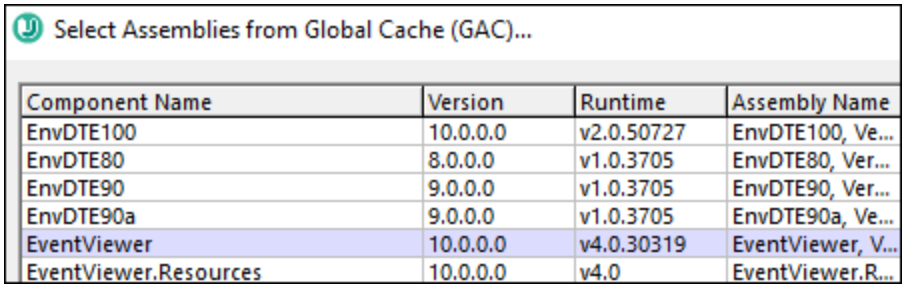


2. Choose the target language of the UModel project (C#, VB.NET, Java). In this example, C# is selected, since we are importing a .NET GAC assembly.
3. If you would like to set a specific language version for the imported UModel project, select it from the adjacent text box. In this example, C# 7.3 is selected.
4. Optionally, select a .NET runtime version from the **Runtime** drop-down list. The default option is *any (use disassembler)*. In this case, UModel will choose a reflection API that is most appropriate for the imported binary.
5. If you import binary types into a new project, select either **Merge Code into Model** or **Overwrite Model according to Code**.
6. Optionally, to generate class diagrams and package diagrams from the imported binary types, select the **Enable diagram generation** check box. If you select this option, more diagram generation options will be available in the next steps. See Generating Class Diagrams 399 and Generating Package Diagrams 408.
7. Click **Next**.

8. Click **Add | Assembly from Global Cache (GAC)** (*see screenshot below*). Note that the option **Assembly from Global Cache (GAC)** is only available for .NET Framework 2.x-4.x. The GAC is not relevant to .NET Core, .NET 5 and later versions. For more information, see the Microsoft documentation. In order to import assembly files for .NET Core, .NET 5 and .NET 6, you will need to extract the required files from the GAC. Then click **Add | Assembly File (DLL/EXE)**, select the assembly files manually and add them to the project.



9. Select an assembly from the dialog box. In this example, the *EventViewer* assembly is selected (*see screenshot below*).



10. Select the types you would like to import and click **Next**. For more information about other options of the **Import Binary Selection** dialog box, see the notes below.
11. Select the import options as applicable. For more information, see Import Binary Options 205.
12. If you enabled diagram generation in Step 6, click **Next** and configure the options applicable to diagram generation. Otherwise, click **Finish**.

UModel performs the conversion and displays a progress log in the **Messages** window. If the conversion of binary types is not possible, the error text may provide additional information. For example, the binary file you are trying to import is targeting a runtime newer than the one selected in the **Import Binary Types** dialog box. In this case, select a newer runtime version and try again.

Notes:

- The text box **Override of PATH variable...** in the **Import Binary Selection** dialog box is applicable only to Java. Optionally, paste here any Java class paths that must be queried in addition to those read from the `CLASSPATH` environment variable. Alternatively, click **Add** and browse for the required folders.
- The check box **use 'reflection only' context...** in the **Import Binary Selection** dialog box is applicable only when you import a C# or VB.NET binary. This is useful when importing a library which has dependencies that cannot be resolved or loaded. Selecting this check box will not execute any static initializer code, which might cause errors when importing.

© 2019-2025 Altova GmbH

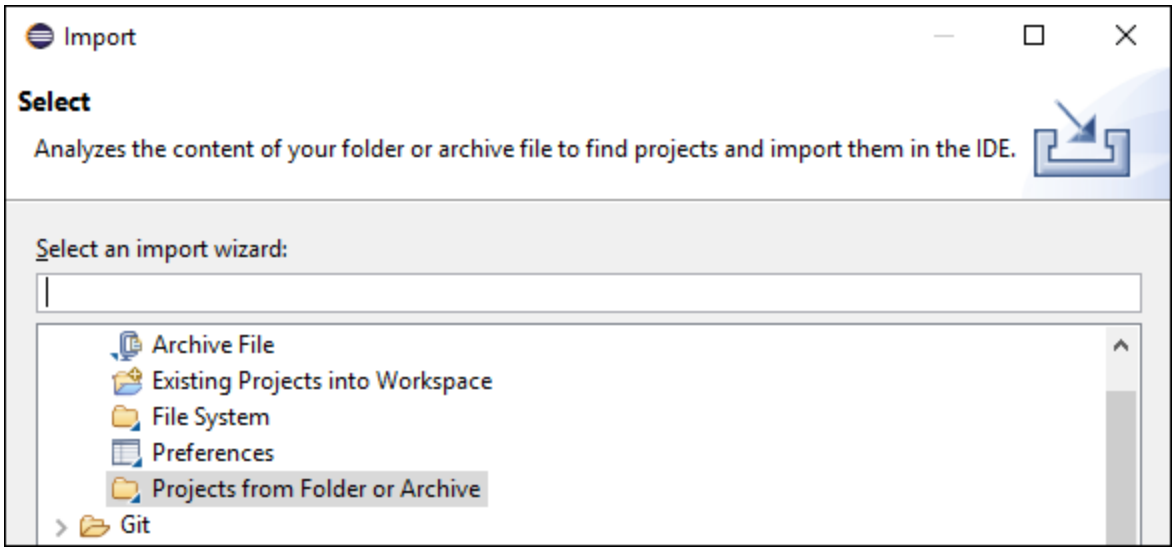
6.4.4 Example: Import Java .class Files

This example shows you how to import compiled Java .class files into UModel. In this example, the source Java .class files originate from a tutorial Java project that was created with UModel, but you can also use other .class files as an alternative.

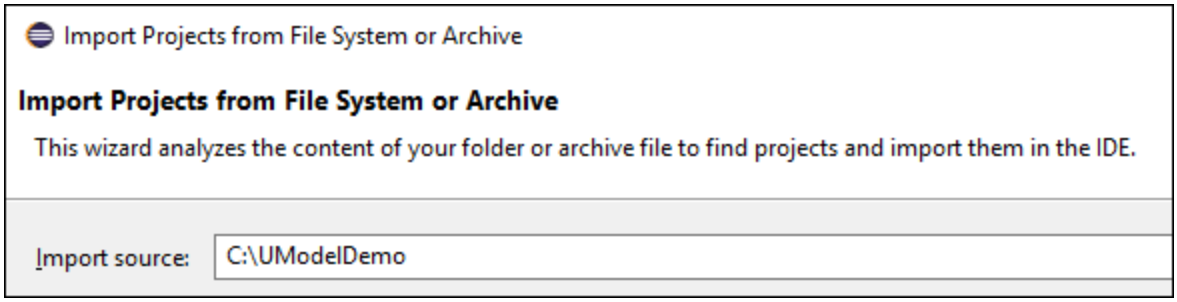
Compiling UModel-generated Java code (optional)

This section shows you how to compile a demo UModel-generated Java project with Eclipse. Note that this step is purely optional, the goal here is to obtain some compiled .class files. You can skip it if you already have readily available Java .class files. In this example, Eclipse is chosen as compilation environment for convenience; however, you can use the Java command line or some other Java development environment to achieve the same result.

1. If you haven't done that already, create a simple Java project with UModel, as shown in Example: Generate Java Code 182. This is a very simple example consisting of a Java package with only one class. When you complete the example, the directory **C:\UModelDemo\src** will contain the required Java source code.
2. Run Eclipse. On the **File** menu, click **Import**.

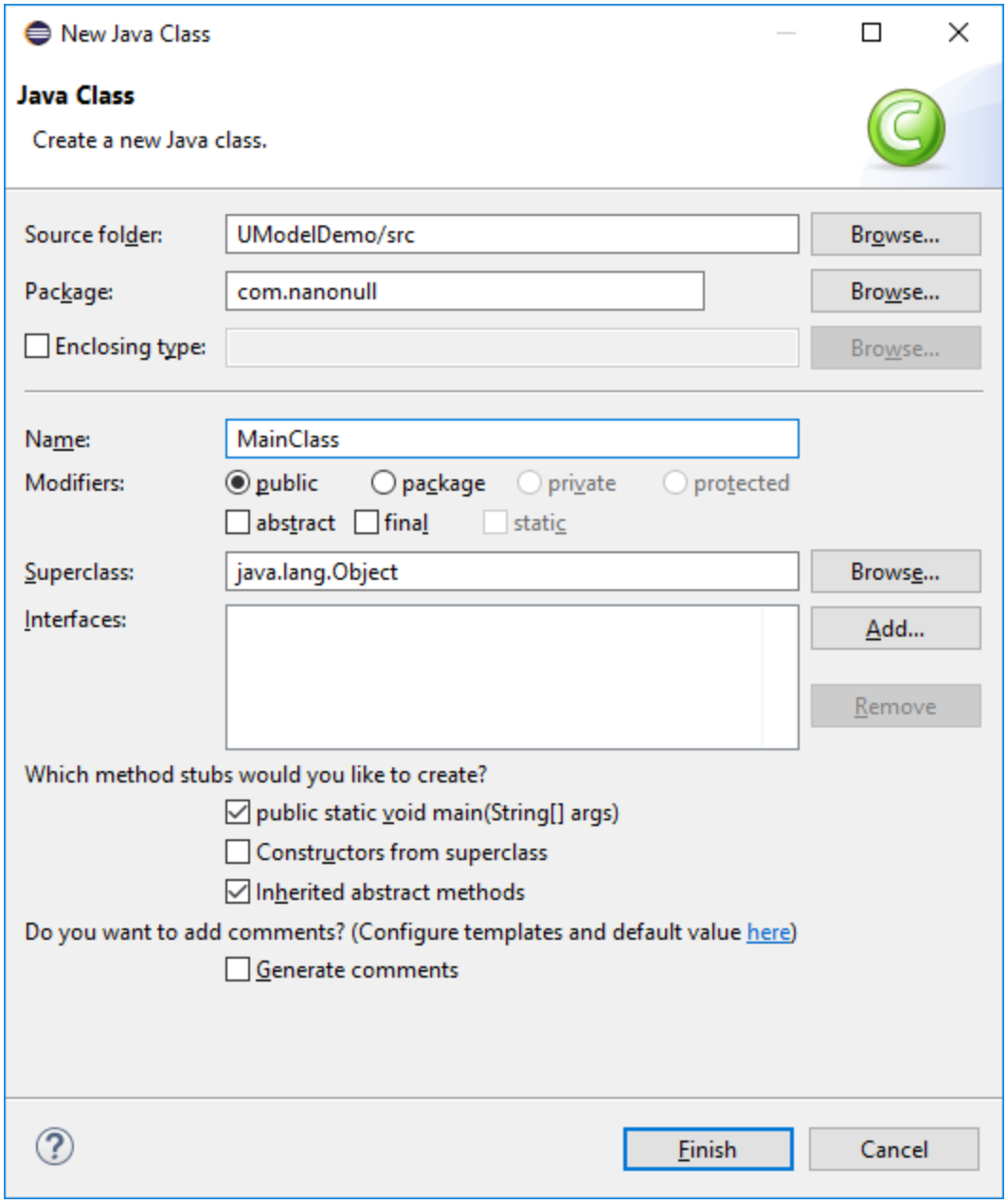


3. Select **Projects from Folder or Archive**, and click **Next**.



4. Enter **C:\UModelDemo** as directory, and click **Finish**.

5. Right-click the **com.nanonull** package in Eclipse's Package Explorer and select **New | Class** from the context menu.
6. Enter a class name ("MainClass", in this example), and select the **public static void main...** check box.



7. On the **Run** menu, click **Run**.

You have now finished compiling the UModel-generated Java project. The compiled .class files should now be available in the **bin** sub-directory of your project's directory.

Finally, take note of the Java version used for compilation—this is important if you intend to import binary types later. By default, if you did not modify your Eclipse project properties, it is likely that it was compiled with the default Java version available to Eclipse. To view the default Java version, do the following in Eclipse:

© 2019-2025 Altova GmbH

1. On the **Window** menu, click **Preferences**.
2. Click **Java**, and then click **Installed JREs**.

Importing Java .class files

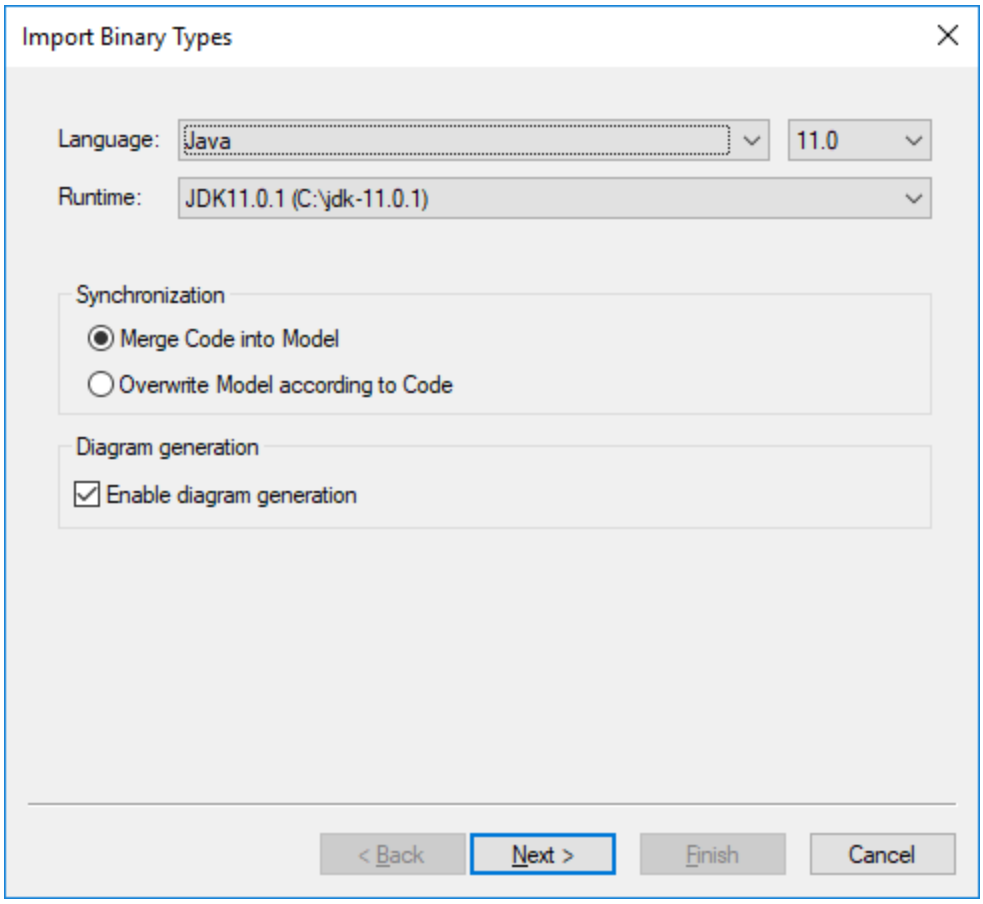
If you already have binary .class files such as the ones compiled previously, you can now proceed to importing them into UModel.

1. Create a new UModel project, or open an existing one. In this example, we are importing binary types into a new project.
2. If your project does not contain the Java JDK types already, do the following:

    a. On the **Project** menu, click **Include subproject**.
    b. Click the **Java** tab and select **Java JDK (types only)**.
    c. Select **Include by reference** when prompted.

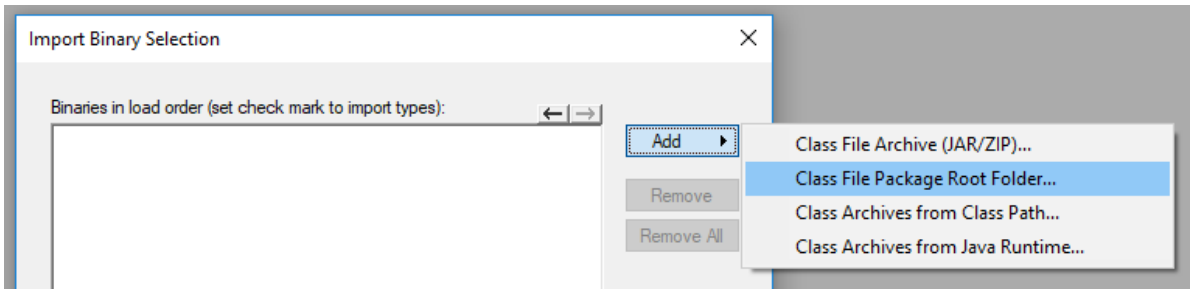
Note: This is an optional step which normally prevents the "Unknown externals" package from appearing in the project after the import is complete.

3. On the **Project** menu, click **Import Binary Types**.
4. Select **Java** as language, and the Java version in which the Java code was compiled (for example, 11.0).
5. Select the Java runtime to be used by UModel for extracting information from the binary files (the so-called "reflection"). The runtime version must be equal or newer than the Java version selected in the previous step.



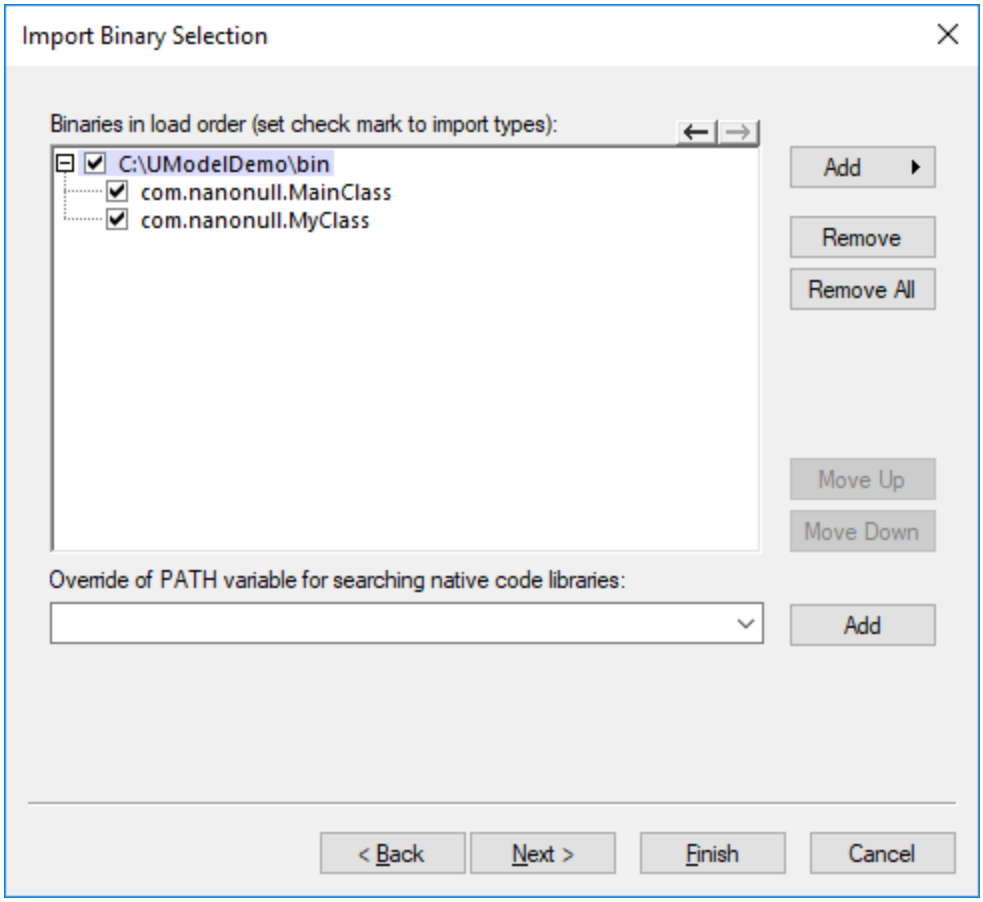
Note: The **Runtime** drop-down list contains only Java JDKs and JREs detected automatically. If your JDK or JRE is not listed, select the entry **Edit user java runtime locations** and browse for the directory where the respective distribution is installed on your machine, see Adding Custom Java Runtimes 205.

6. If you import binary types into a new project, select either **Merge Code into Model** or **Overwrite Model according to Code**. Otherwise, select **Merge code into Model**.
7. Optionally, to generate class diagrams and package diagrams from the imported binary types, select the **Enable diagram generation** check box. If you select this option, more diagram generation options are available in subsequent steps, see also Generating Class Diagrams 399 and Generating Package Diagrams 408.
8. Click **Next**.

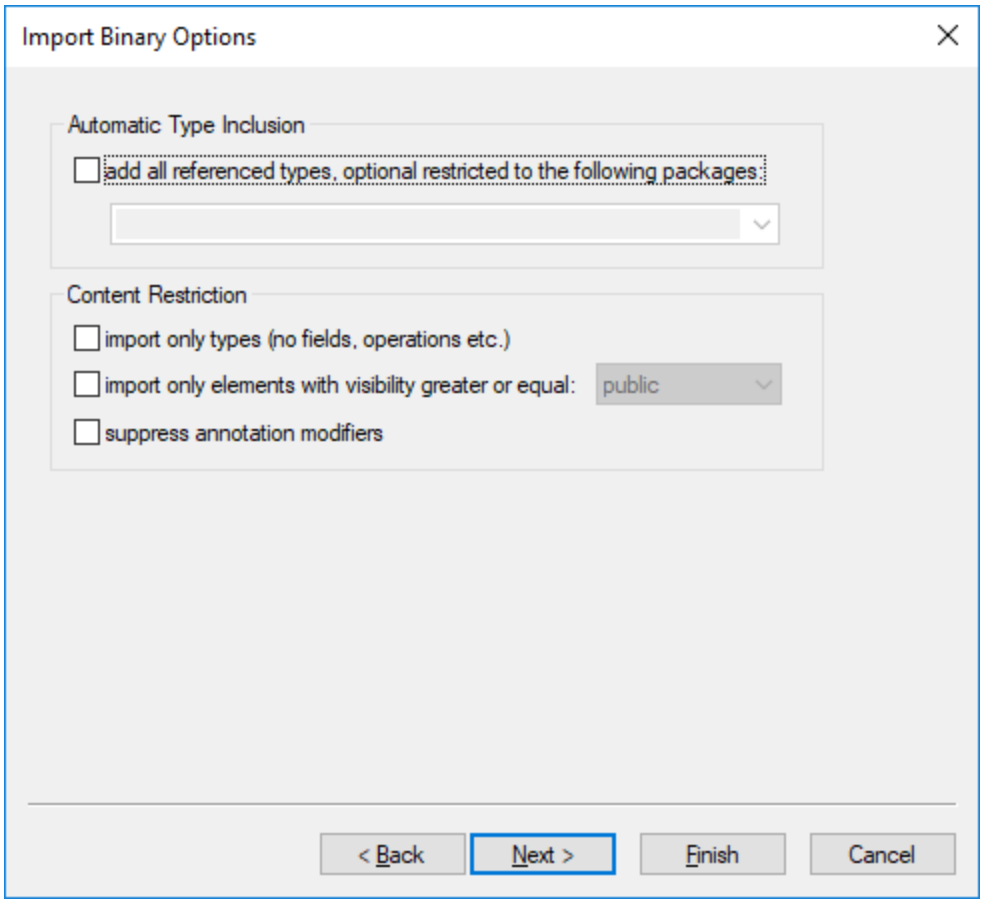


 © 2019-2025 Altova GmbH

9. In this example, we are importing Java .class files from a package root. Select **Add | Class File Package Root Folder**. and browse for the **C:\UModelDemo\bin** directory. If this directory does not exist, make sure to compile the project first, as shown in the first part of this tutorial.



10. Select the classes to be imported, and click **Next**.



11. Select the import options as applicable, see Import Binary Options [205].
12. If you enabled diagram generation in an earlier step, click **Next** and configure the options applicable to diagram generation. Otherwise, click **Finish**.

UModel performs the conversion and displays a progress log in the **Messages** window. If the conversion of binary types is not possible, the error text may provide additional information. For example, the binary file you are trying to import is targeting a runtime newer than the one selected in the **Import Binary Types** dialog box. In this case, select a newer runtime version and try again.

© 2019-2025 Altova GmbH

6.5 Synchronizing the Model and Source Code

You can synchronize the model and code in either direction, and at different levels (for example, project, package or class).

When UModel (Enterprise or Professional) runs as an Eclipse or Visual Studio plug-in, synchronization between model and code takes place automatically. Manual synchronization is possible at the project level; the option to update individual classes or packages is not available. For more information, see UModel Plug-in for Visual Studio 570 and UModel Plug-in for Eclipse 581.

When you right-click an element in the Model Tree (for example, a class), the context menu displays the code synchronization or merging commands under the **Code Engineering** menu item:

- **Merge Program Code from UModel *****
- **Merge UModel ***** from Program Code**

*** is a Project, Package, Component, Class, and so on, depending on your current selection.

Depending on the settings you have defined from **Project | Synchronization Settings**, the alternative name of these two commands may be:

- **Overwrite Program Code from UModel *****
- **Overwrite UModel ***** from Program Code**

To update the entire project (but not classes, packages, or other local elements), you can also use the following commands on the **Project** menu of UModel:

- **Merge (or Overwrite) Program Code from UModel Project**
- **Merge (or Overwrite) UModel Project from Program Code**

For convenience, any of the commands listed above will be generically referred to as "code synchronization commands" further in this topic.

To synchronize at the project or Root package level, do one of the following:

- Right-click the Root package in the Model Tree, and select the required code synchronization command.
- On the **Project** menu, click the required code synchronization command.

To synchronize at package level:

1. Use **Shift**, or **Ctrl + Click** to select the package(s) you want to merge.
2. Right-click the selection, and select the required code synchronization command.

To synchronize at class level:

1. Use **Shift**, or **Ctrl + Click** to select the classes(s) you want to merge.
2. Right-click the selection, and click the required code synchronization command.

To avoid undesired results when synchronizing the model and code, consider the following scenarios:

On the **Project** menu, click **Overwrite UModel Project from Program Code**.	• This checks all directories (project files) of all different code languages you have defined in your project. • New files are identified and added to the project. • An entry "Collecting source files in (...)" appears in the Messages window.
Right-click a class or interface in the Model Tree and select **Code Engineering | Overwrite UModel Class from Program Code**.	• This updates only the selected class (interface) of your project. • If the source code contains classes that are new or modified classes since the last synchronization, those changes will not be added to the model.
Right-click a Component in the Model Tree (within the Component View package) and select **Code Engineering | Overwrite UModel Component from Program Code**.	• This updates the corresponding directory (or project file) only. • New files in the directory (project file) are identified and added to the project. • An entry "Collecting source files in (...)" appears in the Message window.

Note: When synchronizing code, you might be prompted to update your UModel project before synchronization. This occurs when you open UModel projects created before the latest release. Click **Yes** to update your project to the latest release format, and save your project file. The notification message will not occur once this has been done.

6.5.1 Synchronization Tips

Renaming of classifiers and reverse engineering

The process described below applies to the standalone application as well as to the plug-in versions (Visual Studio or Eclipse) when reverse engineering or automatic synchronization takes place.

Renaming a classifier in the code window of your programming application causes it to be deleted and re-inserted as new classifier in the **Model Tree**.

The new classifier is only re-inserted in those modeling diagrams that are automatically created during the reverse-engineering process, or when generating a diagram using the **Show in new Diagram | Content** option. The new classifier is inserted at a default position on the diagram, that will likely differ from the previous location.

See also Refactoring code and synchronization 220.

 © 2019-2025 Altova GmbH

Automatic generation of ComponentRealizations

UModel is capable of automatically generating ComponentRealizations during the code engineering process. ComponentRealizations are only generated where it is absolutely clear to which component a class should be assigned:

- Only one Visual Studio project file exists in the .ump project.
- Multiple Visual Studio projects exist but their classes are completely separate in the model.

To enable automatic generation of ComponentRealizations:

1. Open the menu item **Tool | Options**.
2. Click the **Code Engineering** tab and activate the **Generate missing ComponentRealizations** option.

Automatic ComponentRealizations are created for a **Classifier** that can be assigned one (and only one) Component

- without any ComponentRealizations, or
- contained in a code language namespace.

The way the Component is found differs for the two cases.

Component representing a code project file (property "**projectfile**" set)

- if there is ONE Component having/realizing classifiers in the containing package
- if there is ONE Component having/realizing classifiers in a subpackage of the containing package (top down)
- if there is ONE Component having/realizing classifiers in one of the parent packages (bottom up)
- if there is ONE Component having/realizing classifiers in a subpackage of one of the parent packages (top down)

Component representing a directory (property "**directory**" set)

- if there is ONE Component having/realizing classifiers in the containing package
- if there is ONE Component having/realizing classifiers in one of the parent packages (bottom up)

Notes:

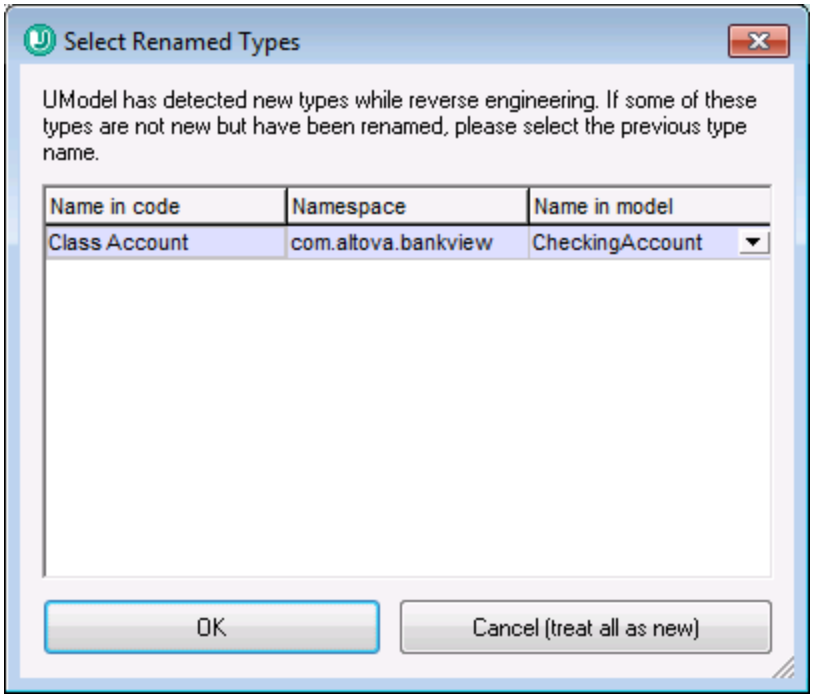
- The option "**Code Engineering | Generate missing ComponentRealizations**" has to be set.
- As soon as ONE viable Component is found during one of the above steps, this Component is used and the remaining steps are ignored.

Error/Warnings:

- If no viable Component was found, a warning is generated (message log)
- If more than one viable Component was found, an error is generated (message log)

6.5.2 Refactoring Code and Synchronization

When refactoring code, it is often the case that class names are changed or updated in the code. If it detects that new types have been added or renamed during reverse engineering, UModel (version 2009 or later) displays a dialog box. The new types are listed in the "Name in code" column while the assumed original type name is listed in the "Name in model" column. UModel attempts to determine the original name by relying on namespace, class content, base classes and other data.



If a class was renamed, select the previous class name using the combo box in the "Name in model" column, e.g. C1. This ensures that all related data are retained and the code engineering process remains accurate.

Changing class names in the model and regenerating code

Having created a model and generated code from it, it is possible that you might want to make changes to the model again before going through the synchronization process.

E.g. You decide that you want to change the class names before generating code the second time round. As you previously assigned a file name to each class, in the "code file name" field of the Properties window, the new class and file name would now be out of sync.

UModel prompts if you want the code file name to agree with the new class name, when you start the synchronization process. Note that you also have the option to change the class constructors as well.

Round-trip engineering and relationships between modeling elements

When updating model from code, associations between modeling elements are automatically displayed, if the option **Diagram Editing | Automatically create Associations** has been activated in the **Tools | Options**

© 2019-2025 Altova GmbH

dialog box. Associations are displayed for those elements where the attributes type is set, and the referenced "type" modeling element is in the same diagram.

`InterfaceRealizations` as well as `Generalizations` are all automatically shown in the diagram when updating model from code.

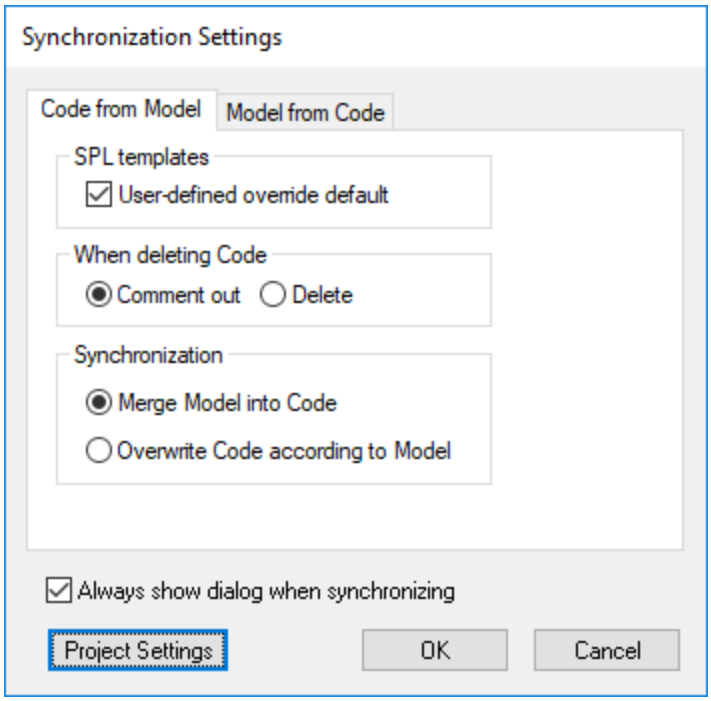
6.5.3 Code Synchronization Settings

The code synchronization settings are relevant in the following scenarios:

- When program code is generated from the model (that is, when either the command **Project | Merge Program Code from UModel Project** or the command **Project | Overwrite Program code from UModel Project** is run)
- When source code is imported into the model (that is, when either the command **Project | Merge UModel Project from Program Code** or the command **Project | Overwrite UModel Project from Program Code** is run)
- When automatic synchronization takes place in either direction (this applies to UModel Enterprise and Professional Editions when UModel runs as a Visual Studio or Eclipse plug-in).

To change the code synchronization settings:

- On the **Project** menu, click **Synchronization Settings**.



Synchronization Settings dialog box

By default, the Synchronization Settings dialog box will be displayed automatically every time when you initiate any of the code synchronization commands. To disable this behaviour, clear the check box **Always show dialog when synchronizing**.

The available options are grouped into two tabs:

- **Code from Model** (options in this tab are applicable when program code is generated from the model)
- **Model from Code** (options in this tab are applicable when program code is imported into the model).

Option	Description
SPL templates	This option is applicable only when generating program code. Select the check box **User-defined override default** check box if you have created custom Spy Programming Language (SPL) templates that should override the default ones supplied with UModel (see also SPL Templates 190).
When deleting code	This option is applicable only when generating program code. Select whether program code should be deleted or commented out during synchronization (assuming the relevant objects no longer exist in the model).
Synchronization	This option is applicable both when generating and importing program code. It lets you specify whether changes should be merged as opposed to being overwritten. Assuming that code has been generated once from a model, and changes have since been made to both model and code, for example: • A new class X has been added in UModel • A new class Y has been added to the external code, **Merge Model into Code** means that: • The newly added class Y in the external code is retained • The newly added class X, from UModel, is added to the code. **Overwrite Code according to Model** means that: • The newly added class Y in the external code is deleted (or commented out, depending on the current settings) • The newly added class X, from UModel, is added to the code. **Merge Code into Model** means that: • The newly added class X in UModel is retained • The newly added class Y, from the external code, is added to the model **Overwrite Model according to Code** means that: • The newly added class X in UModel is deleted (or commented out, depending on the current settings) • The newly added class Y, from the external code, is added to the model.
Project settings	Opens the Project Settings dialog box, where you can modify the code engineering settings applicable to each language. For reference to all settings, see Code Import Options 194 and Code Generation Options 175, respectively.

© 2019-2025 Altova GmbH

Option	Description
	The Project Settings dialog box can also be triggered from the menu command **Project | Project Settings**. Note that the project settings in this dialog box are global (they are saved together with the project and are applicable on any workstation where the UModel project is open) whereas the options you define from **Tools | Options** are local (they are applicable only to the current installation of UModel).

6.6 UModel Element Mappings

This section illustrates how UModel elements map to elements (constructs) in various programming languages (C#, Java, VB.NET), as well as to databases and XML schemas. The mappings are grouped by language, and are applicable when importing code into model, or when generating code from model.

- C# Mappings 224
- VB.NET Mappings 244
- Java Mappings 258
- XML Schema Mappings 264
- Database Mappings 273

6.6.1 C# Mappings

The table below shows the one-to-one correspondence between:

- UModel elements and C# code elements, when outputting model to code
- C# code elements and UModel model elements, when inputting code into model

C# Project

C#		UModel	
Project	projectfile	projectfile	Component
	directory	directory	

C# Namespace

C#		UModel	
Namespace	name	name	Package <<namespace>>

C# Class

C#			UModel		
Class	name		name		Class
	modifiers	internal	visibility	package	
		protected internal		protected <<internal>>	
		public		public	
		protected		protected	
		private		private	
		sealed	leaf		

© 2019-2025 Altova GmbH

C#				UModel			
		abstract		abstract			
		static		<<static>>			
		unsafe		<<unsafe>>			
		partial		<<partial>>			
		new		<<new >>			
	filename			code file name			
	associated projectfile/directory			ComponentRealization			
	base types			Generalization, InterfaceRealization(s)			
	attribute sections			<<attributes>>			
	doc comments			Comment(->Documentation)			
	Field	name		name		Property	
		modifiers	internal	visibility	package		
			protected internal		protected <<internal>>		
			public		public		
			protected		protected		
			private		private		
			static	static			
			readonly	readonly			
			volatile	<<volatile>>			
			unsafe	<<unsafe>>			
			new	<<new >>			
		type		type			
		type dimensions		multiplicity			
		type pointer		type modifier			
		nullable		<<nullable>>			
		default value		default			
		attribute sections		<<attributes>>			
		doc comments		Comment(->Documentation)			
	Constant	name		name		Property <<const>>	
		modifiers	internal	visibility	package		

C#				UModel			
			protected internal		protected <<internal>>		
			public		public		
			protected		protected		
			private		private		
			new	<<new >>			
		type		type			
		type dimensions		multiplicity			
		type pointer		type modifier			
		nullable		<<nullable>>			
		default value		default			
		attribute sections		<<attributes>>			
		doc comments		Comment(->Documentation)			
	Method	name		name			Operation
		modifiers	internal	visibility	package		
			protected internal		protected <<internal>>		
			public		public		
			protected		protected		
			private		private		
			static	static			
			abstract	abstract			
			sealed	leaf			
			override	<<override>>			
			partial	<<partial>>			
			virtual	<<virtual>>			
			new	<<new >>			
			unsafe	<<unsafe>>			
		attribute sections		<<attributes>>			
		doc comments		Comment(->Documentation)			
		implemented interfaces		implements			
		type		direction	return	Parameter	

© 2019-2025 Altova GmbH

C#					UModel			
		Parameter	name		name			
			modifiers	ref	direction	inout		
				out		out		
				params	varArgList			
			type		type			
			type dimensions		multiplicity			
			type pointer		type modifier			
			this		<<this>>			
			nullable		<<nullable>>			
		Type Parameter	name		name		Template Parameter	
			constraint		constraining classifier			
			predefined constraint	struct	<<ValueTypeConstraint>>			
				class	<<ReferenceTypeConstraint>>			
				new ()	<<ConstructorConstraint>>			
			attribute sections		<<attributes>>			
	Constructor	name			name			Operation <<constructor>>
		modifiers	internal		visibility	package		
			protected internal			protected <<internal>>		
			public			public		
			protected			protected		
			private			private		
			static		static			
			unsafe		<<unsafe>>			
		attribute sections			<<attributes>>			
		doc comments			Comment(->Documentation)			
		Parameter	name		name		Parameter	
			modifiers	ref	direction	inout		
				out		out		

C#					UModel				
				params	varArgList				
			type		type				
			type dimensions		multiplicity				
			type pointer		type modifier				
			nullable		<<nullable>>				
	Destructor	name			name			Operation <<destructor>>	
		modifiers	private		visibility	private			
			unsafe		<<unsafe>>				
		attribute sections			<<attributes>>				
		doc comments			Comment(->Documentation)				
	Property	name			name			Operation <<property>>	
		modifiers	internal		visibility	package			
			protected internal			protected <<internal>>			
			public			public			
			protected			protected			
			private			private			
			static		static				
			abstract		abstract				
			sealed		leaf				
			override		<<override>>				
			virtual		<<virtual>>				
			new		<<new >>				
			unsafe		<<unsafe>>				
		attribute sections			<<attributes>>				
		doc comments			Comment(->Documentation)				
		type			direction	return	Parameter		
		type dimensions			multiplicity				
		nullable			<<nullable>>				
		Get	modifiers	internal	visibility	internal	<<GetAcc		

© 2019-2025 Altova GmbH

C#					UModel				
				protected internal		protected internal			
				protected		protected			
				private		private			
		Set Accessor	modifiers	internal	visibility	internal	<<SetAccessor>>		
				protected internal		protected internal			
				protected		protected			
				private		private			
	Operator	name			name			Operation <<operator>>	
		modifiers	public		visibility	public			
			static		static				
			unsafe		<<unsafe>>				
		attribute sections			<<attributes>>				
		doc comments			Comment(->Documentation)				
		type			direction	return	Parameter		
		Parameter	name		name				
			modifier	params	varArgList				
			type		type				
			type dimensions		multiplicity				
			type pointer		type modifier				
			nullable		<<nullable>>				
	Indexer	name (="this")			name (="this")			Operation <<indexer>>	
		modifiers	internal		visibility	package			
			protected internal			protected <<internal>>			
			public			public			
			protected			protected			
			private			private			
			static		static				
			abstract		abstract				
			sealed		leaf				

C#					UModel				
			override		<<override>>				
			virtual		<<virtual>>				
			new		<<new >>				
			unsafe		<<unsafe>>				
		attribute sections			<<attributes>>				
		doc comments			Comment(->Documentation)				
		type			direction	return	Parameter		
		Parameter	name		name				
			modifier	params	varArgList				
			type		type				
			type dimensions		multiplicity				
			type pointer		type modifier				
			nullable		<<nullable>>				
		Get Accessor	modifiers	internal	visibility	internal	<<GetAccessor>>		
				protected internal		protected internal			
				protected		protected			
				private		private			
		Set Accessor	modifiers	internal	visibility	internal	<<SetAccessor>>		
				protected internal		protected internal			
				protected		protected			
				private		private			
	Event	name			name			Operation <<event>>	
		modifiers	internal		visibility	package			
			protected internal			protected <<internal>>			
			public			public			
			protected			protected			
			private			private			
			static		static				
			abstract		abstract				

© 2019-2025 Altova GmbH

C#				UModel				
			sealed	leaf				
			override	<<override>>				
			virtual	<<virtual>>				
			new	<<new >>				
			unsafe	<<unsafe>>				
		attribute sections		<<attributes>>				
		doc comments		Comment(->Documentation)				
		type		direction	return	Parameter		
		type dimensions		multiplicity				
		nullable		<<nullable>>				
		Add Accessor		<<AddRemoveAccessor>>				
		Remove Accessor						
	Type Parameter	name		name			Template Parameter	
		constraint		constraining classifier				
		predefined constraint	struct	<<ValueTypeConstraint>>				
			class	<<ReferenceTypeConstraint>>				
			new ()	<<ConstructorConstraint>>				
		attribute sections		<<attributes>>				

C# Struct

C#			UModel		
Struct	name		name		Class <<struct>>
	modifiers	internal	visibility	package	
		protected internal		protected <<internal>>	
		public		public	
		protected		protected	
		private		private	
		unsafe	<<unsafe>>		
		partial	<<partial>>		
		new	<<new >>		

C#				UModel			
filename				code file name			
associated projectfile/directory				ComponentRealization			
base types				InterfaceRealization(s)			
attribute sections				<<attributes>>			
doc comments				Comment(->Documentation)			
Field	name			name		Property	
	modifiers	internal		visibility	package		
		protected internal			protected <<internal>>		
		public			public		
		protected			protected		
		private			private		
		static		static			
		readonly		readonly			
		volatile		<<volatile>>			
		unsafe		<<unsafe>>			
		new		<<new >>			
	type			type			
	type dimensions			multiplicity			
	type pointer			type modifier			
	nullable			<<nullable>>			
	default value			default			
	attribute sections			<<attributes>>			
	doc comments			Comment(->Documentation)			
Constant	name			name		Property <<const>>	
	modifiers	internal		visibility	package		
		protected internal			protected <<internal>>		
		public			public		
		protected			protected		
		private			private		
		new		<<new >>			

© 2019-2025 Altova GmbH

C#				UModel			
		type		type			
		type dimensions		multiplicity			
		type pointer		type modifier			
		nullable		<<nullable>>			
		default value		default			
		attribute sections		<<attributes>>			
		doc comments		Comment(->Documentation)			
	Fixedsize Buffer	name		name		Property <<fixed>>	
		modifiers	internal	visibility	package		
			protected internal		protected <<internal>>		
			public		public		
			protected		protected		
			private		private		
			unsafe	<<unsafe>>			
			new	<<new >>			
		type		type			
		type pointer		type modifier			
		nullable		<<nullable>>			
		buffer size		default			
		attribute sections		<<attributes>>			
		doc comments		Comment(->Documentation)			
	Method	name		name		Operation	
		modifiers	internal	visibility	package		
			protected internal		protected <<internal>>		
			public		public		
			protected		protected		
			private		private		
			static	static			
			abstract	abstract			
			sealed	leaf			

C#					UModel				
			override		<<override>>				
			partial		<<partial>>				
			virtual		<<virtual>>				
			new		<<new >>				
			unsafe		<<unsafe>>				
		attribute sections			<<attributes>>				
		doc comments			Comment(->Documentation)				
		implemented interfaces			implements				
		type			direction	return	Parameter		
		Parameter	name		name				
			modifiers	ref	direction	inout			
				out		out			
				params	varArgList				
			type		type				
			type dimensions		multiplicity				
			type pointer		type modifier				
			this		<<this>>				
			nullable		<<nullable>>				
		Type Parameter	name		name		Template Parameter		
			constraint		constraining classifier				
			predefined constraint	struct	<<ValueTypeConstraint>>				
				class	<<ReferenceTypeConstraint>>				
				new ()	<<ConstructorConstraint>>				
			attribute sections		<<attributes>>				
	Constructor	name			name			Operation <<constructor>>	
		modifiers	internal		visibility	package			
			protected internal			protected <<internal>>			
			public			public			

© 2019-2025 Altova GmbH

C#					UModel			
			protected			protected		
			private			private		
			static		static			
			unsafe		<<unsafe>>			
		attribute sections			<<attributes>>			
		doc comments			Comment(->Documentation)			
		Parameter	name		name		Parameter	
			modifiers	ref	direction	inout		
				out		out		
				params	varArgList			
			type		type			
			type dimensions		multiplicity			
			type pointer		type modifier			
			nullable		<<nullable>>			
	Destructor	name			name		Operation <<destructor>>	
		modifiers	private		visibility	private		
			unsafe		<<unsafe>>			
		attribute sections			<<attributes>>			
		doc comments			Comment(->Documentation)			
	Property	name			name		Operation <<property>>	
		modifiers	internal		visibility	package		
			protected internal			protected <<internal>>		
			public			public		
			protected			protected		
			private			private		
			static		static			
			abstract		abstract			
			sealed		leaf			
			override		<<override>>			
			virtual		<<virtual>>			

C#					UModel				
			new		<<new >>				
			unsafe		<<unsafe>>				
		attribute sections			<<attributes>>				
		doc comments			Comment(->Documentation)				
		type			direction	return	Parameter		
		type dimensions			multiplicity				
		nullable			<<nullable>>				
		Get Accessor	modifiers	internal	visibility	internal	<<GetAccessor>>		
				protected internal		protected internal			
				protected		protected			
				private		private			
		Set Accessor	modifiers	internal	visibility	internal	<<SetAccessor>>		
				protected internal		protected internal			
				protected		protected			
				private		private			
	Operator	name			name			Operation <<operator>>	
		modifiers	public		visibility	public			
			static		static				
			unsafe		<<unsafe>>				
		attribute sections			<<attributes>>				
		doc comments			Comment(->Documentation)				
		type			direction	return	Parameter		
		Parameter	name		name				
			modifier	params	varArgList				
			type		type				
			type dimensions		multiplicity				
			type pointer		type modifier				
			nullable		<<nullable>>				

© 2019-2025 Altova GmbH

C#					UModel				
	Indexer	name (="this")			name (="this")			Operation <<indexer>>	
		modifiers	internal		visibility	package			
			protected internal			protected <<internal>>			
			public			public			
			protected			protected			
			private			private			
			static		static				
			abstract		abstract				
			sealed		leaf				
			override		<<override>>				
			virtual		<<virtual>>				
			new		<<new >>				
			unsafe		<<unsafe>>				
		attribute sections			<<attributes>>				
		doc comments			Comment(->Documentation)				
		type			direction	return	Parameter		
		Parameter	name		name				
			modifier	params	varArgList				
			type		type				
			type dimensions		multiplicity				
			type pointer		type modifier				
			nullable		<<nullable>>				
		Get Accessor	modifiers	internal	visibility	internal	<<GetAccessor>>		
				protected internal		protected internal			
				protected		protected			
				private		private			
		Set Accessor	modifiers	internal	visibility	internal	<<SetAccessor>>		
				protected internal		protected internal			
				protected		protected			

C#				UModel			
			private		private		
	Event	name		name			Operation <<event>>
		modifiers	internal	visibility	package		
			protected internal		protected <<internal>>		
			public		public		
			protected		protected		
			private		private		
			static	static			
			abstract	abstract			
			sealed	leaf			
			override	<<override>>			
			virtual	<<virtual>>			
			new	<<new >>			
			unsafe	<<unsafe>>			
		attribute sections		<<attributes>>			
		doc comments		Comment(->Documentation)			
		type		direction	return	Parameter	
		type dimensions		multiplicity			
		nullable		<<nullable>>			
		Add Accessor		<<AddRemoveAccessor>>			
		Remove Accessor					
	Type Parameter	name		name			Template Parameter
		constraint		constraining classifier			
		predefined constraint	struct	<<ValueTypeConstraint>>			
			class	<<ReferenceTypeConstraint>>			
			new ()	<<ConstructorConstraint>>			
		attribute sections		<<attributes>>			

© 2019-2025 Altova GmbH

C# Interface

C#				UModel			
Interface	name			name			Interface
	modifiers	internal		visibility	package		
		protected internal			protected <<internal>>		
		public			public		
		protected			protected		
		private			private		
		unsafe		<<unsafe>>			
		partial		<<partial>>			
		new		<<new >>			
	filename			code file name			
	associated projectfile/directory			ComponentRealization			
	base types			Generalization(s)			
	attribute sections			<<attributes>>			
	doc comments			Comment(->Documentation)			
	Method	name		name		Operation	
		modifiers	public	visibility	public		
			new	<<new >>			
			unsafe	<<unsafe>>			
		attribute sections		<<attributes>>			
		doc comments		Comment(->Documentation)			
		type		direction	return	Parameter	
		Parameter	name	name			
			modifiers ref	direction	inout		
			out		out		
			params	varArgList			
			type	type			
			type dimensions	multiplicity			
			type pointer	type modifier			

C#					UModel				
			this		<<this>>				
			nullable		<<nullable>>				
		Type Parameter	name		name		Template Parameter		
			constraint		constraining classifier				
			predefined constraint	struct	<<ValueTypeConstraint>>				
				class	<<ReferenceTypeConstraint>>				
				new ()	<<ConstructorConstraint>>				
			attribute sections		<<attributes>>				
	Property	name			name			Operation <<property>>	
		modifiers	public		visibility	public			
			new		<<new >>				
			unsafe		<<unsafe>>				
		attribute sections			<<attributes>>				
		doc comments			Comment(->Documentation)				
		type			direction	return	Parameter		
		type dimensions			multiplicity				
		nullable			<<nullable>>				
		Get Accessor	modifiers	internal	visibility	internal	<<GetAccessor>>		
				protected internal		protected internal			
				protected		protected			
				private		private			
		Set Accessor	modifiers	internal	visibility	internal	<<SetAccessor>>		
				protected internal		protected internal			
				protected		protected			
				private		private			
	Indexer	name (="this")			name (="this")			Operation <<indexer>>	
		modifiers	public		visibility	public			

© 2019-2025 Altova GmbH

C#					UModel				
			new		<<new >>				
			unsafe		<<unsafe>>				
		attribute sections			<<attributes>>				
		doc comments			Comment(->Documentation)				
		type			direction	return	Parameter		
		Parameter	name		name				
			modifier	params	varArgList				
			type		type				
			type dimensions		multiplicity				
			type pointer		type modifier				
			nullable		<<nullable>>				
		Get Accessor	modifiers	internal	visibility	internal	<<GetAccessor>>		
				protected internal		protected internal			
				protected		protected			
				private		private			
		Set Accessor	modifiers	internal	visibility	internal	<<SetAccessor>>		
				protected internal		protected internal			
				protected		protected			
				private		private			
	Event	name			name			Operation <<event>>	
		modifiers	public		visibility	public			
			new		<<new >>				
			unsafe		<<unsafe>>				
		attribute sections			<<attributes>>				
		doc comments			Comment(->Documentation)				
		type			direction	return	Parameter		
		type dimensions			multiplicity				
		nullable			<<nullable>>				
		Add Accessor			<<AddRemoveAccessor>>				

C#				UModel			
		Remove Accessor					
	Type Parameter	name		name		Template Parameter	
		constraint		constraining classifier			
		predefined constraint	struct	<<ValueTypeConstraint>>			
			class	<<ReferenceTypeConstraint>>			
			new ()	<<ConstructorConstraint>>			
		attribute sections		<<attributes>>			

C# Delegate

C#				UModel				
Delegate	name			name				Class <<delegate>>
	modifiers	internal		visibility	package			
		protected internal			protected <<internal>>			
		public			public			
		protected			protected			
		private			private			
		unsafe		<<unsafe>>				
		new		<<new >>				
	filename			code file name				
	associated projectfile/directory			ComponentRealization				
	attribute sections			<<attributes>>				
	doc comments			Comment(->Documentation)				
	type			direction	return	Parameter	Operation	
	Parameter	name		name				
		modifiers	ref	direction	inout			
			out		out			
			params	varArgList				
		type		type				
		type dimensions		multiplicity				
		type pointer		type modifier				

© 2019-2025 Altova GmbH

C#					UModel				
		nullable			<<nullable>>				
	Type Parameter	name			name			Template Parameter	
		constraint			constraining classifier				
		predefined constraint	struct		<<ValueTypeConstraint>>				
			class		<<ReferenceTypeConstraint>>				
			new ()		<<ConstructorConstraint>>				
		attribute sections			<<attributes>>				

C# Enum

C#			UModel		
Enum	name		name		Enumeration
	modifiers	internal	visibility	package	
		protected internal		protected <<internal>>	
		public		public	
		protected		protected	
		private		private	
		new	<<new >>		
	filename		code file name		
	associated projectfile/directory		ComponentRealization		
	base type		type	<<BaseType>>	
	attribute sections		<<attributes>>		
	doc comments		Comment(->Documentation)		
	Enum Constant	name	name	Enumeration Literal	
		default value	default		
		attribute sections	<<attributes>>		

C#			UModel		
		doc comments	Comment(->Documentation)		

C# Parameterized Type

C#	UModel
Parameterized Type	Anonymous Bound Element

6.6.2 VB.NET Mappings

The table below shows the one-to-one correspondence between:

- UModel elements and VB.NET code elements, when outputting model to code
- VB.NET code elements and UModel model elements, when inputting code into model

VB.NET			UModel		
Project	projectfile		projectfile		Component
	directory		directory		
Namespace	name		name		Package <<namespace>>
Class	name		name		Class
	modifiers	Friend	visibility	package	
		Protected Friend		protected <<Friend>>	
		Public		public	
		Protected		protected	
		Private		private	
		NotInheritable	leaf		
		MustInherit	abstract		
		Partial	<<Partial>>		
		Shadows	<<Shadows>>		
	filename		code file name		
	associated projectfile/directory		ComponentRealization		
	base types		Generalization, InterfaceRealization(s)		

© 2019-2025 Altova GmbH

VB.NET			UModel		
attribute sections			<<Attributes>>		
doc comments			Comment(->Documentation)		
Field	name		name		Property
	modifiers	Friend	visibility	package	
		Protected Friend		protected <<Friend>>	
		Public		public	
		Protected		protected	
		Private		private	
		Shared	static		
		ReadOnly	readonly		
		Shadows	<<Shadows>>		
	type		type		
	type dimensions		multiplicity		
	nullable		<<Nullable>>		
	default value		default		
	attribute sections		<<Attributes>>		
	doc comments		Comment(->Documentation)		
Constant	name		name		Property <<Const>>
	modifiers	Friend	visibility	package	
		Protected Friend		protected <<Friend>>	
		Public		public	
		Protected		protected	
		Private		private	
		Shadows	<<Shadows>>		
	type		type		
	type dimensions		multiplicity		
	nullable		<<Nullable>>		
	default value		default		
	attribute sections		<<Attributes>>		
	doc comments		Comment(->Documentation)		

VB.NET					UModel				
	Method	name			name			Operation	
		modifiers	Friend		visibility	package			
			Protected Friend			protected <<Friend>>			
			Public			public			
			Protected			protected			
			Private			private			
			Shared		static				
			MustOverride		abstract				
			NotOverridable		leaf				
			Overrides		<<Overrides>>				
			Overridable		<<Overridable>>				
			Partial		<<Partial>>				
			Shadows		<<Shadows>>				
			Overloads		<<Overloads>>				
		attribute sections			<<Attributes>>				
		doc comments			Comment(->Documentation)				
		implemented interfaces			implements				
		type (function)			direction	return	Parameter		
		Parameter	name		name				
			modifiers	ByRef	direction	inout			
				ByVal		in			
				ParamArray	varArgList				
				Optional	default				
			type		type				
			type dimensions		multiplicity				
			nullable		<<Nullable>>				
		Type Parameter	name		name		Template Parameter		
			constraint		constraining classifier				
			predefined	Structure	<<ValueTypeConstraint>>				

© 2019-2025 Altova GmbH

VB.NET					UModel			
			constraint	Class	<<ReferenceTypeConstraint>>			
				New	<<ConstructorConstraint>>			
			attribute sections		<<Attributes>>			
	Constructor	name			name			Operation <<Constructor>>
		modifiers	Friend		visibility	package		
			Protected Friend			protected <<Friend>>		
			Public			public		
			Protected			protected		
			Private			private		
			Shared		static			
		attribute sections			<<Attributes>>			
		doc comments			Comment(->Documentation)			
		Parameter	name		name		Parameter	
			modifiers	ByRef	direction	inout		
				ByVal		in		
				ParamArray	varArgList			
				Optional	default			
			type		type			
			type dimensions		multiplicity			
			nullable		<<Nullable>>			
	Property	name			name			Operation <<Property>>
		modifiers	Friend		visibility	package		
			Protected Friend			protected <<Friend>>		
			Public			public		
			Protected			protected		
			Private			private		
			Default		<<Property>> (Default <= IsDefault)			
			Shared		static			

VB.NET					UModel						
			MustOverride		abstract						
			NotOverridable		leaf						
			Overrides		<<Overrides>>						
			Overridable		<<Overridable>>						
			Shadows		<<Shadows>>						
			Overloads		<<Overloads>>						
			ReadOnly		<<GetAccessor>> (without <<SetAccessor>>)						
			WriteOnly		<<SetAccessor>> (without <<GetAccessor>>)						
		attribute sections			<<Attributes>>						
		doc comments			Comment(->Documentation)						
		type			direction	return	Parameter				
		type dimensions			multiplicity						
		nullable			<<Nullable>>						
		Get Accessor	modifiers	Friend	visibility	Friend	<<GetAccessor>>				
				Protected Friend		Protected Friend					
				Protected		Protected					
				Private		Private					
		Set Accessor	modifiers	Friend	visibility	Friend	<<SetAccessor>>				
				Protected Friend		Protected Friend					
				Protected		Protected					
				Private		Private					
	Operator	name			name			Operation <<Operator>>			
		modifiers	Public		visibility	Public					
			Shared		static						
			Narrowing		name <= Narrowing						
			Widening		name <= Widening						
		attribute sections			<<Attributes>>						
		doc comments			Comment(->Documentation)						

© 2019-2025 Altova GmbH

VB.NET					UModel				
		type			direction	return	Parameter		
		Parameter	name		name				
			modifier	ByVal	direction	in			
			type		type				
			type dimensions		multiplicity				
			nullable		<<Nullable>>				
	Event	name			name			Operation <<Event>>	
		modifiers	Friend		visibility	package			
			Protected Friend			protected <<Friend>>			
			Public			public			
			Protected			protected			
			Private			private			
			Shared		static				
			MustOverride		abstract				
			NotOverridable		leaf				
			Overrides		<<Overrides>>				
			Overridable		<<Overridable>>				
			Shadows		<<Shadows>>				
			Overloads		<<Overloads>>				
		kind	without specifying a delegate type		<<Event>> (Type <= Simple)				
			with specifying a delegate type		<<Event>> (Type <= Regular)				
			with custom accessors		<<Event>> (Type <= Custom)				
		attribute sections			<<Attributes>>				
		doc comments			Comment(->Documentation)				
		type			direction	return	Parameter		
		type dimensions			multiplicity				
		nullable			<<Nullable>>				
	Type Parameter	name			name			Template Parameter	
		constraint			constraining classifier				

VB.NET				UModel			
		predefined constraint	Structure	<<ValueTypeConstraint>>			
			Class	<<ReferenceTypeConstraint>>			
			New	<<ConstructorConstraint>>			
		attribute sections		<<Attributes>>			
Structure	name			name			Class <<Structure>>
	modifiers	Friend		visibility	package		
		Protected Friend			protected <<Friend>>		
		Public			public		
		Protected			protected		
		Private			private		
		Partial		<<Partial>>			
		Shadows		<<Shadows>>			
	filename			code file name			
	associated projectfile/directory			ComponentRealization			
	base types			InterfaceRealization(s)			
	attribute sections			<<Attributes>>			
	doc comments			Comment(->Documentation)			
	Field	name		name		Property	
		modifiers	Friend	visibility	package		
			Public		public		
			Private		private		
			Shared	static			
			ReadOnly	readonly			
			Shadows	<<Shadows>>			
		type		type			
		type dimensions		multiplicity			
		nullable		<<Nullable>>			
		default value		default			
		attribute sections		<<Attributes>>			
		doc comments		Comment(->Documentation)			

© 2019-2025 Altova GmbH

VB.NET					UModel				
	Constant	name			name			Property <<Const>>	
		modifiers	Friend		visibility	package			
			Public			public			
			Private			private			
			Shadows		<<Shadows>>				
		type			type				
		type dimensions			multiplicity				
		nullable			<<Nullable>>				
		default value			default				
		attribute sections			<<Attributes>>				
		doc comments			Comment(->Documentation)				
	Method	name			name			Operation	
		modifiers	Friend		visibility	package			
			Public			public			
			Private			private			
			Shared		static				
			MustOverride		abstract				
			NotOverridable		leaf				
			Overrides		<<Overrides>>				
			Overridable		<<Overridable>>				
			Partial		<<Partial>>				
			Shadows		<<Shadows>>				
			Overloads		<<Overloads>>				
		attribute sections			<<Attributes>>				
		doc comments			Comment(->Documentation)				
		implemented interfaces			implements				
		type (function)			direction	return	Parameter		
		Parameter	name		name				
			modifiers	ByRef	direction	inout			
				ByVal		in			

VB.NET					UModel			
				ParamArray	varArgList			
				Optional	default			
			type		type			
			type dimensions		multiplicity			
			nullable		<<Nullable>>			
		Type Parameter	name		name	Template Parameter		
			constraint		constraining classifier			
			predefined constraint	Structure	<<ValueTypeConstraint>>			
				Class	<<ReferenceTypeConstraint>>			
				New	<<ConstructorConstraint>>			
			attribute sections		<<Attributes>>			
	Constructor	name			name		Operation <<Constructor>>	
		modifiers	Friend		visibility	package		
			Public			public		
			Private			private		
			Shared		static			
		attribute sections			<<Attributes>>			
		doc comments			Comment(->Documentation)			
		Parameter	name		name		Parameter	
			modifiers	ByRef	direction	inout		
				ByVal		in		
				ParamArray	varArgList			
				Optional	default			
			type		type			
			type dimensions		multiplicity			
			nullable		<<Nullable>>			
	Property	name			name		Operation <<Propert	

© 2019-2025 Altova GmbH

VB.NET					UModel				
		modifiers	Friend		visibility	package		y>>	
			Public			public			
			Private			private			
			Shared		static				
			Default		<<Property>> (Default <= IsDefault)				
			MustOverride		abstract				
			NotOverridable		leaf				
			Overrides		<<Overrides>>				
			Overridable		<<Overridable>>				
			Shadow s		<<Shadow s>>				
			Overloads		<<Overloads>>				
			ReadOnly		<<GetAccessor>> (w ithout <<SetAccessor>>)				
			WriteOnly		<<SetAccessor>> (w ithout <<GetAccessor>>)				
		attribute sections			<<Attributes>>				
		doc comments			Comment(->Documentation)				
		type			direction	return	Parameter		
		type dimensions			multiplicity				
		nullable			<<Nullable>>				
		Get Accessor	modifiers	Friend	visibility	Friend	<<GetAccessor>>		
				Private		Private			
		Set Accessor	modifiers	Friend	visibility	Friend	<<SetAccessor>>		
				Private		Private			
	Operator	name			name			Operation <<Operator>>	
		modifiers	Public		visibility	Public			
			Shared		static				
			Narrow ing		name <= Narrow ing				
			Widening		name <= Widening				
		attribute sections			<<Attributes>>				
		doc comments			Comment(->Documentation)				

VB.NET					UModel				
		type			direction	return	Parameter		
		Parameter	name		name				
			modifier	ByVal	direction	in			
			type		type				
			type dimensions		multiplicity				
			nullable		<<Nullable>>				
	Event	name			name			Operation <<Event>>	
		modifiers	Friend		visibility	package			
			Public			public			
			Private			private			
			Shared		static				
			MustOverride		abstract				
			NotOverridable		leaf				
			Overrides		<<Overrides>>				
			Overridable		<<Overridable>>				
			Shadows		<<Shadows>>				
			Overloads		<<Overloads>>				
		kind	without specifying a delegate type		<<Event>> (Type <= Simple)				
			with specifying a delegate type		<<Event>> (Type <= Regular)				
			with custom accessors		<<Event>> (Type <= Custom)				
		attribute sections			<<Attributes>>				
		doc comments			Comment(->Documentation)				
		type			direction	return	Parameter		
		type dimensions			multiplicity				
		nullable			<<Nullable>>				
	Type Parameter	name			name			Template Parameter	
		constraint			constraining classifier				
		predefined constraint	Structure		<<ValueTypeConstraint>>				
			Class		<<ReferenceTypeConstraint>>				

© 2019-2025 Altova GmbH

VB.NET				UModel			
			New	<<ConstructorConstraint>>			
		attribute sections		<<Attributes>>			
Interface	name			name			Interface
	modifiers	Friend		visibility	package		
		Protected Friend			protected <<Friend>>		
		Public			public		
		Protected			protected		
		Private			private		
		Shadows		<<Shadows>>			
	filename			code file name			
	associated projectfile/directory			ComponentRealization			
	base types			Generalization(s)			
	attribute sections			<<Attributes>>			
	doc comments			Comment(->Documentation)			
	Method	name		name			Operation
		modifiers	Public	visibility	public		
			Shadows	<<Shadows>>			
		attribute sections		<<Attributes>>			
		doc comments		Comment(->Documentation)			
		type (function)		direction	return	Parameter	
		Parameter	name	name			
			modifiers ByRef	direction	inout		
			modifiers ByVal		in		
			modifiers ParamArray	varArgList			
			modifiers Optional	default			
			type	type			
			type dimensions	multiplicity			
			nullable	<<Nullable>>			
		Type Parameter	name	name		Template Parameter	

VB.NET					UModel				
			constraint		constraining classifier				
			predefined constraint	Structure	<<ValueTypeConstraint>>				
				Class	<<ReferenceTypeConstraint>>				
				New	<<ConstructorConstraint>>				
			attribute sections		<<Attributes>>				
	Property	name			name			Operation <<Property>>	
		modifiers	Public		visibility	public			
			Default		<<Property>> (Default <= IsDefault)				
			Shadows		<<Shadows>>				
			ReadOnly		<<GetAccessor>> (without <<SetAccessor>>)				
			WriteOnly		<<SetAccessor>> (without <<GetAccessor>>)				
		attribute sections			<<Attributes>>				
		doc comments			Comment(->Documentation)				
		type			direction	return	Parameter		
		type dimensions			multiplicity				
		nullable			<<Nullable>>				
	Event	name			name			Operation <<Event>>	
		modifiers	Public		visibility	public			
			Shadows		<<Shadows>>				
		kind	without specifying a delegate type		<<Event>> (Type <= Simple)				
			with specifying a delegate type		<<Event>> (Type <= Regular)				
		attribute sections			<<Attributes>>				
		doc comments			Comment(->Documentation)				
		type			direction	return	Parameter		
		type dimensions			multiplicity				
		nullable			<<Nullable>>				

© 2019-2025 Altova GmbH

VB.NET				UModel			
	Type Parameter	name		name		Template Parameter	
		constraint		constraining classifier			
		predefined constraint	Structure	<<ValueTypeConstraint>>			
			Class	<<ReferenceTypeConstraint>>			
			New	<<ConstructorConstraint>>			
		attribute sections		<<Attributes>>			
Delegate	name			name			Class <<Delegate>>
	modifiers	Friend		visibility	package		
		Protected Friend			protected <<Friend>>		
		Public			public		
		Protected			protected		
		Private			private		
		Shadows		<<Shadows>>			
	filename			code file name			
	associated projectfile/directory			ComponentRealization			
	attribute sections			<<Attributes>>			
	doc comments			Comment(->Documentation)			
	type			direction	return	Parameter	Operation
	Parameter	name		name			
		modifiers	ByRef	direction	inout		
			ByVal		in		
		type		type			
		type dimensions		multiplicity			
		nullable		<<Nullable>>			
	Type Parameter	name		name			Template Parameter
		constraint		constraining classifier			
		predefined constraint	struct	<<ValueTypeConstraint>>			
			class	<<ReferenceTypeConstraint>>			
			new()	<<ConstructorConstraint>>			
		attribute sections		<<Attributes>>			

VB.NET			UModel		
Enum	name		name		Enumeration
	modifiers	Friend	visibility	package	
		Protected Friend		protected <<Friend>>	
		Public		public	
		Protected		protected	
		Private		private	
		Shadows	<<Shadows>>		
	filename		code file name		
	associated projectfile/directory		ComponentRealization		
	base type		type	<<BaseType>>	
	attribute sections		<<Attributes>>		
	doc comments		Comment(->Documentation)		
	Enum Constant	name	name	Enumeration Literal	
		default value	default		
		attribute sections doc comments	<<Attributes>>		
			Comment(->Documentation)		
Parameterized Type			Anonymous Bound Element		

6.6.3 Java Mappings

The table below shows the one-to-one correspondence between:

- UModel elements and Java code elements, when outputting model to code
- Java code elements and UModel model elements, when inputting code into model

Java		UModel	
Project	projectfile	projectfile	Component
	directory	directory	
Package	name	name	Package <<namespace>>
Class	name	name	Class

© 2019-2025 Altova GmbH

Java				UModel			
	modifiers	package		visibility	package		
		public			public		
		protected			protected		
		private			private		
		abstract		abstract			
		strictfp		<<strictfp>>			
		final		<<final>>			
	filename			code file name			
	associated projectfile/directory			ComponentRealization			
	extends clause			Generalization			
	implements clause			InterfaceRealization(s)			
	java docs			Comment(->Documentation)			
	Field	name		name		Property	
		modifiers	package	visibility	package		
			public		public		
			protected		protected		
			private		private		
			static	static			
			transient	<<transient>>			
			volatile	<<volatile>>			
			final	<<final>>			
		type		type			
		type dimensions		multiplicity			
		default value		default			
		java docs		Comment(->Documentation)			
	Method	name		name		Operation	
		modifiers	package	visibility	package		
			public		public		
			protected		protected		
			private		private		

Java					UModel				
			static		static				
			abstract		abstract				
			final		<<final>>				
			native		<<native>>				
			strictfp		<<strictfp>>				
			synchronized		<<synchronized>>				
		throws clause			raised exceptions				
		java docs			Comment(->Documentation)				
		type			direction	return	Parameter		
		Parameter	name		name				
			modifier	final	<<final>>				
			...		varArgList				
			type		type				
			type dimensions		multiplicity				
		Type Parameter	name		name		Template Parameter		
			bound		constraining classifier				
	Constructor	name			name			Operation <<constructor>>	
		modifiers	public		visibility	public			
			protected			protected			
			private			private			
		throws clause			raised exceptions				
		java docs			Comment(->Documentation)				
		Parameter	name		name		Parameter		
			modifier	final	<<final>>				
			...		varArgList				
			type		type				
			type dimensions		multiplicity				
		Type Parameter	name		name		Template Parameter		
			bound		constraining classifier				

© 2019-2025 Altova GmbH

Java					UModel				
	Type Parameter	name			name			Template Parameter	
		bound			constraining classifier				
Interface	name				name				Interface
	modifiers	package			visibility	package			
		public				public			
		protected				protected			
		private				private			
		abstract			abstract				
		strictfp			<<strictfp>>				
	filename				code file name				
	associated projectfile/directory				ComponentRealization				
	extends clause				Generalization(s)				
	java docs				Comment(->Documentation)				
	Field	name			name			Property	
		modifiers	public		visibility	public			
			static		static				
			final		<<final>>				
		type			type				
		type dimensions			multiplicity				
		default value			default				
		java docs			Comment(->Documentation)				
	Method	name			name			Operation	
		modifiers	public		visibility	public			
			abstract		abstract				
		throws clause			raised exceptions				
		java docs			Comment(->Documentation)				
		type			direction	return	Parameter		
		Parameter	name		name				
			modifier	final	<<final>>				
			...		varArgList				

Java				UModel			
			type	type			
			type dimensions	multiplicity			
		Type Parameter	name	name	Template Parameter		
			bound	constraining classifier			
	Type Parameter	name		name			Template Parameter
		bound		constraining classifier			
Enum	name			name			Enumeration
	modifiers	package		visibility	package		
		public			public		
		protected			protected		
		private			private		
	filename			code file name			
	associated projectfile/directory			ComponentRealization			
	java docs			Comment(->Documentation)			
	Enum Constant	name		name		Enumeration Literal	
	Field	name		name		Property	
		modifiers	package	visibility	package		
			public		public		
			protected		protected		
			private		private		
			static	static			
			transient	<<transient>>			
			volatile	<<volatile>>			
			final	<<final>>			
		type		type			
		type dimensions		multiplicity			
		default value		default			
		java docs		Comment(->Documentation)			
	Method	name		name		Operation	

© 2019-2025 Altova GmbH

Java					UModel				
		modifiers	package		visibility	package			
			public			public			
			protected			protected			
			private			private			
			static		static				
			abstract		abstract				
			final		<<final>>				
			native		<<native>>				
			strictfp		<<strictfp>>				
			synchronized		<<synchronized>>				
		throws clause			raised exceptions				
		java docs			Comment(->Documentation)				
		type			direction	return	Parameter		
		Parameter	name		name				
			modifier	final	<<final>>				
			...		varArgList				
			type		type				
			type dimensions		multiplicity				
		Type Parameter	name		name		Template Parameter		
			bound		constraining classifier				
	Constructor	name			name			Operation <<constructor>>	
		modifiers	public		visibility	public			
			protected			protected			
			private			private			
		throws clause			raised exceptions				
		java docs			Comment(->Documentation)				
		Parameter	name		name		Parameter		
			modifier	final	<<final>>				
			...		varArgList				
			type		type				

Java				UModel			
			type dimensions	multiplicity			
		Type Parameter	name	name	Template Parameter		
			bound	constraining classifier			
Parameterized Type				Anonymous Bound Element			
Annotation				<<annotations> modifiers			

6.6.4 XML Schema Mappings

The table below shows the one-to-one correspondence between:

- UModel elements and XML Schema elements, when outputting model to code
- XML Schema elements and UModel model elements, when inputting code into model

Legend:

XSD/UML Element

Stereotype property (=tagged value)

XSD			UModel		
file path			projectfile		Component
schema	target namespace		name		Package <<namespace>>
	attributeFormDefault		attributeFormDefault		Class <<schema>>
	blockDefault		blockDefault		
	elementFormDefault		elementFormDefault		
	finalDefault		finalDefault		
	version		version		
	xml:lang		xml:lang		
	xmlns		xmlns		
	annotation	source	source		
		appinfo		Comment <<appinfo	

© 2019-2025 Altova GmbH

XSD					UModel			
								>>
		documentation	xml:lang		xml:lang			Comment <<documentation>>
	attributeGroup	name			name			Class <<attributeGroup>>
		annotation	appinfo				Comment <<appinfo>>	
			documentation				Comment <<documentation>>	
		attribute	name		name		Property <<attribute>>	
			form		form			
			use		use			
			ref		type			
			type					
			default		default			
			fixed			fixed		
		attributeGroup	ref		type		Property <<attributeGroup>>	
		anyAttribute	namespace		namespace		Property <<anyAttribute>>	
			processContents		processContents			
	attribute	name			name			Class <<attribute>>
		form			form			
		use			use			
		type			type		Property	
		default			default			
		fixed				fixed		
		annotation	appinfo				Comment <<appinfo>>	
			documentation				Comment <<documentation>>	

XSD					UModel			
		simpleType			name (= name of Class + "_anonymousType[n]")	DataType <<simpleType>>		
	element	name			name			Class <<element>>
		abstract			abstract			
		block			block			
		final			final			
		form			form			
		nillable			nillable			
		type			type		Property	
		default			default			
		fixed				fixed		
		substitutionGroup			general		Generalization <<substitution>>	
		annotation	appinfo				Comment <<appinfo>>	
			documentation				Comment <<documentation>>	
		simpleType			name (= name of Class + "_anonymousType[n]")		DataType <<simpleType>>	
		complexType			name (= name of Class + "_anonymousType[n]")		Class <<complexType>>	
	group	name			name			Class <<group>>
		annotation	appinfo				Comment <<appinfo>>	
			documentation				Comment <<documentation>>	
		all			name (= "_all")		Property	
					name (= "mg"_ + "all")		Class	

© 2019-2025 Altova GmbH

XSD					UModel				
			annotation	appinfo		Comment <<appinfo>>			
				documentation		Comment <<documentation>>			
			element	name	name	Property <<element>>			
				ref	type				
				type					
		choice			name (= "_choice")		Property		
					name (= "mg"_ + "choice")		Class <<choice>>		
			annotation	appinfo		Comment <<appinfo>>			
				documentation		Comment <<documentation>>			
			element	name	name	Property <<element>>			
				ref	type				
				type					
			group			Property <<group>>			
			any	namespace	namespace	Property <<any>>			
				processContents	processContents				
			choice			Property			
						Class <<choice>>			
			sequence			Property			
						Class <<sequence>>			
		sequence			name (= "_sequence")		Property		

XSD					UModel			
					name (= "mg"_ + "sequence")		Class <<sequence>>	
			annotation	appinfo		Comment <<appinfo>>		
				documentation		Comment <<documentation>>		
			element	name	name	Property <<element>>		
				ref	type			
				type				
			group			Property <<group>>		
			any	namespace	namespace	Property <<any>>		
				processContents	processContents			
			choice			Property		
						Class <<choice>>		
			sequence			Property		
						Class <<sequence>>		
	notation	name			name			DataType <<notation>>
		system			system			
		public			public			
		annotation	appinfo				Comment <<appinfo>>	
			documentation				Comment <<documentation>>	
	complexType	name			name			Class <<complexType>>
		abstract			abstract			
		block			block			

© 2019-2025 Altova GmbH

XSD					UModel			
		final			final			
		mixed			mixed			
		annotation	source		source			
			appinfo			Comment <<appinfo>>		
			documentation	xml:lang	xml:lang	Comment <<documentation>>		
		group			name (= "_ref[n]")	Property <<group>>		
			maxOccurs		multiplicity			
			minOccurs					
			ref		type			
		all			name (= "mg"_ + "all")	Class <<all>>		
					name (= "_all")	Property		
			maxOccurs		multiplicity			
			minOccurs					
		choice			name (= "mg"_ + "choice[n]")	Class <<choice>>		
					name (= "_choice[n]")	Property		
			maxOccurs		multiplicity			
			minOccurs					
		sequence			name (= "mg"_ + "sequence[n]")	Class <<sequence>>		
					name (= "_sequence[n]")	Property		
			maxOccurs		multiplicity			
			minOccurs					
		attribute	name		name	Property <<attribute>>		
			ref		type			

XSD					UModel			
			type					
		attributeGroup	ref		type	Property <<attributeGroup>>		
		anyAttribute	namespace		namespace	Property <<anyAttribute>>		
			processContents		processContents			
		complexContent	restriction	base	general	Generalization <<restriction>>		
			extension			Generalization <<extension>>		
	simpleType	name			name		DataType <<simpleType>> Enumeration <<simpleType>>	
		final			final			
		annotation	source		source			
			appinfo				Comment <<appinfo>>	
			documentation	xml:lang	xml:lang		Comment <<documentation>>	
		list	itemType		name (= "_itemType")	Property <<itemType>>	<<list>>	
			simpleType		DataType <<simpleType>>			
		union	memberTypes		name (= "memberType[n]")	Property <<memberType>>	<<union>>	
			simpleType		DataType <<simpleType>>			
		minExclusive	value		value		<<minExclusive>>	
			fixed		fixed			
		minInclusive	value		value		<<minInclusive>>	
			fixed		fixed			
		maxExclu	value		value		<<maxExc	

© 2019-2025 Altova GmbH

XSD					UModel			
			fixed		fixed			
		maxInclusive	value		value	<<maxInclusive>>		
			fixed		fixed			
		totalDigits	value		value	<<totalDigits>>		
			fixed		fixed			
		fractionDigits	value		value	<<fractionDigits>>		
			fixed		fixed			
		length	value		value	<<length>>		
			fixed		fixed			
		minLength	value		value	<<minLength>>		
			fixed		fixed			
		maxLength	value		value	<<maxLength>>		
			fixed		fixed			
		whitespace	value		value	<<whitespace>>		
			fixed		fixed			
		pattern	value		value	<<whitespace>>		
		enumeration	value		name	EnumerationLiteral		
		simpleType				DataType <<simpleType>>		
		restriction	base		general	Generalization <<restriction>>		
	complexType simpleContent	name			name		DataType <<complexType>> <<simpleContent>>	
		annotation	source		source			
			appinfo			Comment <<appinfo>>		
			documentation	xml:lang	xml:lang	Comment <<documentation>>		

XSD					UModel				
		minExclusive	value		value		<<minExclusive>>		
			fixed		fixed				
		minInclusive	value		value		<<minInclusive>>		
			fixed		fixed				
		maxExclusive	value		value		<<maxExclusive>>		
			fixed		fixed				
		maxInclusive	value		value		<<maxInclusive>>		
			fixed		fixed				
		totalDigits	value		value		<<totalDigits>>		
			fixed		fixed				
		fractionDigits	value		value		<<fractionDigits>>		
			fixed		fixed				
		length	value		value		<<length>>		
			fixed		fixed				
		minLength	value		value		<<minLength>>		
			fixed		fixed				
		maxLength	value		value		<<maxLength>>		
			fixed		fixed				
		whitespace	value		value		<<whitespace>>		
			fixed		fixed				
		pattern	value		value		<<whitespace>>		
		attribute	name		name		Property <<attribute>>		
			ref		type				
			type						
		attributeGroup	ref		type		Property <<attributeGroup>>		
		anyAttribute	namespace		namespace		Property <<anyAttribute>>		

© 2019-2025 Altova GmbH

XSD					UModel				
			processContents		processContents				
		simpleType					DataType <<simpleType>>		
		restriction	base		general		Generalization <<restriction>>		
		extension	base		general		Generalization <<extension>>		
	import	schemaLocation			schemaLocation			ElementImport <<import>>	
		namespace			namespace				
	include	schemaLocation			schemaLocation			ElementImport <<include>>	
	redefine	schemaLocation			schemaLocation			ElementImport <<redefine>>	
		simpleType			<<redefine>>			DataType <<simpleType>>	
		complexType						Class <<complexType>>	
		attributeGroup						Class <<attributeGroup>>	
		group						Class <<group>>	

6.6.5 Database Mappings

The table below shows the one-to-one correspondence between:

- UModel elements and database elements, when outputting model to code
- Database elements and UModel model elements, when inputting code into model

<table>
<tr><th colspan="6">Database</th><th colspan="6">UModel</th></tr>
<tr><td>Database</td><td colspan="5">connection</td><td colspan="5">connection</td><td>Component</td></tr>
<tr><td rowspan="20">Database</td><td colspan="5">name</td><td colspan="5">name</td><td rowspan="20">Package <<name space>> <<Database>></td></tr>
<tr><td>Schema</td><td colspan="4">name</td><td colspan="4">name</td><td rowspan="19">Package <<name space>> <<Schema>></td></tr>
<tr><td rowspan="18">Table</td><td colspan="3">name</td><td colspan="3">name</td><td rowspan="18">Class <<Table>></td></tr>
<tr><td rowspan="11">Column</td><td>name</td><td>name</td><td>Property</td></tr>
<tr><td>Data Type</td><td>type</td></tr>
<tr><td>Not Null</td><td><<not_null>></td></tr>
<tr><td>Null</td><td><<nullable>></td></tr>
<tr><td>Length</td><td rowspan="3">Multiplicity</td></tr>
<tr><td>Precision</td></tr>
<tr><td>Scale</td></tr>
<tr><td>Default</td><td>default</td></tr>
<tr><td>Autoincrement</td><td><<autoincrement>></td></tr>
<tr><td>Part of Primary Key</td><td><<PK>></td></tr>
<tr><td>Part of Foreign Key</td><td><<FK>></td></tr>
<tr><td>Part of Unique Key</td><td><<unique>></td></tr>
<tr><td rowspan="2">Primary Key</td><td colspan="2">name</td><td colspan="2">name</td><td rowspan="2">Class <<PrimaryKey>></td></tr>
<tr><td>Column</td><td>name</td><td>name</td><td>Property</td></tr>
<tr><td rowspan="3">Foreign Key</td><td colspan="2">name</td><td colspan="2">name</td><td rowspan="3">Class <<ForeignKey>></td></tr>
<tr><td>Column</td><td>name</td><td>name</td><td>Property</td></tr>
<tr><td>Foreign Column</td><td>name</td><td>name</td><td>Property</td></tr>
<tr><td rowspan="2">Unique Key</td><td colspan="2">name</td><td colspan="2">name</td><td rowspan="2">Class <<UniqueKey>></td></tr>
<tr><td>Column</td><td>name</td><td>name</td><td>Property</td></tr>
<tr><td rowspan="3">Index</td><td colspan="2">name</td><td colspan="2">name</td><td rowspan="3">Class <<Index>></td></tr>
<tr><td rowspan="2">Column</td><td>name</td><td></td><td>Property</td></tr>
<tr><td>order: ascending</td><td><<ascending>></td></tr>
</table>

© 2019-2025 Altova GmbH

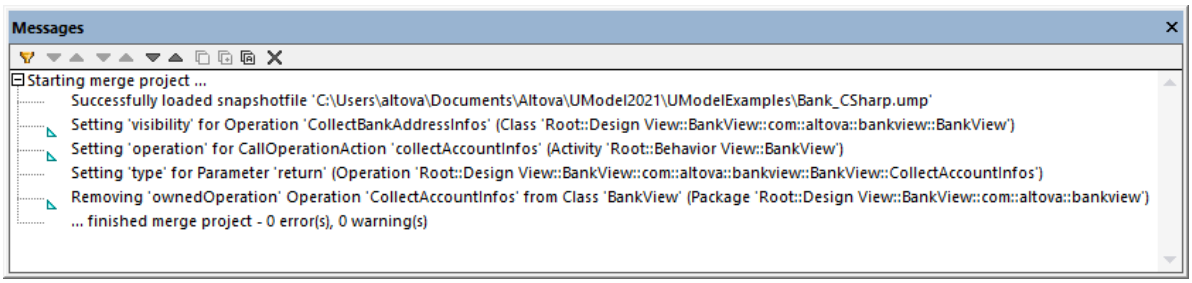
Database						UModel					
					order: descending	<<descending>>					
			CheckConstraint	name		name		Class <<CheckConstraint>>			
				definition		definition					
		View	name			name			Class <<View>>		
			definition			definition					
			Column	name		name		Property			
				Data Type		type					
				Not Null		<<not_null>>					
				Null		<<nullable>>					
				Length		Multiplicity					
				Precision							
				Scale							
				Default		default					
				Autoincrement		<<autoincrement>>					
		Stored Procedure	name			name		Operation <<StoredProcedure>>	Class <<StoredProcedures>>		
			definition			definition					
			Parameter	name		name	Parameter				
				direction mode		direction					
				data type		type					
		Function	name			name		Operation <<Function>>	Class <<Functions>>		
			definition			definition					
			Parameter	name		name	Parameter				
				direction mode		direction					
				data type		type					
		Trigger	name			name			Class <<Trigger>>		
			definition			definition					

6.7 Merging UModel Projects

It is possible to perform a two-way or three-way project merge in UModel. Both operations merge different UModel project files into a common UModel *.ump model. This option is useful if multiple persons are working on the same project at the same time, or you just want to consolidate your work into one model.

To merge two UML projects:

1. Open the UML file that is to be the target of the merge process, i.e. the file into which the second model will be merged - the merged project file.
2. Select the menu option **Project | Merge Project...**.
3. Select the second UML project that is to be merged into the first one. The Messages window reports on the merge process, and logs the relevant details.



Note: Clicking on one of the entries in the Messages window displays that modeling element in the Model Tree.

Merging results:

- New modeling elements i.e. those that do not exist in the source, are added to the merged model.
- Differences in the same modeling elements; the elements from the second model take precedence, e.g. there can only be one default value of an attribute, the default value of the second file is used.
- Diagram differences: UModel first checks to see if there are differences between diagrams of the two models. If there are, then the new/different diagram is added to the merged model (with a running number suffix, activity1 etc.) and the original diagram is retained. If there are no differences, then identical diagrams(s) are ignored, and nothing is changed. You can then decide which of the diagrams you want to keep or delete, you can of course keep both of them if you want.
- The whole merge process can be undone step-by-step by clicking the **Undo** toolbar button, or pressing **Ctrl+Z**.
- Clicking an entry in the message window displays that element in the Model Tree.
- The file name of the merged file (the first file you opened) is retained.

6.7.1 3-Way Project Merge

UModel supports the merging of multiple UModel projects that have been simultaneously edited by different developers, in a 3-way project merge. The 3-way project merge works with top-level UModel projects, i.e. main projects that may contain subprojects, it does not support individual file merging, when these files have unresolved references to other files.

© 2019-2025 Altova GmbH

When merging main projects, any editable subprojects are automatically merged as well. There is no need for a separate subproject merging process. For an example, see Example: Manual 3-Way Project Merge 278. Note the following:

- The whole merge process can be undone step-by-step by clicking the **Undo** toolbar button, or pressing **Ctrl+Z**.
- Clicking an entry in the message window displays that element in the Model Tree.
- The file name of the merged file, the first file you opened, is retained.

Merging results

In the following text, "source" means the initial/first project file you open before starting the merge process.

- New modeling elements in the second file i.e. that do not exist in the source, are added to the merged model.
- New modeling elements in the source file i.e. that do not exist in the second file, remain in the merged model.
- Deleted modeling elements from the second file i.e. those that still exist in the source, are removed from the merged model.
- Deleted modeling elements from the source file i.e. that still exist in the second file, remain deleted from the merged model.

Differences to the same modeling elements:

- If a property (e.g. the visibility of a class) is changed in either the source, or second file, the updated value is used in the merged model.
- If a property (e.g. the visibility of a class) is changed in both source and second file, the value of the second file is used (and a warning is shown in the messages window).

Moved elements:

- If an element is moved in the source, or second file, then the element is moved in the merged model.
- If an element is moved (to different parents) in both the source and second file, a prompt appears, and you have to manually select the parent element in the merged model.

Diagram differences:

UModel first checks to see if there are differences between diagrams of the two models. If yes, then the new/different diagram is added to the merged model (with a running number suffix, activity1 etc.) and the original diagram is retained. If there are no differences, then identical diagrams(s) are ignored, and nothing is changed. You can then decide which of the diagrams you want to keep or delete, you can of course keep both of them if you want.

Source control systems support for 3-way merging

When checking in/out project files, UModel automatically generates "Common ancestor" (or snapshot) files which are then used for the 3-way merge process. This enables a much finer merge result than the normal 2-way merge.

The specific source control system you use, determines if the automatic snapshot 3-way merge process is supported by UModel. A manual 3-way merge is however, always possible.

- Source control systems that perform automatic file merging without user intervention, will probably not support an automatic 3-way merge.
- Source control systems that prompt you to choose between Replace or Merge, when a project file has been changed, will generally support a 3-way merge. After the source control plug-in has replaced the file, selecting the Replace command activates the UModel file alert which then allows you to do a 3-way merge. UModel must be used for the check in/out process.
- Main projects as well as subprojects can be placed under source control. Changing data in a subproject automatically prompts you if the subproject(s) should be checked out.
- Each check in/out action, creates a Common ancestor, or a snapshot, file which is then used during the 3-way project merge process.

Note: Snapshot files are automatically created and used only with the standalone versions of UModel, i.e. these functions are not available in the Eclipse or Visual Studio plug-in versions.

Example

User A edits a UModel project file and changes the name of a class in the BankView Main diagram. User B opens the same project file and changes the visibility of the same class.

As snapshot files are created for each user, the snapshot editing history allows the individual changes to be merged into the project. Both the name and visibility changes are merged into the project file during the 3-way merge process.

6.7.2 Example: Manual 3-Way Project Merge

This example illustrates a simple 3-way project merge. Let's suppose that two users, Tom and Alice, created their own copies of a UModel project and made changes to them. There are now three versions of the same project: the original one, Tom's copy, and Alice's copy. In the context of 3-way merging, the original project represents the "common ancestor file".

For the scope of this example, let's assume that the common ancestor file is **Bank_CSharp.ump** project, available in the folder **C:\Users\<username>\Documents\Altova\UModel2025\UModelExamples**. The copies of Tom and Alice must be created manually. Therefore, let's first create two copies of the **Bank_Csharp.ump** project in child folders below the ...\UModelExamples folder. Let's call the child folders **Alice** and **Tom**; the project name can remain as is.

> Use the **File | Save Project As** command to create the copies of Tom and Alice. When prompted to adjust the relative paths, click **Yes**. This way you will avoid introducing syntax errors in the project copies.

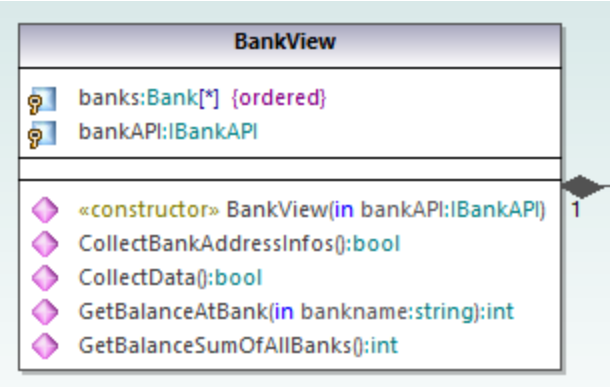
The goal of the example is to show how Alice should merge changes not only from the original **Bank_CSharp.ump**, but also from Tom's project into a new merged model (a so-called "3-way merge").

Step 1: Prepare Tom's project

Tom opens the **Bank_CSharp.ump** project file in folder **Tom**, opens the "BankView Main" diagram, and makes changes to the `BankView` class.

1. Operation `CollectAccountInfos():bool` is deleted from the `BankView` class.
2. The visibility of the `CollectBankAddressInfos():bool` operation is changed from "protected" to "public".

© 2019-2025 Altova GmbH

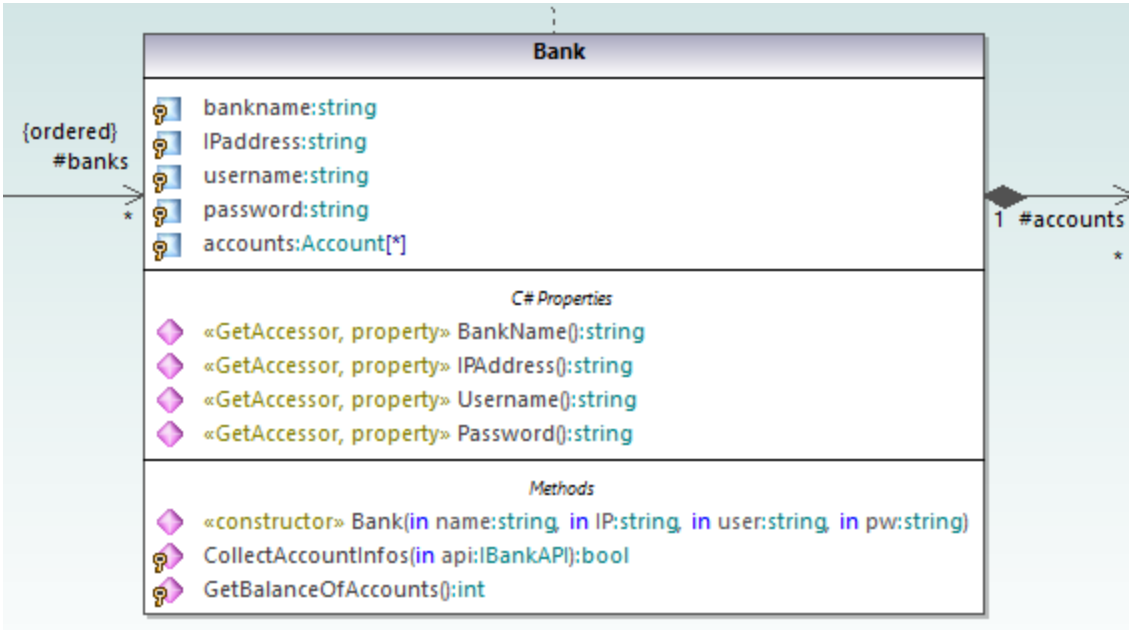


3. The project is then saved.

Step 2: Prepare Alice's project

Alice opens the **Bank_CSharp.ump** project file in folder **Alice**, opens the "BankView Main" diagram, and makes changes to the `Bank` class.

1. The operations `CollectAccountInfos` and `GetBalanceOfAccounts` are both changed from "public" to "protected".



2. The project is then saved.

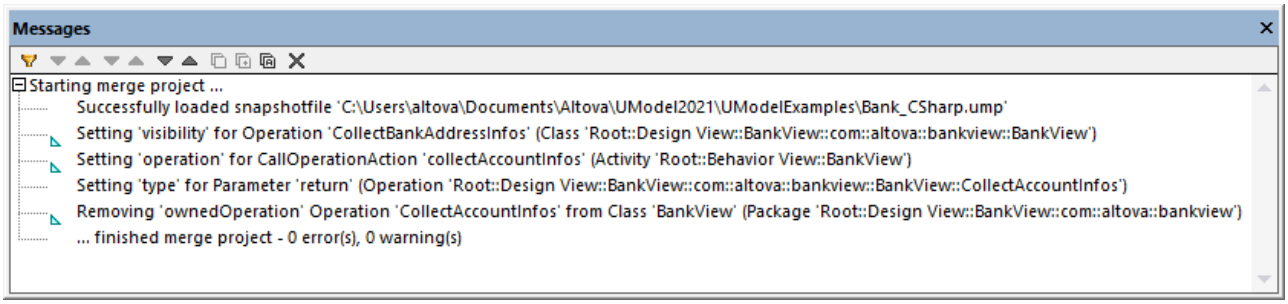
Step 3: Perform the 3-way merge

Alice now starts a 3-way project merge:

1. Open Alice's project from **Alice** folder.
2. On the **Project** menu, click **Merge Project (3-way)**, and select the project file changed by Tom from **Tom** folder.

3. You are now prompted to open the common ancestor file. Select the original **Bank_CSharp.ump** project file from the **...\UModelExamples** folder.

The 3-way merge process is started and you return to the project file from which you started the 3-way merge process, i.e. from the project file in the **Alice** folder. The Messages window shows you the merge process in detail.



The outcome of the 3-way merge is as follows:

- The changes made to the project by Tom are replicated in Alice's project.
- The changes made to the project by Alice are retained in the project file.

Note: The project file in the **Alice** folder should now be used as the common ancestor file for future 3-way merges between the project files in folders **Tom** and **Alice**.

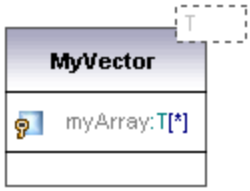
© 2019-2025 Altova GmbH

6.8 UML Templates

UModel supports the use of UML templates and their mapping to or from Java, C# and Visual Basic generics.

- Templates are "potential" model elements with unbound formal parameters.
- These parameterized model elements, describe a group of model elements of a particular type: classifiers, or operations.
- Templates cannot be used directly as types, the parameters have to be bound.
- Instantiate means binding the template parameters to actual values.
- Actual values for parameters are expressions.
- The binding between a template and model element, produces a new model element (a bound element) based on the template.
- If multiple constraining classifiers exist in C#, then the template parameters can be directly edited in the Properties tab, when the template parameter is selected.

Template **signature** display in UModel:



- Class template called **MyVector**, with formal template parameter "**T**", visible in the dashed rectangle.
- Formal parameters without type info (T) are implicitly classifiers: Class, Datatype, Enumeration, PrimitiveType, Interface. All other parameter types must be shown explicitly e.g. Integer.
- Property **myArray** with unbounded number of elements of type T.

  Right clicking the template and selecting **Show | Bound elements**, displays the actual bound elements.

Template **binding** display:

- A bound named template **intvector**
- Template of type, **MyVector**, where
- Parameter **T** is substituted/replaced by **int.**
- **"Substituted by"** is shown by **->**.

Template use in properties/operations:

An anonymous template binding:

- Property MyFloatVector of type **MyVector<T->float>**

Templates can also be defined when defining properties or operations. The autocomplete function helps you with the correct syntax when doing this.

- Operation1 returns a vector of floats.

6.8.1 Template Signatures

A Template signature is a string that specifies the formal template parameters. A template is a parameterized element that is used to generate new model elements by substituting/binding the formal parameters to actual parameters (values).

Formal template parameter

T
Template with a single untyped formal parameter
(stores elements of type T)

Multiple formal template parameters

KeyType:DateType, ValueType

Parameter substitution

T>aBaseClass

The parameter substitution must be of type "aBaseClass", or derived from it.

Default values for template parameters

T=aDefaultValue

Substituting classifiers

T>{contract}aBaseClass

allowsSubstitutable is true
Parameter must be a classifier that may be substituted for the classifier designated by the classifier name.

Constraining template parameters

T:Interface>anInterface

When constraining to anything other than a class, (interface, data type), the constraint is displayed after the colon ":" character. E.g. T is constrained to an interface (T:Interface) which must be of type "anInterface" (>anInterface).

Using wildcards in template signatures

T>vector<T->?<aBaseClass>

Template parameter T must be of type "vector" which contains objects which are a supertype of aBaseClass.

© 2019-2025 Altova GmbH

Extending template parameters
T>Comparable<T->T>

6.8.2 Template Binding

Template binding involves the substitution of the formal parameters by actual values, i.e. the template is instantiated. UModel automatically generates anonymously bound classes, when this binding occurs. Bindings can be defined in the class name field as shown below.

Substituting/binding formal parameters
vector <T->int>

Create bindings using the class name
a_float_vector:vector<T->float>

Binding multiple templates simultaneously
Class5:vector<T->int, map<KeyType->int, ValueType<T->int>

Using wildcards ? as parameters (Java 5.0)
vector<T->?>

Constraining wildcards - upper bounds (UModel extension)
vector<T->?>aBaseClass>

Constraining wildcards - lower bounds (UModel extension)
vector<T->?<aDerivedClass>

6.8.3 Template Usage in Operations and Properties

Operation returning a bound template
Class1
Operation1():vector<T->int>

Parameter T is bound to "int". Operation1 returns a vector of ints.

Class containing a template operation
Class1
Operation1<T>(in T):T

Using wildcards
Class1
Property1:vector<T->?>

This class contains a generic vector of unspecified type (? is the wildcard).

Typed properties can be displayed as associations as follows:

- Right click a property and select **Show | PropertyX as Association**, or
- Drag a property onto the diagram background.

© 2019-2025 Altova GmbH

7 Generating UML Documentation

Altova website: UML project documentation

Run the **Project | Generate Documentation** menu command to generate detailed documentation about your UML project in HTML, Microsoft Word, RTF or PDF format. The documentation generated by this command can be freely altered and used; permission from Altova to do so is not required.

Notes

- To generate documentation in PDF format or to customize the generated documentation, Altova StyleVision (https://www.altova.com/stylevision) must be installed and licensed.
- To generate documentation in Microsoft Word format, Microsoft Word 2000 or later is required.

Documentation is generated for the modeling elements you select in the Generate Documentation dialog box. You can either use the fixed design, or specify a custom StyleVision Power Stylesheet (SPS). Using a StyleVision Power Stylesheet enables you to customize the output of the generated documentation, see Customizing Output with StyleVision 294.

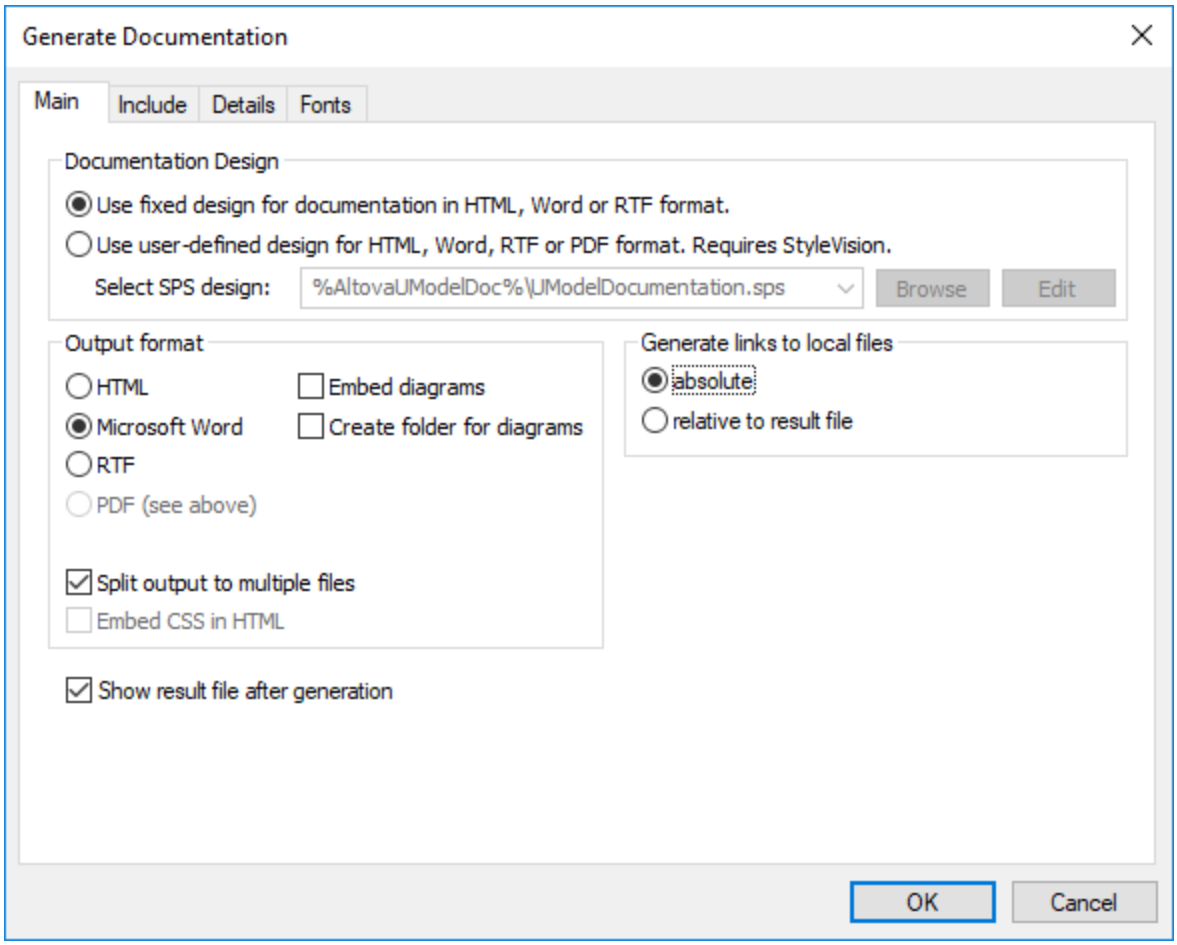
You can also create partial documentation of modeling elements. To do this, right-click an element (or multiple elements using **Ctrl+Click**) in the Model Tree and select **Generate Documentation**. The element can be a folder, class, interface, and so on. The documentation options are the same in both cases.

Related elements are hyperlinked in the generated output, enabling you to navigate from component to component. All manually created hyperlinks also appear in the documentation.

If your project contains UModel profiles (such as C#, Java, VB.NET, and so on), the generated documentation will include these if the **Included subprojects** option is enabled in the **Include** tab, see Documentation Generation Options 289.

To generate documentation:

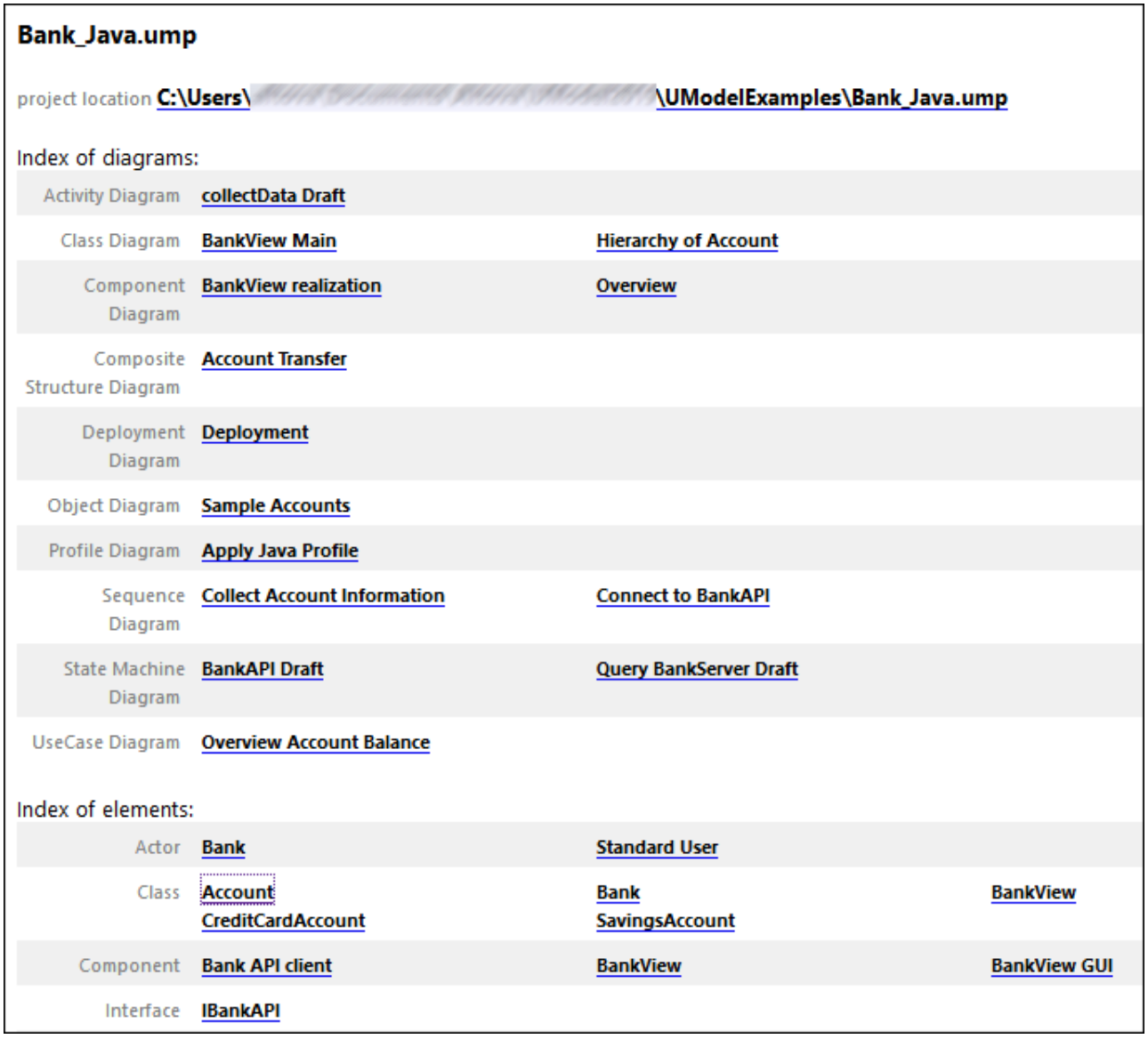
1. Open a project (for example, **C:\Users\<username>\Documents\Altova\UModel2025\UModelExamples\Bank_Java.ump**).
2. On the **Project** menu, click **Generate Documentation**.



3. Select an output format (HTML, Word, RTF, PDF).
4. Optionally, customize the generation options, see Documentation Generation Options 289.
5. Click **OK** and choose a target output folder.

The following image shows a fragment of UModel fixed-design documentation generated from the **Bank_Java.ump** project file.

© 2019-2025 Altova GmbH



Bank_Java.ump

project location **C:\Users\ \UModelExamples\Bank_Java.ump**

Index of diagrams:

Activity Diagram	collectData Draft		
Class Diagram	BankView Main	Hierarchy of Account	
Component Diagram	BankView realization	Overview	
Composite Structure Diagram	Account Transfer		
Deployment Diagram	Deployment		
Object Diagram	Sample Accounts		
Profile Diagram	Apply Java Profile		
Sequence Diagram	Collect Account Information	Connect to BankAPI	
State Machine Diagram	BankAPI Draft	Query BankServer Draft	
UseCase Diagram	Overview Account Balance		

Index of elements:

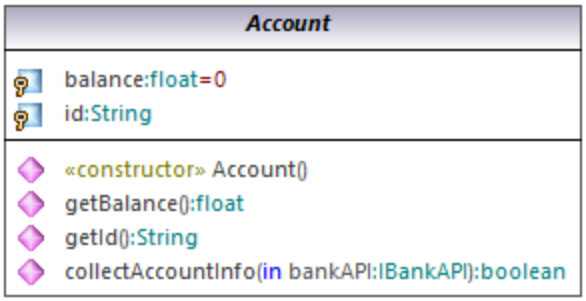
Actor	Bank	Standard User	
Class	Account CreditCardAccount	Bank SavingsAccount	BankView
Component	Bank API client	BankView	BankView GUI
Interface	IBankAPI		

As illustrated above, the generated documentation includes an index of diagrams and elements (with links) at the top of the HTML file.

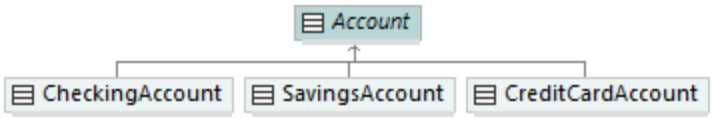
The image below shows a fragment of the generated documentation for the `Account` class. Note that the individual members in class diagrams are also hyperlinked to their definitions. For example, clicking a property or operation takes you to its definition. The hierarchy classes, as well as all underlined text, are also hyperlinked.

Class **Account**

diagram



hierarchy



owner **bankview**

properties

qualified name	Design View::BankView::com::altova::bankview::Account
visibility	public
leaf	false
abstract	true
isFinalSpecialization	false
active	false
code file name	Account.java
code file path	C:\UML_Bank_Sample\JavaCode\com\altova\bankview\Account.java
«annotations»	false
«static»	false
«strictfp»	false

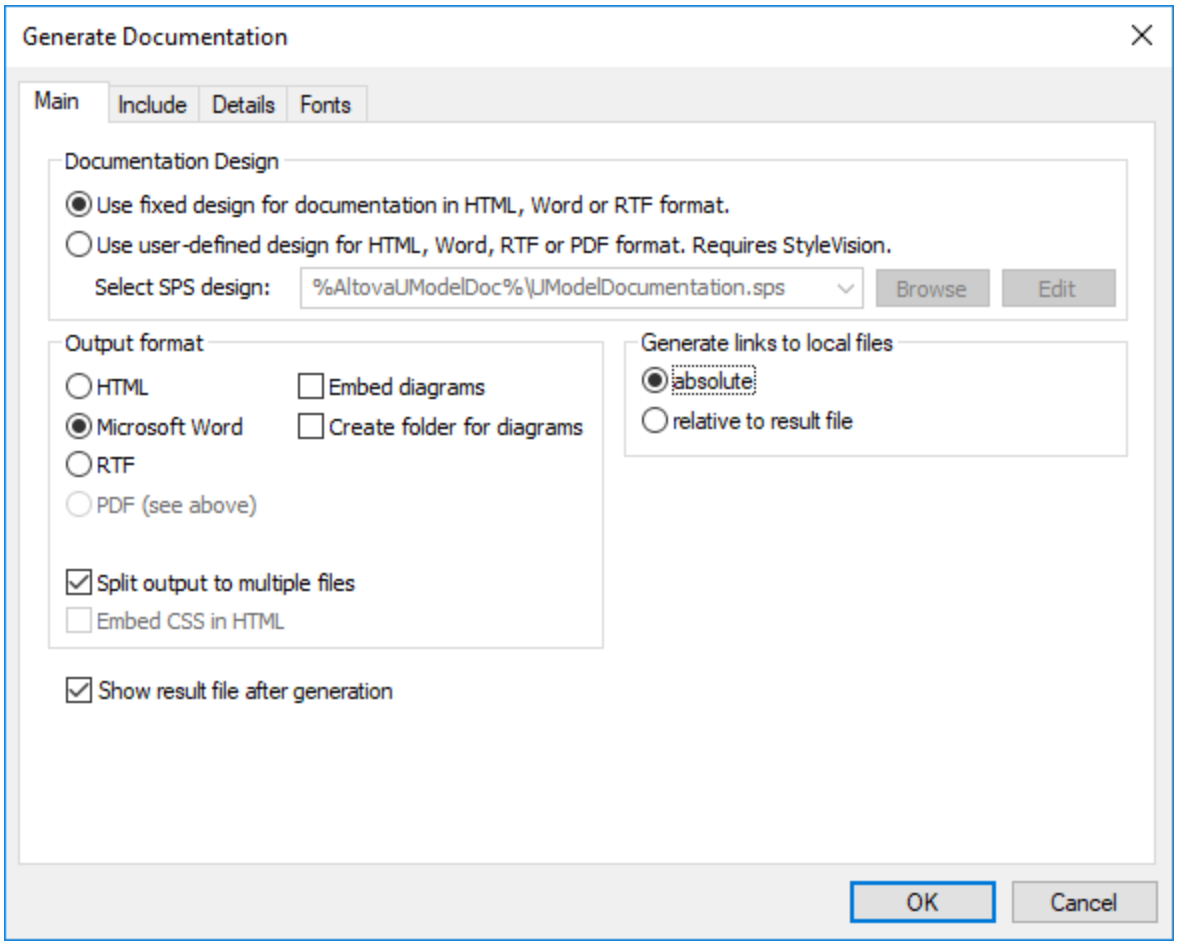
© 2019-2025 Altova GmbH

7.1 Documentation Generation Options

When generating documentation from UModel projects, you can set various options as described below. The options are organized by the tab in which they appear in the "Generate Documentation" dialog box.

Main tab

The **Main** tab includes the general documentation generation options.



Documentation Design:

- Select **Use fixed design...** to use the UModel built-in documentation design.
- Select **Use user-defined...** to generate documentation formatted with the help of a custom StyleVision Power Stylesheet (.sps file) created in StyleVision. Note: This option requires Altova StyleVision to be installed, see also Customizing Output with StyleVision 294.
- Click **Browse** to browse for a predefined stylesheet file.
- Click **Edit** to launch StyleVision and open the selected stylesheet file in a StyleVision window.

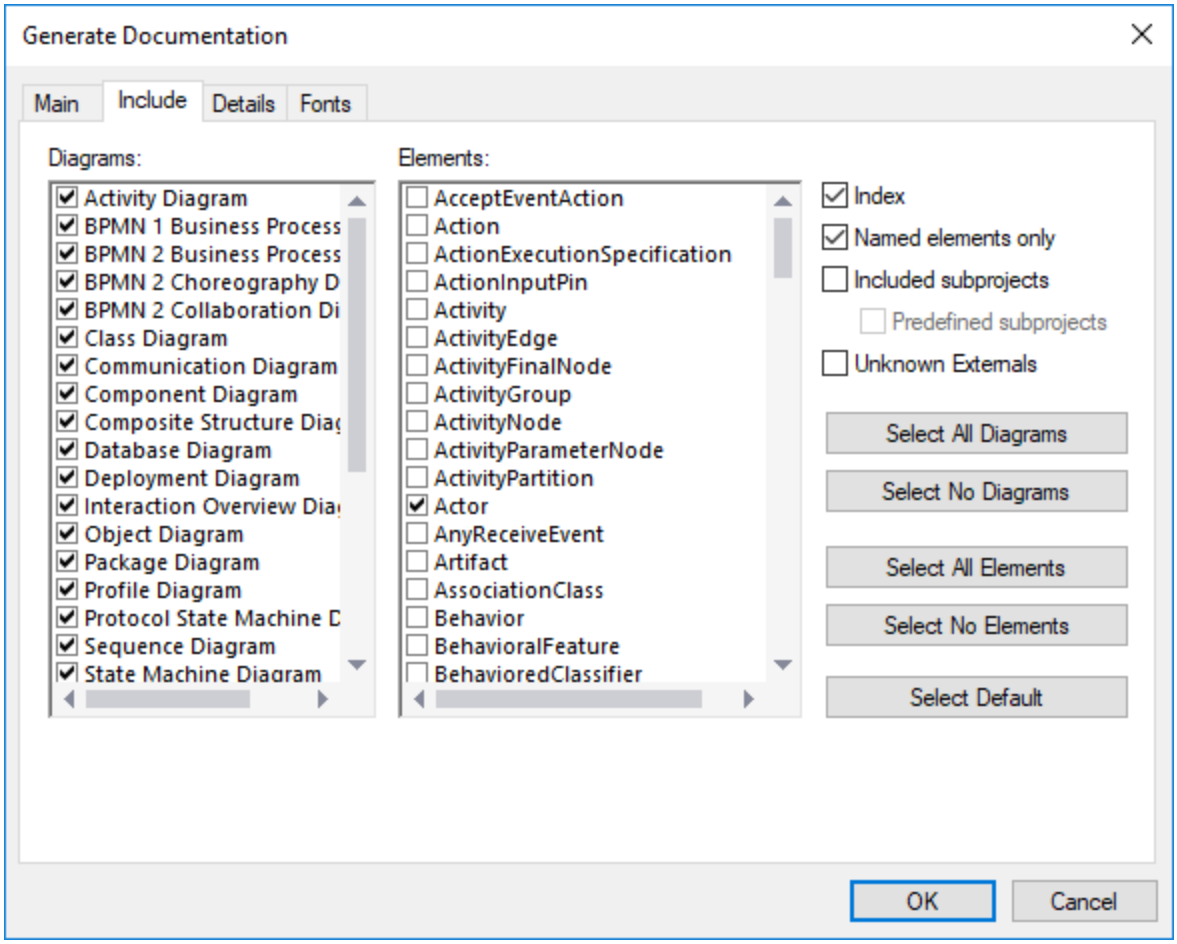
Output format:

- The output format can be one of the following: HTML, Microsoft Word, RTF, or PDF. Microsoft Word documents are created with the .doc file extension when generated using a fixed design, and with a .docx file extension when generated using a StyleVision Power Stylesheet. The PDF output format requires Altova StyleVision to be installed.
- **Split output to multiple files** generates an output file for each modeling element (class, interface, diagram, and so on). Clear this check box to generate one global file with all modeling elements.
- Select the **Embed CSS in HTML** check box to embed the generated CSS code in the HTML documentation. Clear this check box to keep the CSS file external.
- The **Embed diagrams** option is enabled for the Microsoft Word and RTF output options. When this check box is selected, diagrams are embedded in the generated file. Diagrams are created as .png files, which are displayed in the result file via object links.
- **Create folder for diagrams** generates a subfolder below the selected output folder, that will contain all diagrams.
- The **Show result file after generation** option is enabled for all output formats. When this check box is selected, the main generated file is displayed in the default browser (for HTML files), in Microsoft Word (for Word files), or in the default application (for .pdf or .rtf files).
- The **Generate links to local files** option allows you to specify if the generated links are to be absolute, or relative, to the output file.

Include tab

This tab allows you to select which diagrams and modeling elements are to appear in the documentation.

© 2019-2025 Altova GmbH

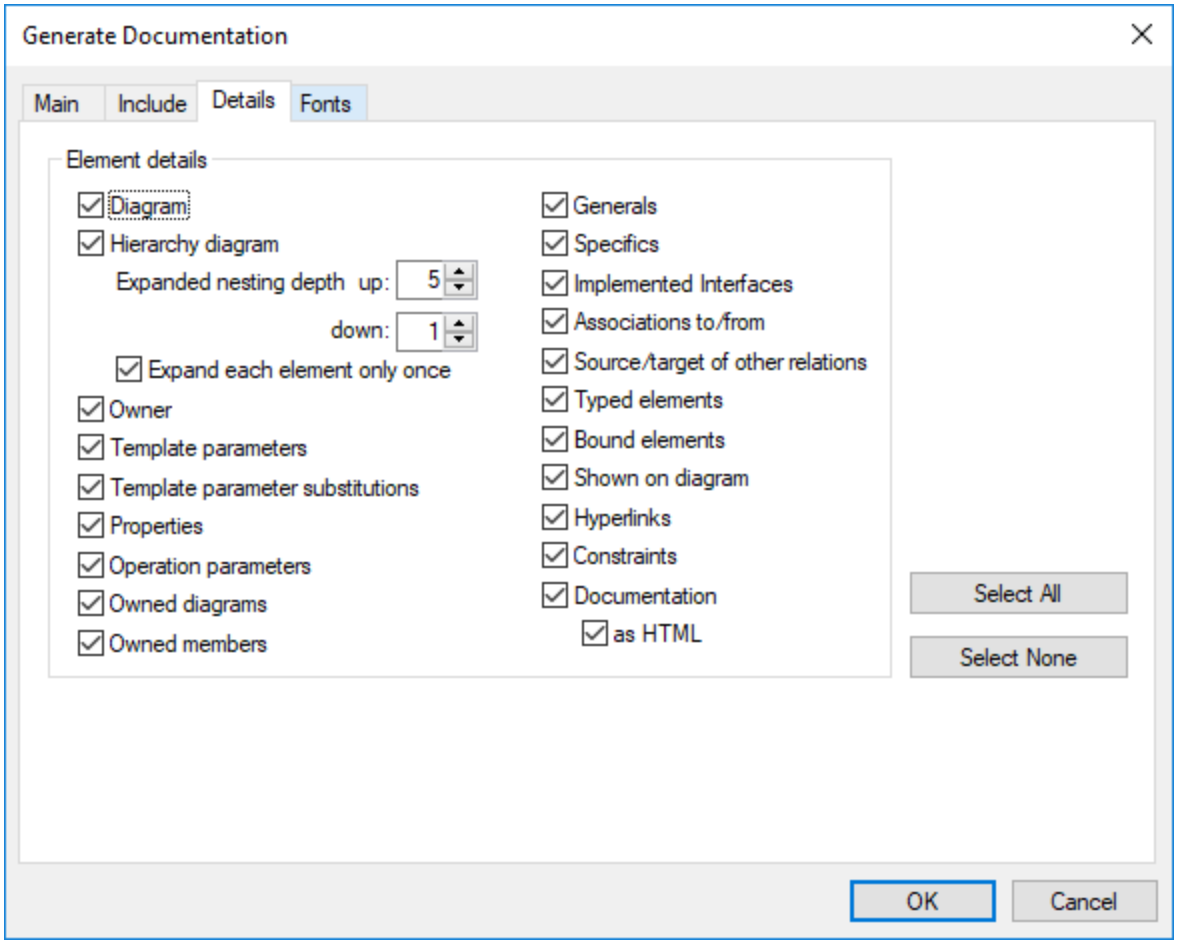


To prevent subprojects or profiles from being documented, clear the **Included subprojects** check box. Be aware that, if this check box is not selected, any elements or diagrams that are in subprojects will not be included in generated documentation. Select the **Predefined subprojects** check box to include UModel built-in profiles such as C# or Java profiles. Note, however, that generating documentation from predefined projects takes a very long time. **Unknown externals** refers to elements whose kind could not be identified—this usually happens after you import source code into UModel without first including the built-in subprojects for that language or language version, see Including Subprojects 164 for more information.

Details tab

This tab allows you to select the element details that are to appear in the documentation.

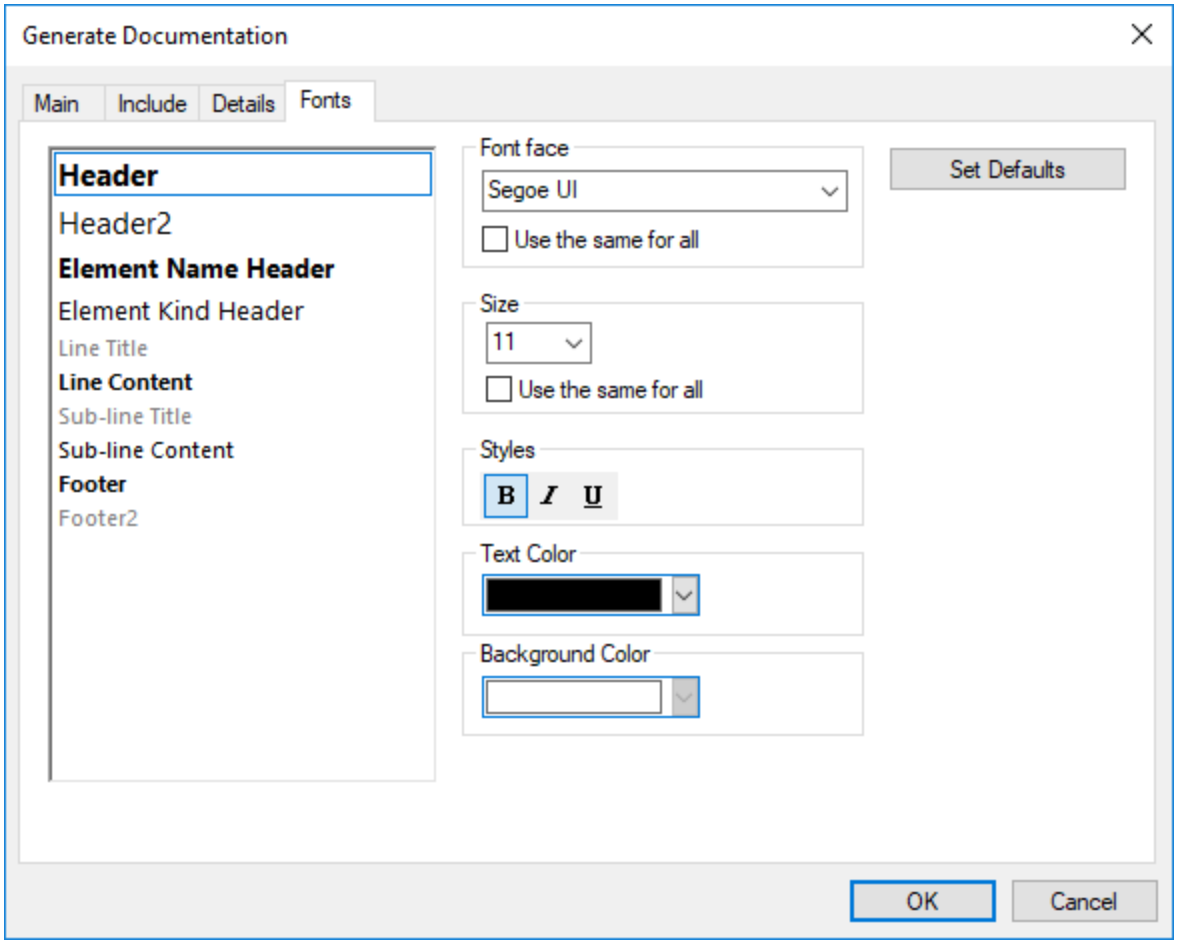
- If you intend to import XML tags text in your documentation, clear the **as HTML** option under the **Documentation** option.
- The **up** and **down** fields allow you to define the nesting depth shown above or below the current class in the hierarchy diagram.
- The **expand each element only once** option allows only one of the same classifiers to be expanded in the same image or diagram.



Fonts tab

This tab allows you to customize the font settings for the various headers and text content.

© 2019-2025 Altova GmbH

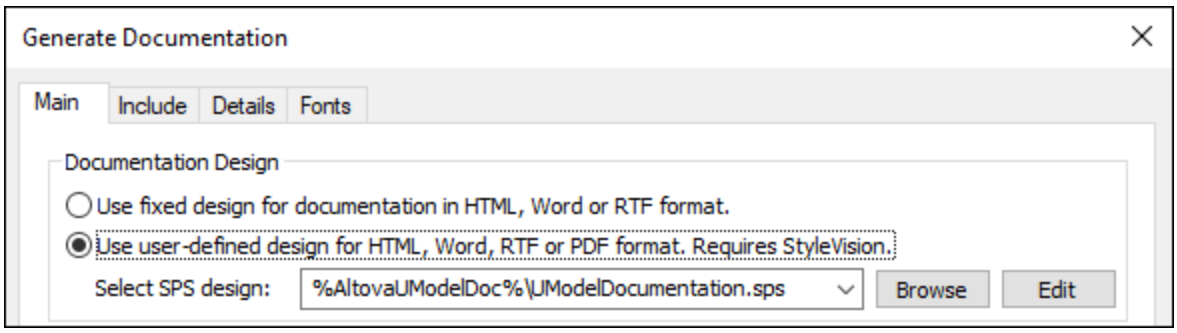

Generate Documentation
Main
Include
Details
Fonts
Header
Header2
Element Name Header
Element Kind Header
Line Title
Line Content
Sub-line Title
Sub-line Content
Footer
Footer2
Font face
Segoe UI
Use the same for all
Size
11
Use the same for all
Styles
B
I
U
Text Color
Background Color
Set Defaults
OK
Cancel

7.2 Customizing Output with StyleVision

You can customize the design of UModel-generated documentation with the help of StyleVision Power Stylesheet (.sps) files. Such files are created in Altova StyleVision (https://www.altova.com/stylevision). The advantage of using an .sps file is that you have complete control over the design of the documentation. In addition, PDF output is available if an .sps file is used.

> To generate documentation with .sps files, Altova StyleVision must be installed and licensed.

UModel includes a predefined .sps file, which is available at the following path: **C:\users\<username\Documents\UModel2025\Documentation\UModel\UModelDocumentation.sps**. To format the generated documentation using a custom .sps file, select this option while generating documentation, for example:

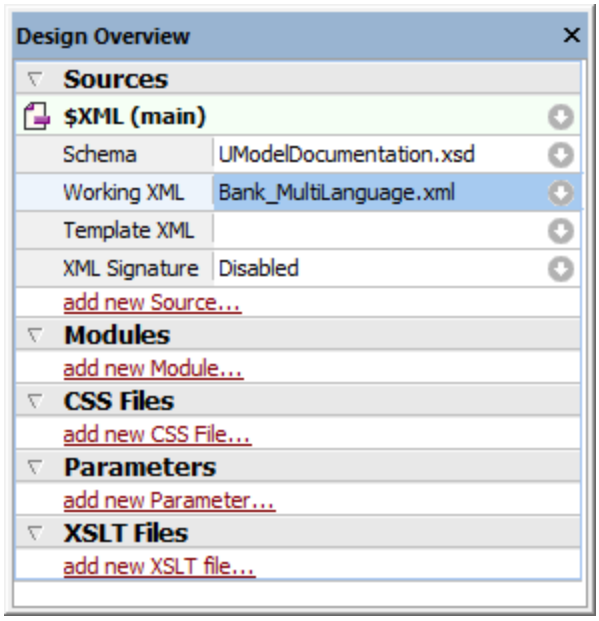


You can begin the customization by creating a copy of the default **UModelDocumentation.sps** and editing it in StyleVision. For example, you can change the existing formatting or add links and images to the design.

Any StyleVision Power Stylesheet is based on an XML Schema. In case of stylesheets that control the design of UModel-generated documentation, this schema is available at the following path: **C:\users\<username\Documents\UModel2025\Documentation\UModel\UModelDocumentation.xsd**. Note that the **UModelDocumentation.xsd** file references the **Documentation.xsd** file located in the folder above it.

When you author custom .sps files in StyleVision for UModel documentation, the **UModelDocumentation.xsd** file must be used as a schema. The image below illustrates the Design Overview window of StyleVison after you open the **UModelDocumentation.sps** file. Notice that it uses the **UModelDocumentation.xsd** schema file, and a working XML required to preview the design. The working XML file is available in the **SampleData** subfolder relative to the schema file.

© 2019-2025 Altova GmbH



For instructions about how to edit .sps files, refer to the StyleVision documentation (https://www.altova.com/documentation).

8 UML Diagrams

Altova website: UML diagrams

There are two major groups of UML diagrams, Structural diagrams, which show the static view of the model, and Behavioral diagrams, which show the dynamic view. UModel supports all fourteen diagrams of the UML 2.5 specification, as well as Additional diagrams.

- Behavioral diagrams 297 include Activity, State machine, Protocol State Machine and Use Case diagrams; as well as the Interaction, Communication, Interaction Overview, Sequence, and Timing diagrams.
- Structural diagrams 387 include: Class, Composite Structure, Component, Deployment, Object, and Package diagrams.
- Additional diagrams 424 XML schema diagrams, Business Processing Modeling Notation (BPMN), SysML diagrams, Database diagrams.

Note: The **Ctrl+Enter** keys can be used to create multi-line labels for most of the modeling diagrams, e.g. Lifeline labels in sequence diagrams, timing diagrams; guard conditions, state names, activity names etc.

© 2019-2025 Altova GmbH

8.1 Behavioral Diagrams

These diagrams depict behavioral features of a system or business process, and include a subset of diagrams which emphasize object interactions.

- Activity Diagram
- State Machine Diagram
- Protocol State Machine Diagram
- Use Case Diagram 342

A subset of the Behavioral diagrams are those that depict the object interactions, namely:

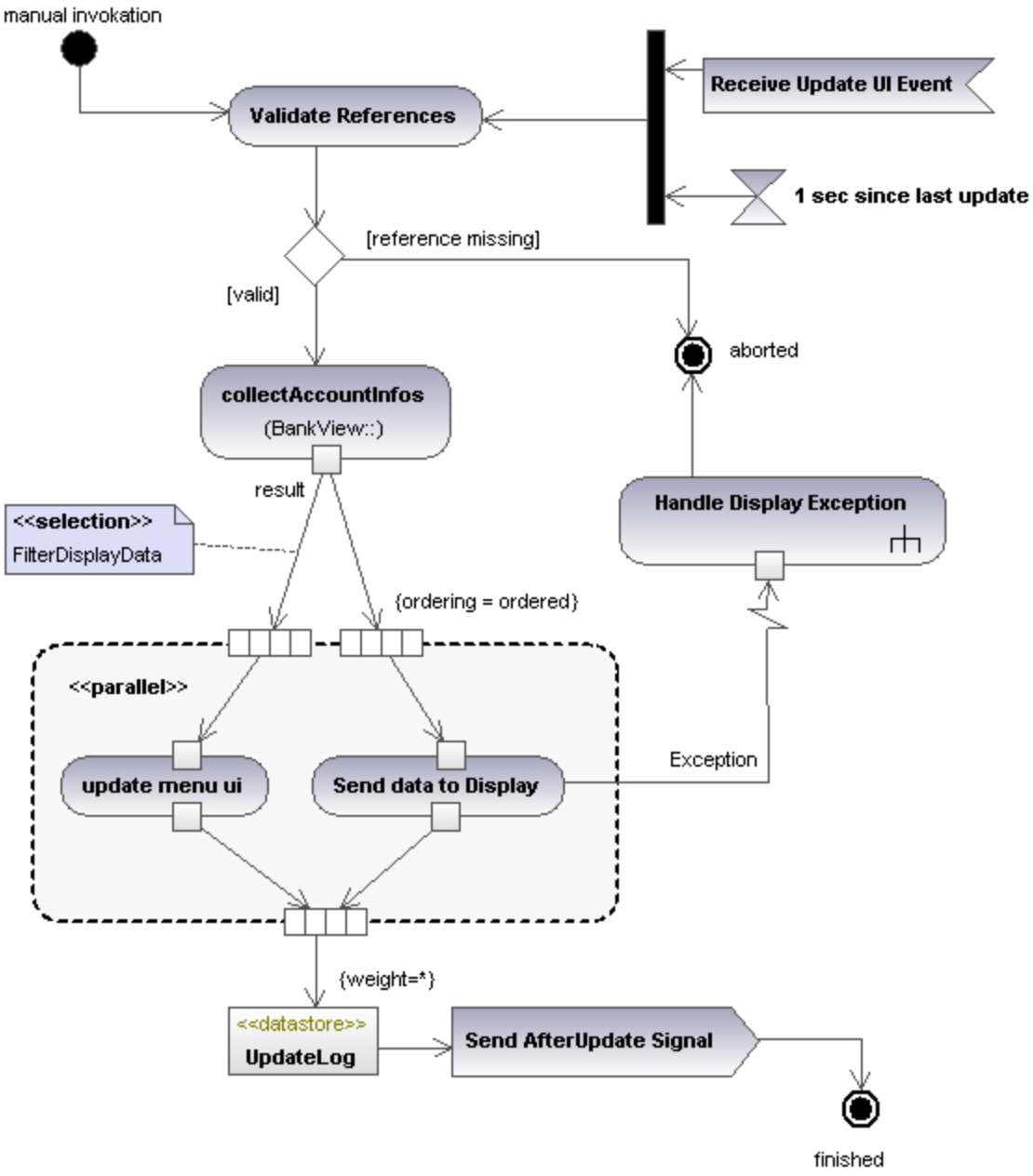
- Communication Diagram
- Interaction Overview Diagram
- Sequence Diagram
- Timing Diagram 378

8.1.1 Activity Diagram

Altova website: UML Activity diagrams

Activity diagrams are useful for modeling real-world workflows of business processes, and display which actions need to take place and what the behavioral dependencies are. The Activity diagram describes the specific sequencing of activities and supports both conditional and parallel processing. The Activity diagram is a variant of the State diagram, with the states being activities.

The Activity diagram shown below is available in the **Bank_MultiLanguage.ump** sample, in the **...\UModelExamples** folder supplied with UModel.

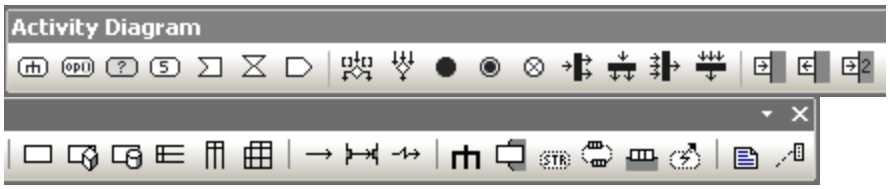


8.1.1.1 Inserting Activity Diagram elements

To add elements to the diagram:

1. Click the element's toolbar button in the Activity Diagram toolbar.

© 2019-2025 Altova GmbH



2. Click in the Activity Diagram to insert the element.

To insert multiple elements of the selected type, hold down the **Ctrl** key and click in the diagram window.

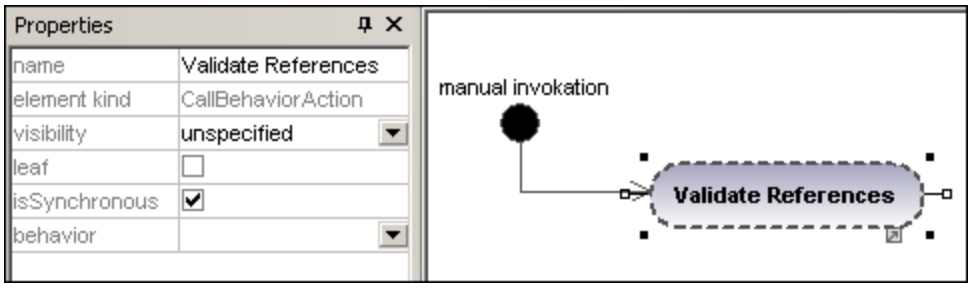
Dragging existing elements into the activity diagram

Most elements occurring in other activity diagrams can be inserted into an existing activity diagram.

1. Locate the element you want to insert in the Model Tree Window[83] (you can use the search function text box, or press **Ctrl+F** to search for any element).
2. Drag the element(s) into the activity diagram.

Inserting an action (CallBehavior)

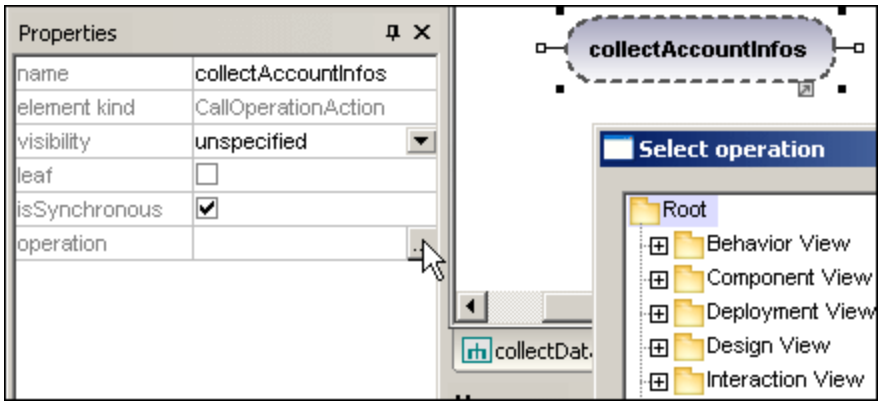
1. Click the **Action (CallBehavior)** toolbar button, and click in the Activity diagram to insert it.
2. Enter the name of the Action, e.g. "Validate References", and press **Enter** to confirm.



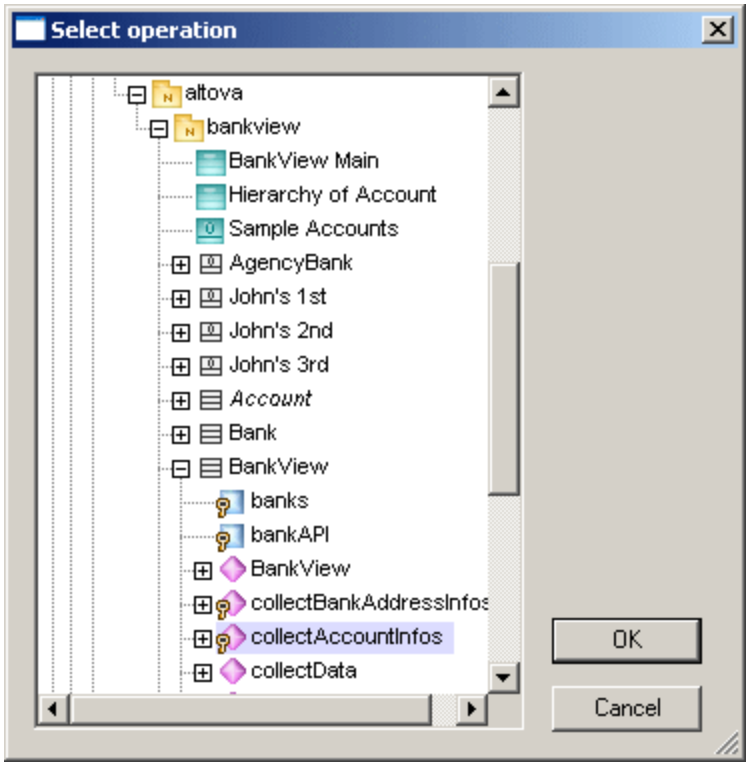
Note: Use **Ctrl+Enter** to create a multi-line name.

Inserting an action (CallOperation) and selecting a specific operation

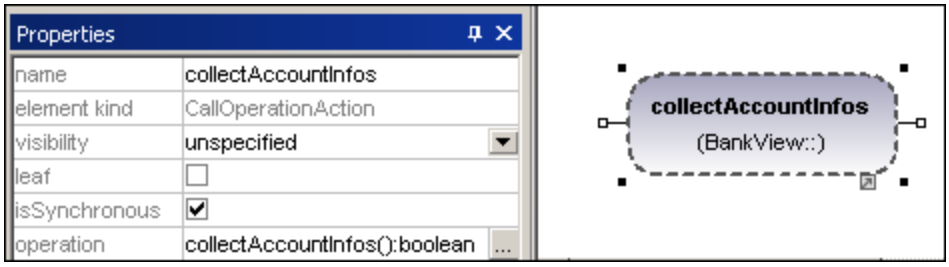
1. Click the **Action (CallOperation)** icon in the icon bar, and click in the Activity diagram to insert it.
2. Enter the name of the Action, e.g. "collectAccountInfo", and press **Enter** to confirm.
3. Click the **Browse** button to the right of the operation field in the **Properties** tab. This opens the "Select Operation" dialog box in which you can select the specific operation.



4. Navigate to the specific operation that you want to insert, and click **OK** to confirm.



In this example, the operation "collectAccountInfos" is in the `BankView` class.



© 2019-2025 Altova GmbH

8.1.1.2 Creating branches and merges

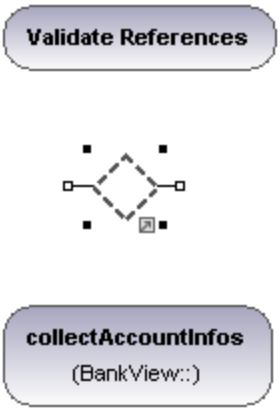
A branch has a single incoming flow and multiple outgoing guarded flows. Only one of the outgoing flows can be traversed, so the guards should be mutually exclusive.

In this example the (BankView) references are to be validated:

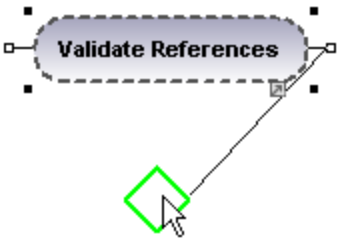
- branch1 has the guard "reference missing", which transitions to the `abort` activity
- branch2 has the guard "valid", which transitions to the `collectAccountInfos` activity.

Creating a branch (alternate flow)

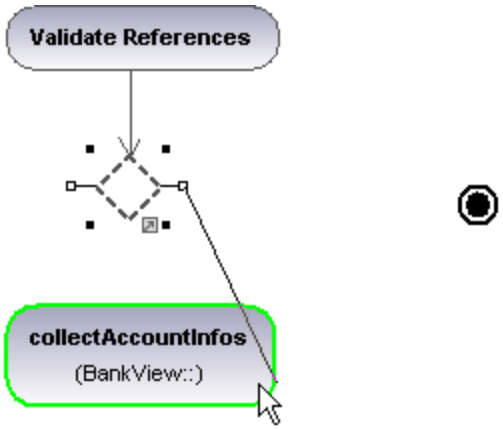
1. Click the **DecisionNode** icon in the title bar, and insert it in the Activity diagram.



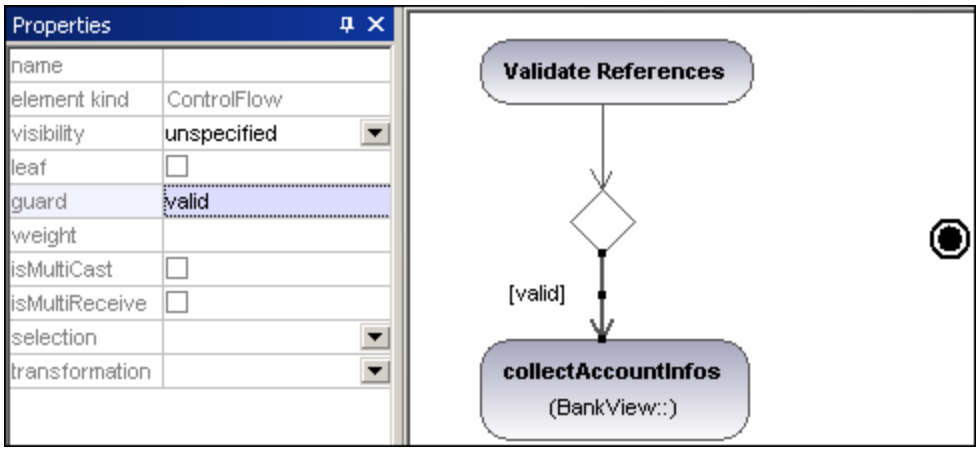
2. Click the **ActivityFinalNode** icon which represents the abort activity, and insert it into the Activity diagram.
3. Click the "Validate References" activity to select it, then click the right-hand handle, **ControlFlow**, and drag the resulting connector onto the "DecisionNode" element. The element is highlighted when you can drop the connector.



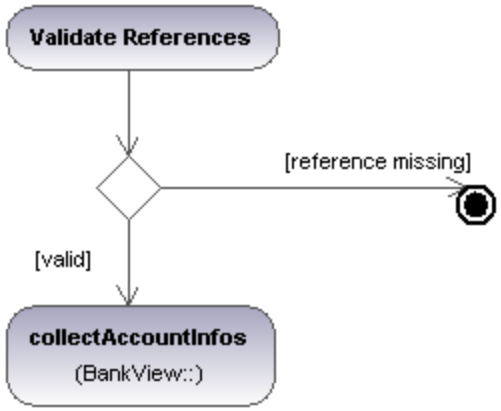
4. Click the "DecisionNode" element, click the right-hand connector, **ControlFlow**, and drop it on the "collectAccountInfos" action. Please see "Inserting an Action (CallOperation 299" for more information.



5. Enter the guard condition "valid", in the guard field of the **Properties** tab.



6. Click the DecisionNode element and drag from the right-hand handle, **ControlFlow**, and drop it on the "ActivityFinalNode" element. The guard condition on this transition is automatically defined as "else". Double click the guard condition in the diagram to change it e.g. "reference missing".

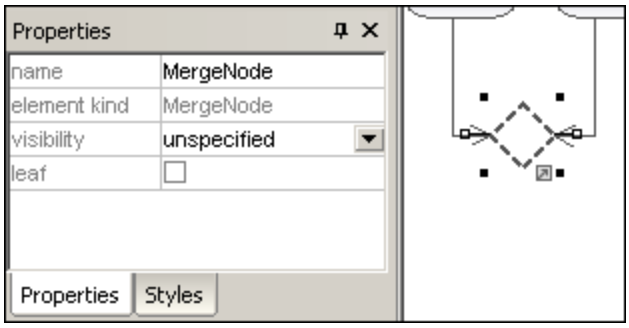


Note: UModel does not validate, or check, the number of Control/Object Flows in a diagram.

© 2019-2025 Altova GmbH

Creating a merge

1. Click the **MergeNode** icon in the icon bar, then click in the Activity diagram to insert it.

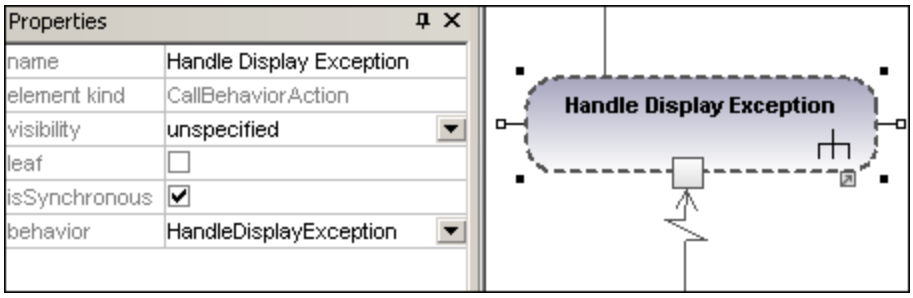


2. Click the ControlFlow (ObjectFlow) handles of the actions that are to be merged, and drop the arrow(s) on the "MergeNode" symbol.

8.1.1.3 Activity Diagram elements

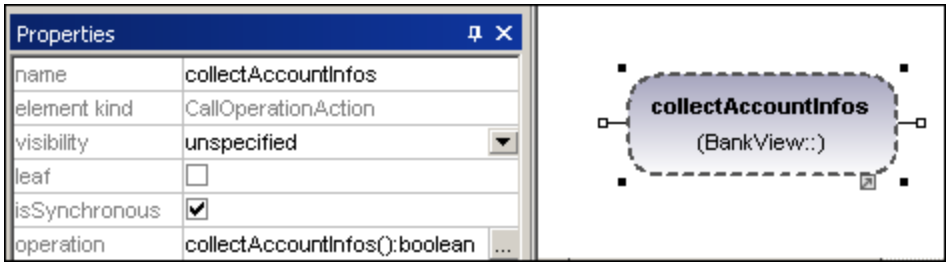
Action (CallBehavior)

Inserts a **CallBehaviorAction** element which directly invokes a specific behavior. Selecting an existing behavior using the **behavior** combo box, e.g. HandleDisplayException, displays a rake symbol within the element.



Action (CallOperation)

Inserts a **CallOperationAction** which indirectly invokes a specific behavior as a method. Please see "Inserting an action (CallOperation) 299" for more information.



Action (OpaqueAction)

A type of action used to specify implementation information. Can be used as a placeholder until you decide which specific action type you want to use.

Action (ValueSpecificationAction)

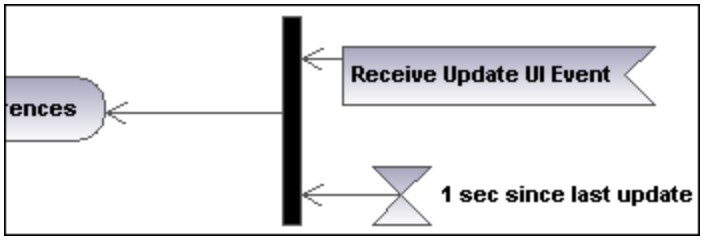
A type of action that evaluates(/generates) a specific value at the output pin. (Defined by the specific properties, e.g. upperBound.)

AcceptEventAction

Inserts the Accept Event action which waits for the occurrence of an event which meets specific conditions.

AcceptEventAction (TimeEvent)

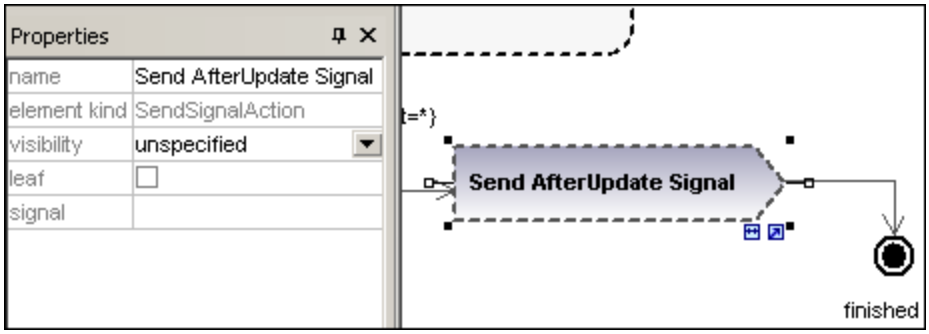
Inserts an **AcceptEventAction**, triggered by a time event, which specifies an instant of time by an expression e.g. 1 sec. since last update.



SendSignalAction

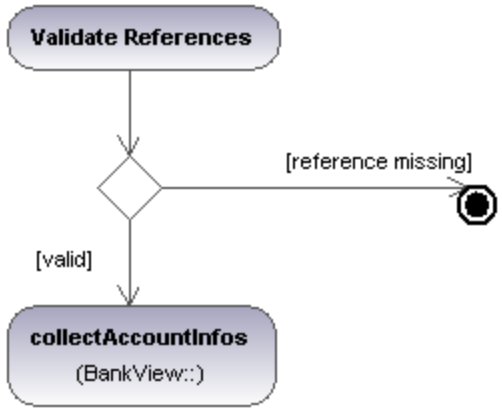
Inserts the **SendSignalAction**, which creates a signal from its inputs and transmits the signal to the target object, where it may cause the execution of an activity.

© 2019-2025 Altova GmbH



DecisionNode

Inserts a Decision Node which has a single incoming transition and multiple outgoing guarded transitions. Please see "Creating a branch 301" for more information.



MergeNode

Inserts a Merge Node which merges multiple alternate transitions defined by the Decision Node. The Merge Node does not synchronize concurrent processes, but selects one of the processes.

InitialNode

The beginning of the activity process. An activity can have more than one initial node.

ActivityFinalNode

The end of the activity process. An activity can have more that one final node, all flows in the activity stop when the "first" final node is encountered.

FlowFinalNode

Inserts the Flow Final Node, which terminates a flow. The termination does not affect any other flows in the activity.

ForkNode

Inserts a vertical Fork node. Used to divide flows into multiple concurrent flows.

ForkNode (Horizontal)

Inserts a horizontal Fork node. Used to divide flows into multiple concurrent flows.

JoinNode

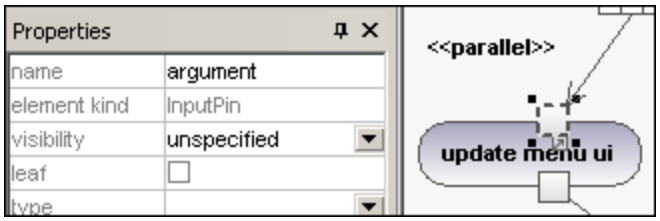
Inserts a vertical Fork node. A Join node synchronizes multiple flows defined by the Fork node.

Join Node (horizontal)

Inserts a horizontal Fork node. A Join node synchronizes multiple flows defined by the Fork node.

InputPin

Inserts an input pin onto a Call Behavior, or Call Operation action. Input pins supply input values that are used by an action. A default name, "argument", is automatically assigned to an input pin.

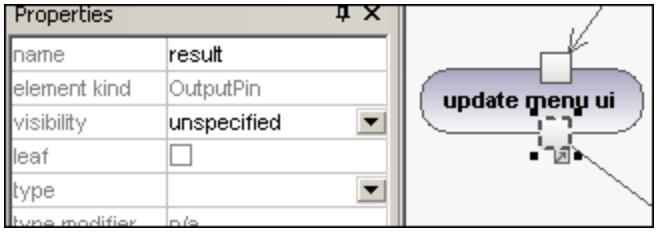


The input pin symbol can only be placed onto those activity elements where the mouse pointer changes to the hand symbol . Dragging the symbol repositions it on the element border.

OutputPin

Inserts an output pin action. Output pins contain output values produced by an action. A name corresponding to the UML property of that action e.g. result, is automatically assigned to the output pin.

© 2019-2025 Altova GmbH



The output pin symbol can only be placed onto those activity elements where the mouse pointer changes to the hand symbol. Dragging the symbol repositions it on the element border.

Exception Pin

An OutputPin can be changed to an Exception pin by clicking the pin and selecting "isExceptionPin" from the Properties pane.

ValuePin

Inserts a Value Pin which is an input pin that provides a value to an action, that does not come from an incoming object flow. It is displayed as an input pin symbol, and has the same properties as an input pin.

ObjectNode

Inserts an object node which is an abstract activity node that defines object flow in an activity. Object nodes can only contain values at runtime that conform to the type of the object node.

CentralBufferNode

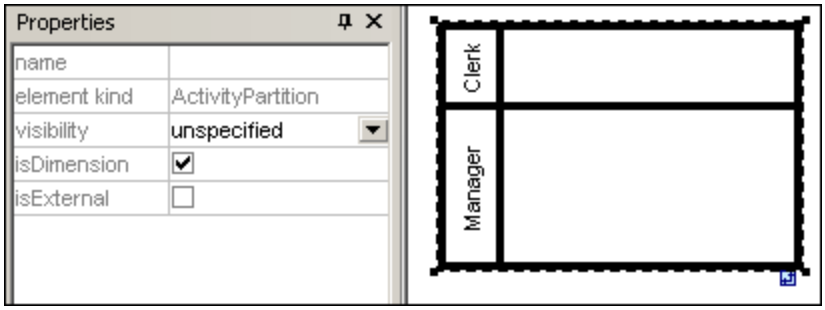
Inserts a Central Buffer Node which acts as a buffer for multiple in- and out flows from other object nodes.

DataStoreNode

Inserts a Data Store Node which is a special "Central Buffer Node" used to store persistent (i.e. non transient) data.

ActivityPartition (horizontal)

Inserts a horizontal Activity Partition, which is a type of activity group used to identify actions that have some characteristic in common. This often corresponds to organizational units in a business model.



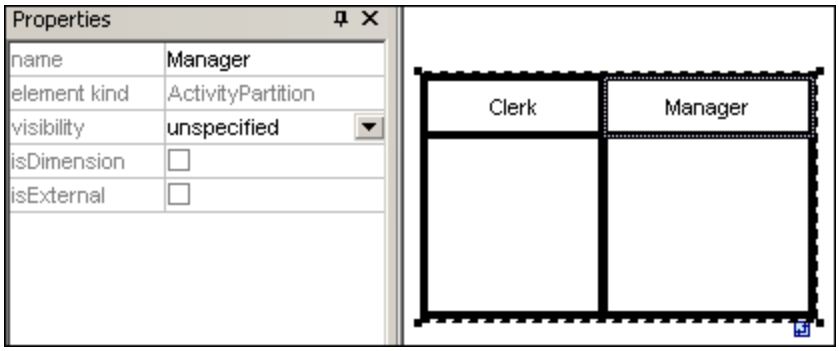
Double clicking a label allows you to edit it directly; pressing Enter orients the text correctly.

Please note that Activity Partitions are the UML 2.0 update to the "swimlane" functionality of previous UML versions.

- Elements placed within a ActivityPartition become part of it when the boundary is highlighted.
- Objects within an ActivityPartition can be individually selected using **Ctrl+Click**, or by dragging the marquee inside the boundary.
- Click the ActivityPartition boundary, or title, and drag to reposition it.

ActivityPartition (vertical)

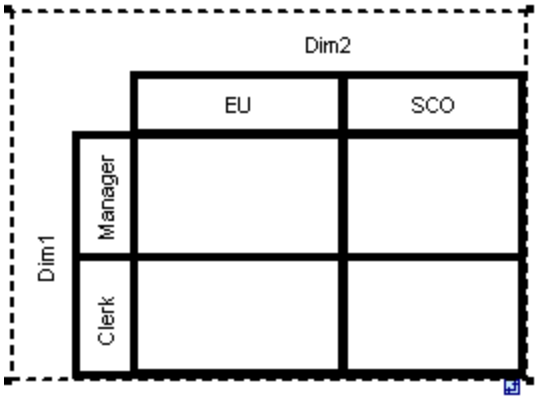
Inserts a vertical Activity Partition, which is a type of activity group used to identify actions that have some characteristic in common. This often corresponds to organizational units in a business model.



ActivityPartition (2 Dimensional)

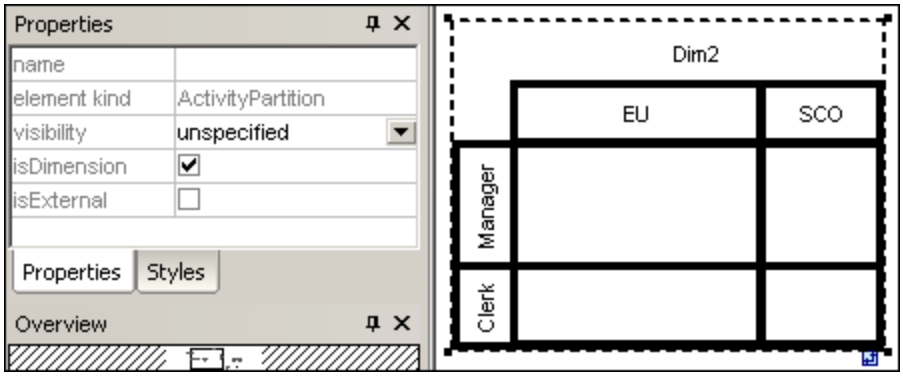
Inserts a two dimensional Activity Partition, which is a type of activity group used to identify actions that have some characteristic in common. Both axes have editable labels.

© 2019-2025 Altova GmbH



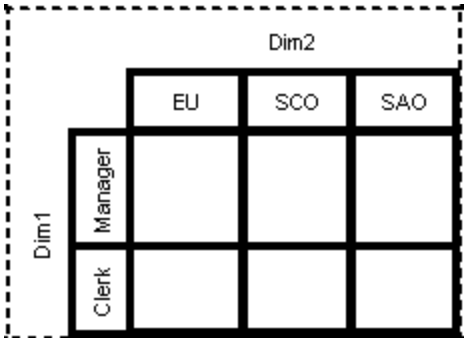
To remove the Dim1, Dim2 dimension labels:

1. Click the dimension label you want to remove e.g. Dim1
2. Double click in the Dim1 entry in the Properties tab, delete the Dim1 entry, and press Enter to confirm.



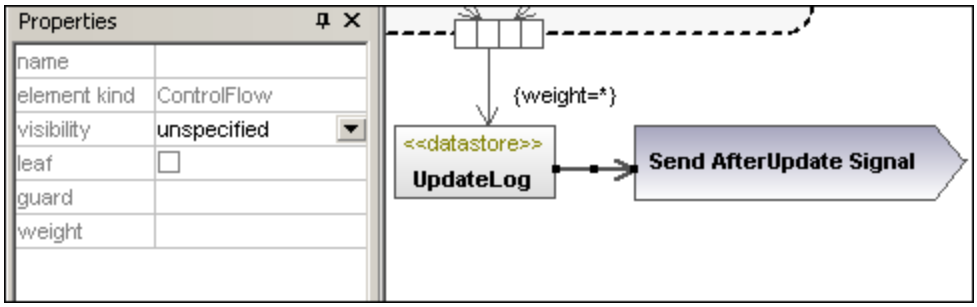
Note that Activity Partitions can be nested:

1. Right click the label where you want to insert a new partition.
2. Select **New | ActivityPartition**.



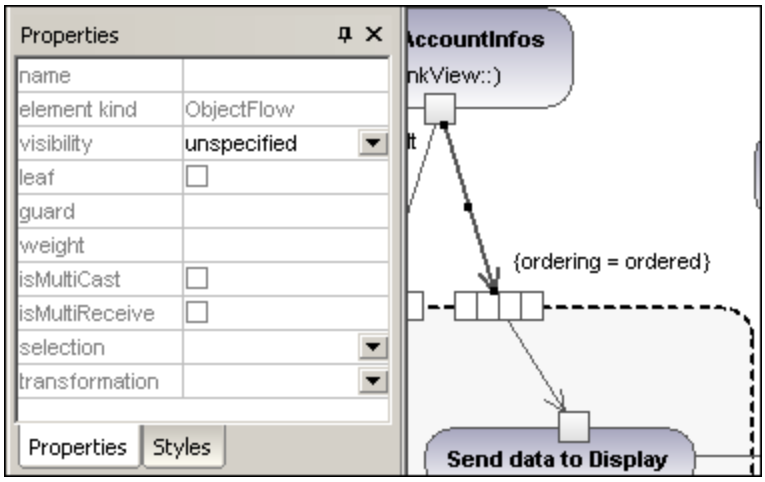
ControlFlow

A Control Flow is an edge, i.e. an arrowed line, that connects two activities/behaviours, and starts an activity after the previous one has been completed.



ObjectFlow

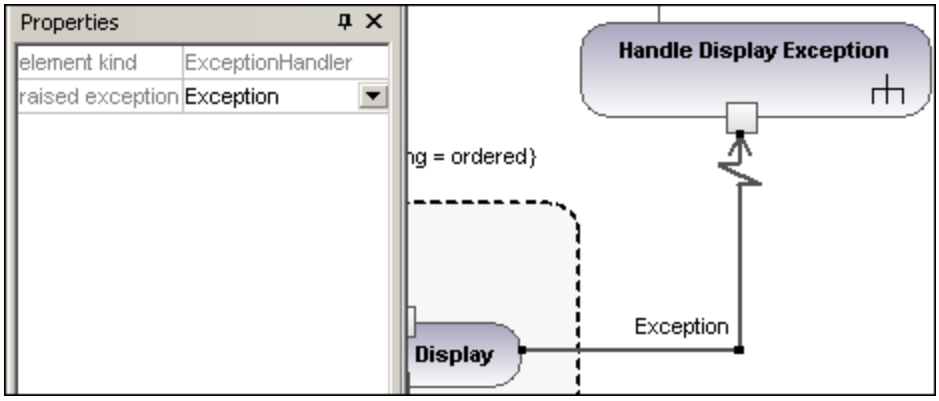
A Object Flow is an edge, i.e. an arrowed line, that connects two actions/object nodes, and starts an activity after the previous one has been completed. Objects or data can be passed along an Object Flow.



ExceptionHandler

An Exception Handler is an element that specifies what action is to be executed if a specified exception occurs during the execution of the protected node.

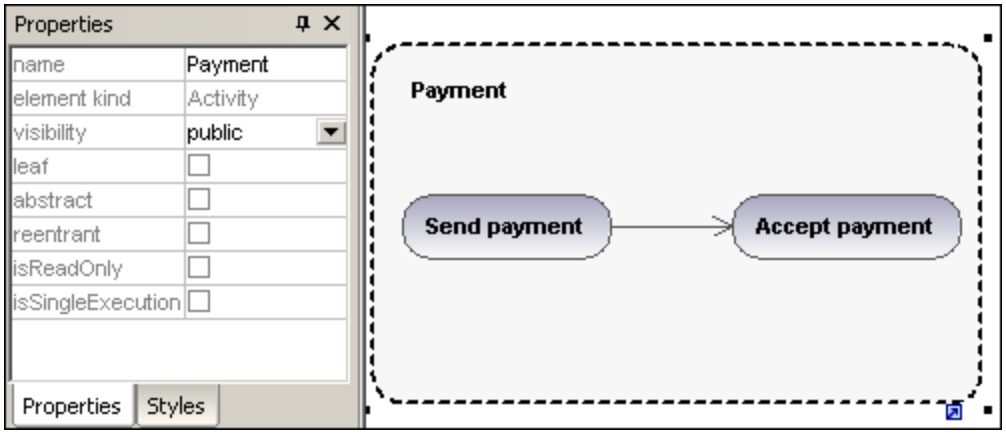
© 2019-2025 Altova GmbH



An Exception Handler can only be dropped on an Input Pin of an Action.

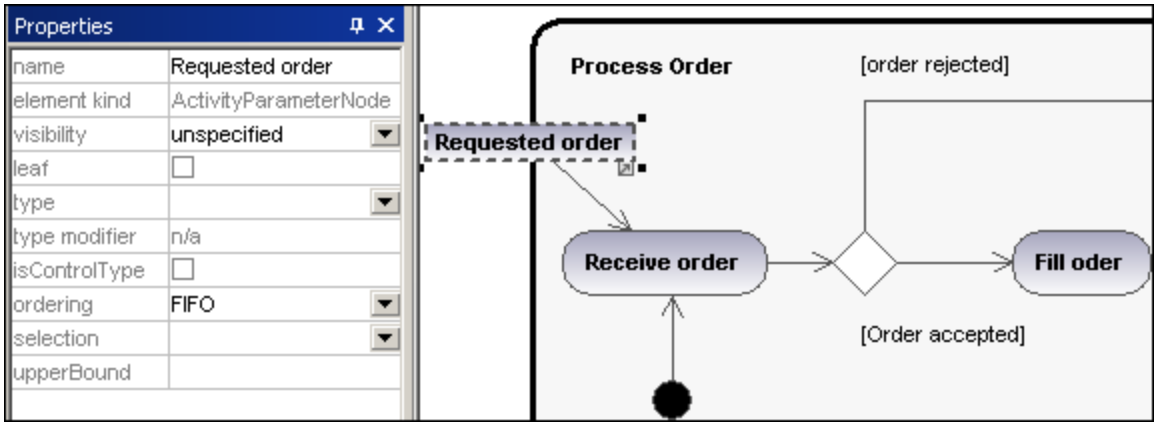
Activity

Inserts an Activity into the activity diagram.



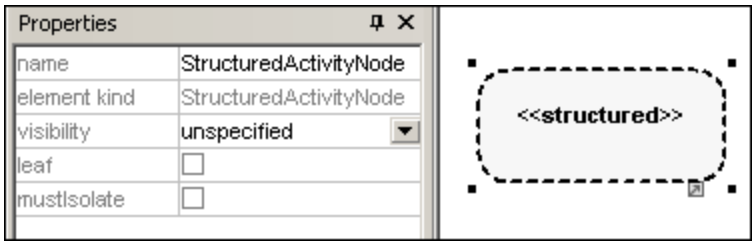
ActivityParameterNode

Inserts an Activity Parameter node onto an activity. Clicking anywhere in the activity places the parameter node on the activity boundary.



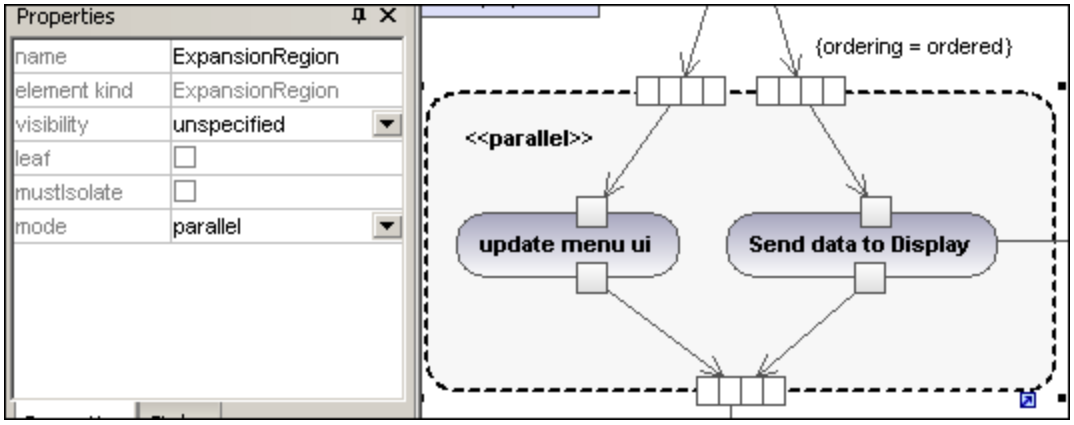
StructuredActivityNode

Inserts a Structured Activity Node which is a structured part of the activity, that is not shared with any other structured node.



ExpansionRegion

An expansion region is a region of an activity having explicit input and outputs (using ExpansionNodes). Each input is a collection of values.

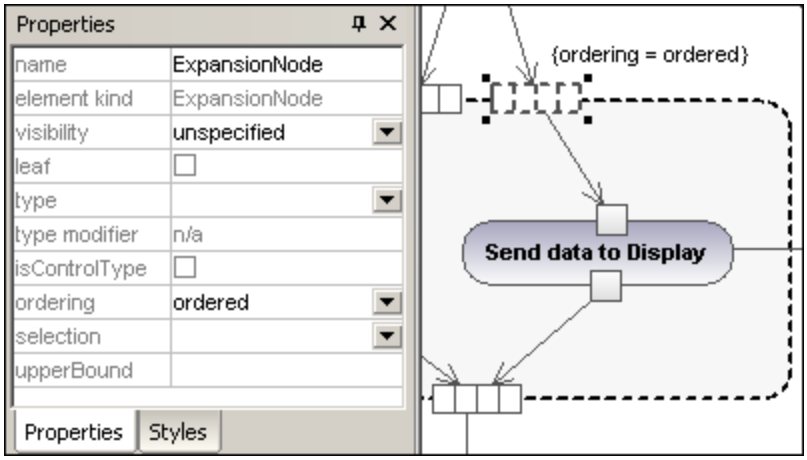


© 2019-2025 Altova GmbH

The expansion region mode is displayed as a keyword, and can be changed by clicking the "mode" combo box in the Properties tab. Available settings are:parallel, iterative, or stream.

ExpansionNode

Inserts an Expansion Node onto an Expansion Region. Expansion nodes are input and output nodes for the Expansion Region, where each input/output is a collection of values. The arrows into, or out of, the expansion region, determine the specific type of expansion node.

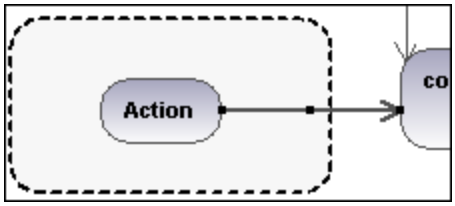


InterruptableActivityRegion

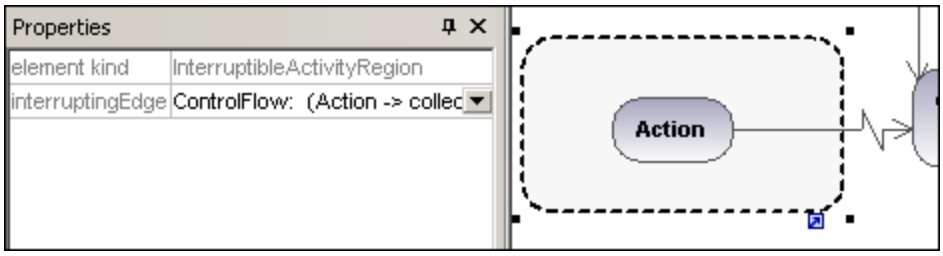
An interruptible region contains activity nodes. When a control flow leaves an interruptible region all flows and behaviors in the region are terminated.

To add an interrupting edge:

1. Make sure that an Action element is present in the InterruptableActivityRegion, as well as an outgoing Control Flow to another action:



2. Right click the Control Flow arrow, and select **New | InterruptingEdge**.



Note: You can also add an InterrupingEdge by clicking the InterruptableActivityRegion, right clicking in the Properties window, and selecting Add InterruptingEdge from the pop-up menu.

8.1.2 State Machine Diagram

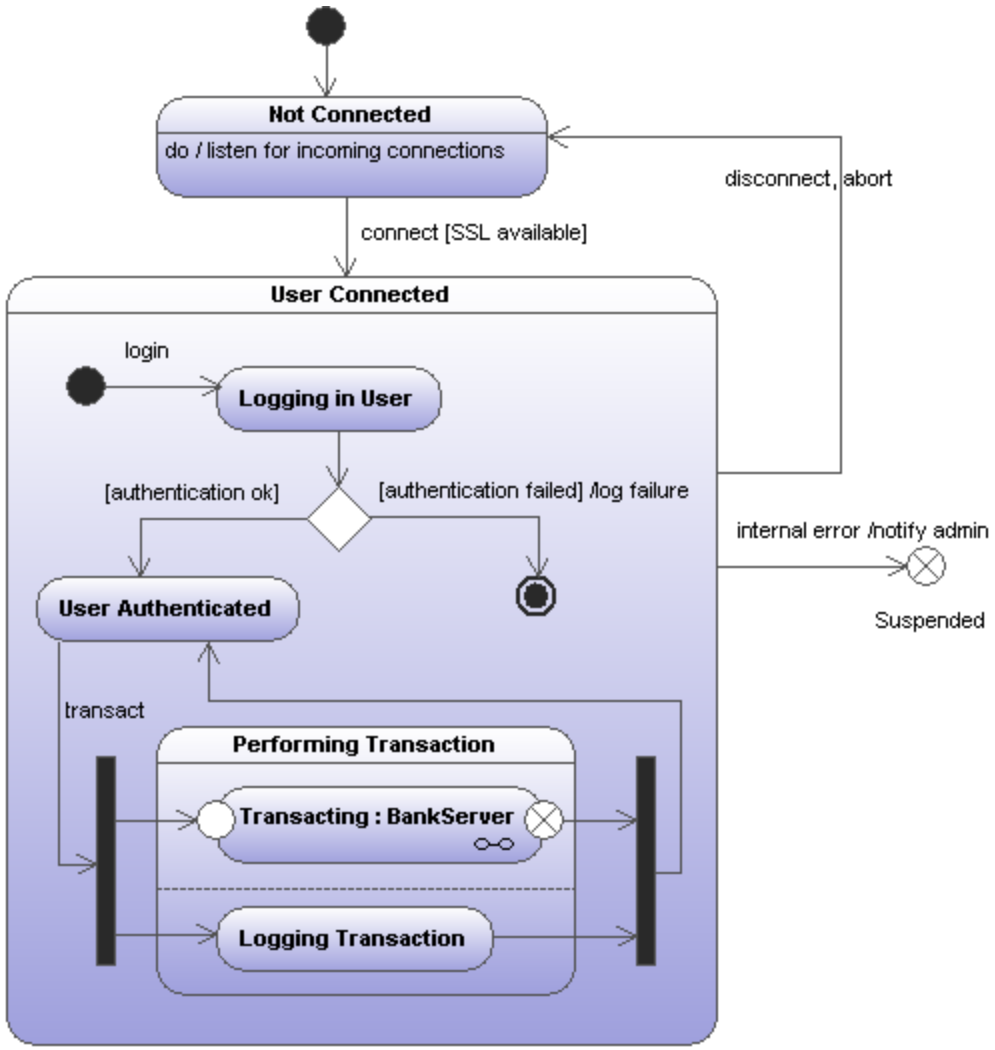
The State Machine Diagram models the behavior of a system by describing the various states an object may be in, and the transitions between those states. They are generally used to describe the behavior of an object spanning several use cases.

Two types of processes can achieve this:

1. **Actions,** which are associated to **transitions,** are short-term processes that cannot be interrupted (for example, **internal error /notify admin** in the diagram below)
2. State **Activities** (behaviors), which are associated to **states,** are longer-term processes that may be interrupted by other events (for example, **listen for incoming connections**, in the diagram below).

A state machine can have any number of State Machine Diagrams (or State Diagrams) in UModel.

© 2019-2025 Altova GmbH



Sample State Machine diagram

The State machine diagram illustrated above is available in the following sample UModel project: **C:\Users\<username>\Documents\Altova\UModel2025\UModelExamples\Bank_MultiLanguage.ump**.

8.1.2.1 Inserting state machine diagram elements

To insert state machine diagram elements:

1. Click the specific state machine diagram icon in the State Machine Diagram toolbar.

2. Click in the State Diagram to insert the element. To insert multiple elements of the selected type, hold down the **Ctrl** key and click in the diagram window.

Dragging existing elements into the state machine diagram

Most elements occurring in other state machine diagrams can be inserted into an existing state machine.

1. Locate the element you want to insert in the **Model Tree** tab (you can use the search function text box, or press **Ctrl+F** to search for any element).
2. Drag the element(s) into the state diagram.

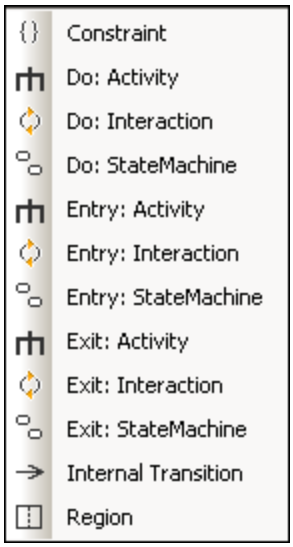
8.1.2.2 Creating states, activities and transitions

To add a simple state:

1. Click the **State** toolbar icon ( ), and then click inside the diagram.
2. Enter the name of the state and press **Enter** to confirm.

To add an activity to a state:

- Right-click the state element, select **New**, and then one of the entries from the context menu.



The **Entry**, **Exit**, and **Do** activities are associated with one of the following possible behaviors: "Activity", "Interaction", and "StateMachine". Therefore, the options available in the context menu are:

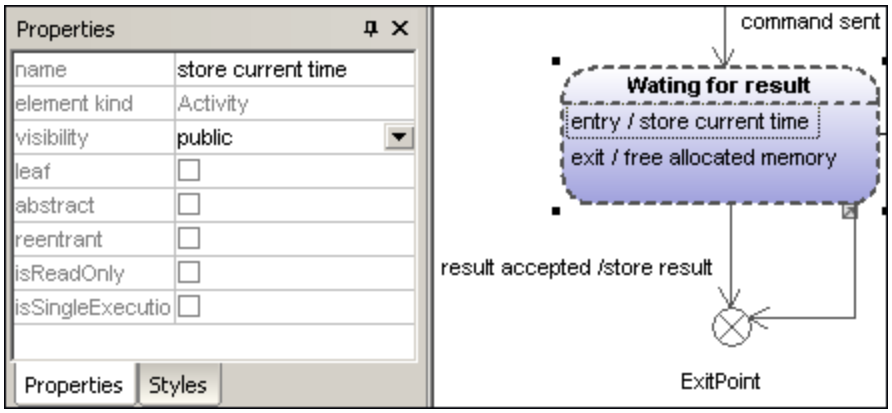
- Do: Activity
- Do: Interaction
- Do: StateMachine
- Entry: Activity
- Entry: Interaction

© 2019-2025 Altova GmbH

- Entry: StateMachine
- Exit: Activity
- Exit: Interaction
- Exit: StateMachine

These options originate in the UML specification. Namely, each of these internal actions are behaviors, and, in the UML specification, three classes derive from the "Behavior" class: Activity, StateMachine, and Interaction. In the generated code, it does not make a difference which particular behavior (Activity, StateMachine, or Interaction) has been selected.

You can select one action from the **Do**, **Entry** and **Exit** action categories. Activities are placed in their own compartment in the state element, though not in a separate region. The type of activity that you select is used as a prefix for the activity e.g. **entry / store current time**.

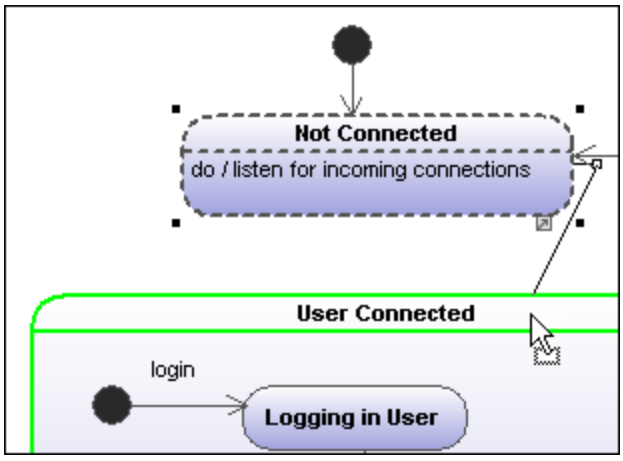


To delete an activity:

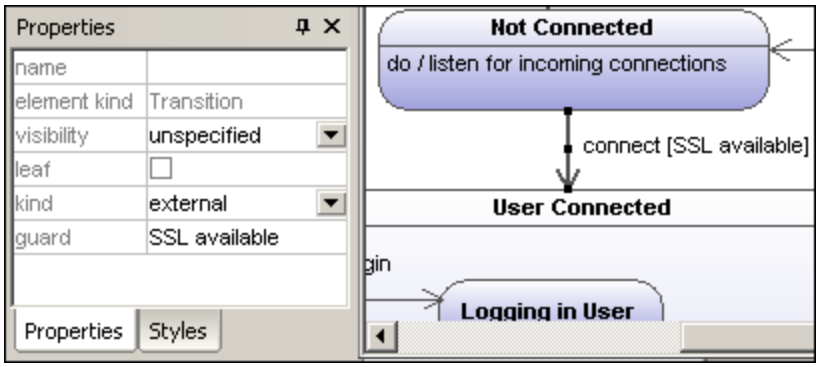
- Click the respective activity in the state element and press the **Del** key.

To create a transition between two states:

1. Click the Transition handle of the source state (on the right of the element).
2. Drag-and-drop the transition arrow onto the target state.



The Transition properties are now visible in the **Properties** tab. Clicking the "kind" combo box, allows you to define the transition type: external, internal or local.



Transitions can have an event trigger, a guard condition and an action in the form **eventTrigger [guard condition] /activity**.

To automatically create operations from transitions:

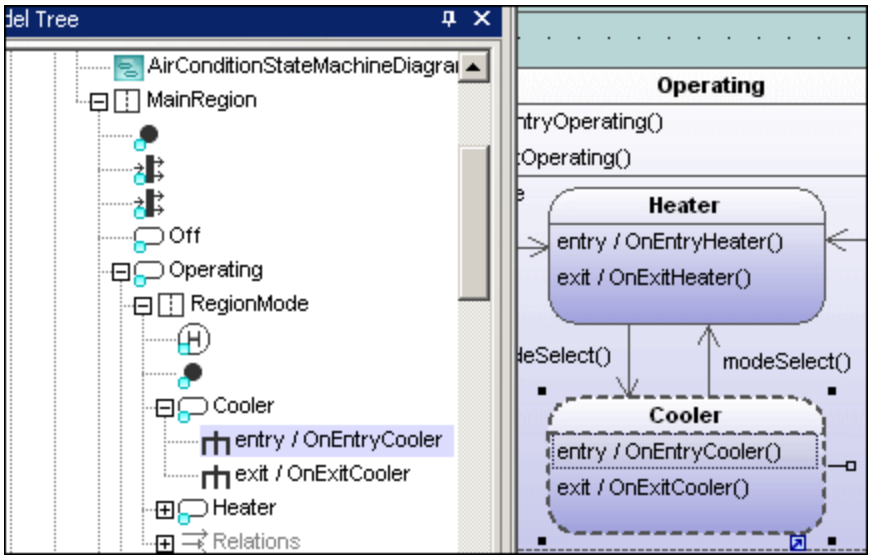
Activating the "Toggle automatic creation of operations in target by typing operation names" icon , automatically creates the corresponding operation in the referenced class, when creating a transition and entering a name e.g. myOperation().

Note: Operations can only be created automatically when the state machine is inside a class or interface.

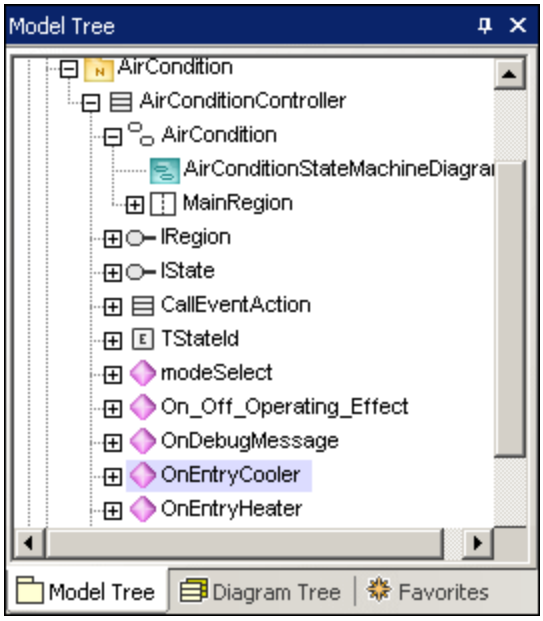
To automatically create operations from activities:

1. Right click the State and select the specific action/activity, e.g. New | Entry:Activity.
2. Enter the name of the activity making sure to finish with the open/close brackets "**()**", e.g. **entry / OnEntryCooler()**.

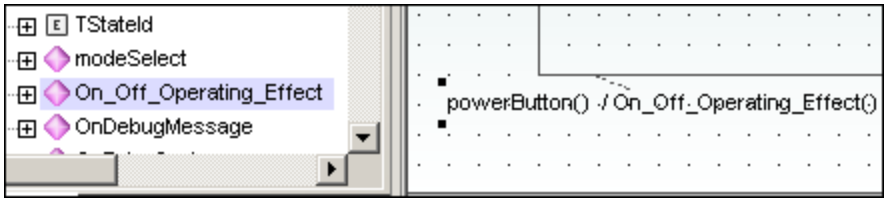
© 2019-2025 Altova GmbH



The new element is also visible in the Model Tree. Scrolling down the Model Tree, you will notice that the OnEntryCooler operation has been added to the parent class AirConditionController.



Note: Operations are automatically added for: Do:Activity, Entry:Activity, Exit:Activity, as well as guard condition activities and effects (on transitions).



To create a transition trigger:

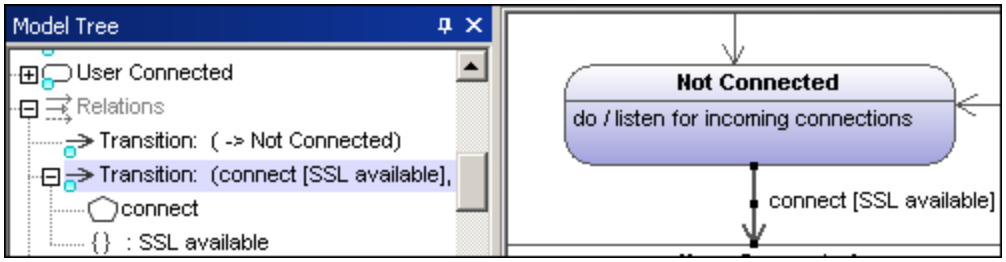
1. Right-click a previously created transition (arrow).
2. Select **New | Trigger**.

An "a" character appears in the transition label above the transition arrow, if it is the first trigger in the state diagram. Triggers are assigned default values of the form alphabetic letter, source state -> target state.

3. Double-click the new character and enter the transition properties in the form **eventTrigger [guard condition] / activity**.

> Transition property syntax
> The text entered before the square brackets is the trigger; the text between brackets is the guard condition, and the text after the slash—the activity. Manipulating this string automatically creates or deletes the respective elements in the Model Tree.

Note: To see the individual transition properties, right-click the transition (arrow) and select "Select in Model Tree". The event, activity and constraint elements are all shown below the selected transition.

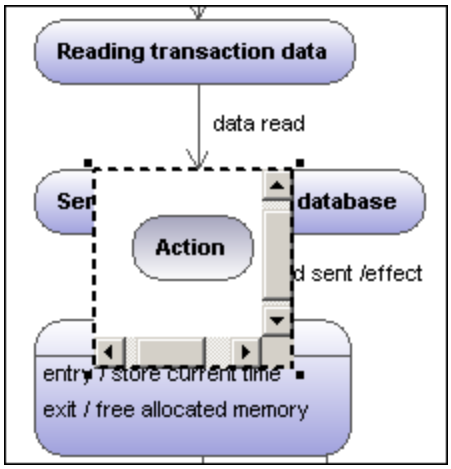


Adding an Activity diagram to a transition

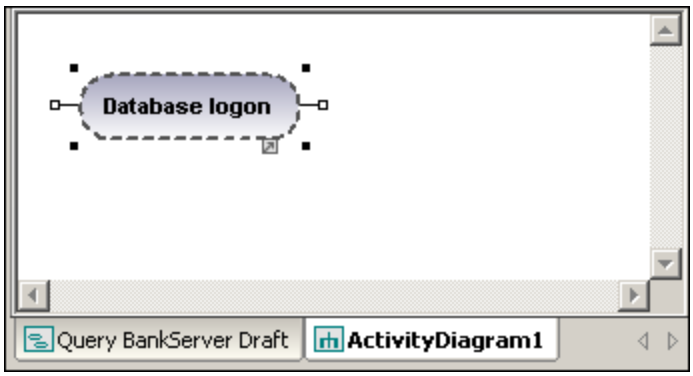
UModel has the unique capability of allowing you to add an Activity diagram to a transition, to describe the transition in more detail.

1. Right-click a transition arrow in the diagram, and select **New | Activity Diagram**. This inserts an Activity diagram window into the diagram at the position of the transition arrow.
2. Click the inserted window to make it active. You can now use the scroll bars to scroll within the window.

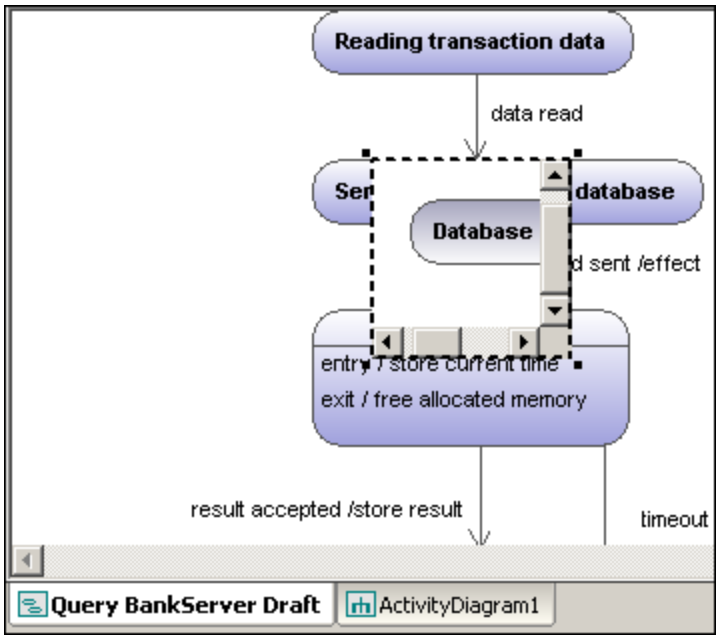
© 2019-2025 Altova GmbH



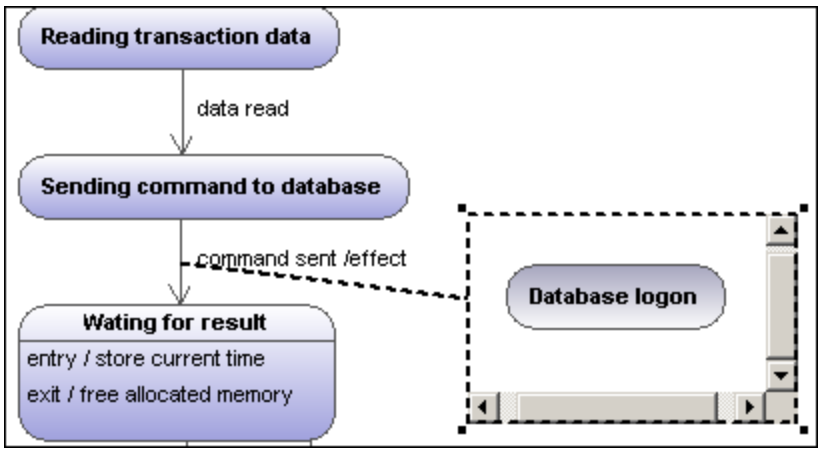
3. Double-click the Action window to switch into the Activity diagram and further define the transition, e.g. change the Action name to "Database logon". Note that a new **Activity Diagram** tab has now been added to the project. You can add any activity modeling elements to the diagram, please see "Activity Diagram 297" for more information.



4. Click the **State Machine Diagram** tab to switch back to see the updated transition.

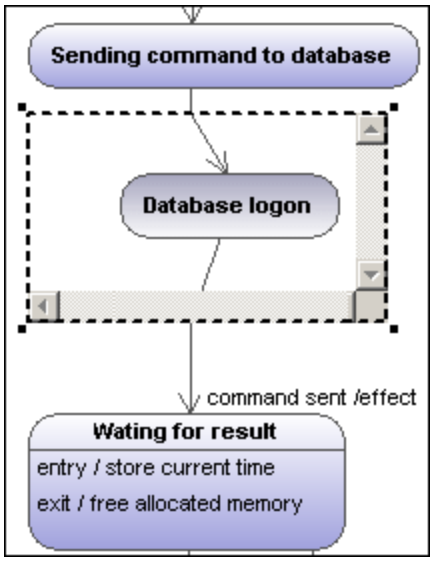


5. Drag the Activity window to reposition it in the diagram, and click the resize handle if necessary.



Dragging the Activity window between the two states displays the transition in and out of the activity.

© 2019-2025 Altova GmbH



8.1.2.3 Composite states

Composite state

This type of state contains a second compartment comprised of a single region. Any number of states may be placed within this region.

To add a region to a composite state:

- Right-click the composite state and select **New | Region** from the context menu. A new region is added to the state. Regions are divided by dashed lines.

To delete a region:

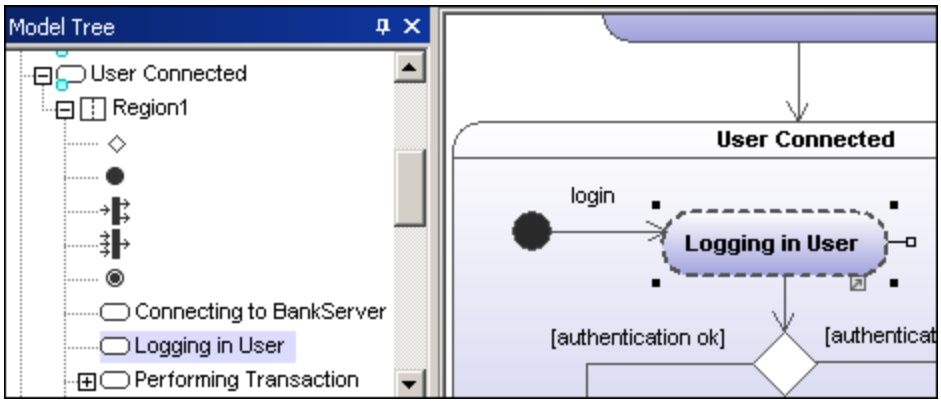
- Click the region you want to delete in the composite state and press the **Del** key.

Deleting a region of an orthogonal state reverts it back to a composite state; deleting the last region of a composite state changes it back to a simple state.

To place a state within a composite state:

- Click the state element you want to insert (e.g. Logging in User), and drop it into the region compartment of the composite state.

  The region compartment is highlighted when you can drop the element. The inserted element is now part of the region, and appears as a child element of the region in the Model Tree pane.

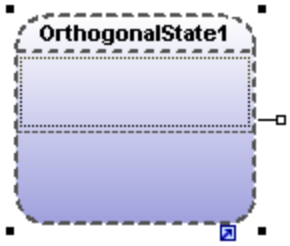


Moving the composite state moves all contained states along with it.

Orthogonal state

This type of state contains a second compartment comprised of two or more regions, where the separate regions indicate concurrency.

Right clicking a state and selecting **New | Region** allows you add new regions.



To show/hide region names:

- Click the **Styles** tab, scroll to the "Show region names on states" entry, and select true/false.

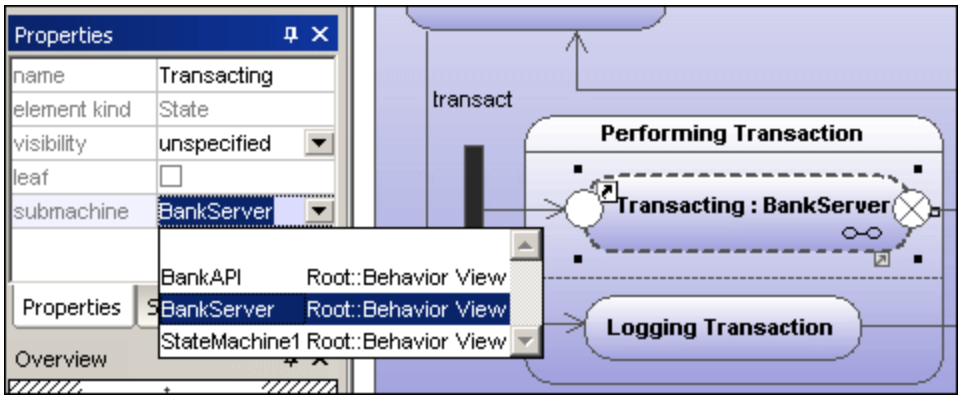
Submachine state

This state is used to hide details of a state machine. This state does not have any regions but is associated to a separate state machine.

To define a submachine state:

1. Having selected a state, click the **submachine** combo box in the **Properties** tab. A list containing the currently defined state machines appears.
2. Select the state machine that you want this submachine to reference.

© 2019-2025 Altova GmbH

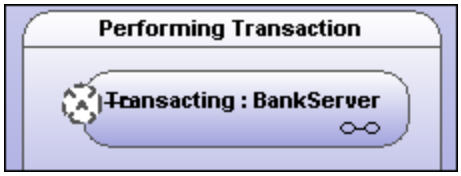


A hyperlink icon automatically appears in the submachine. Clicking it opens the referenced state machine, "BankServer" in this case.

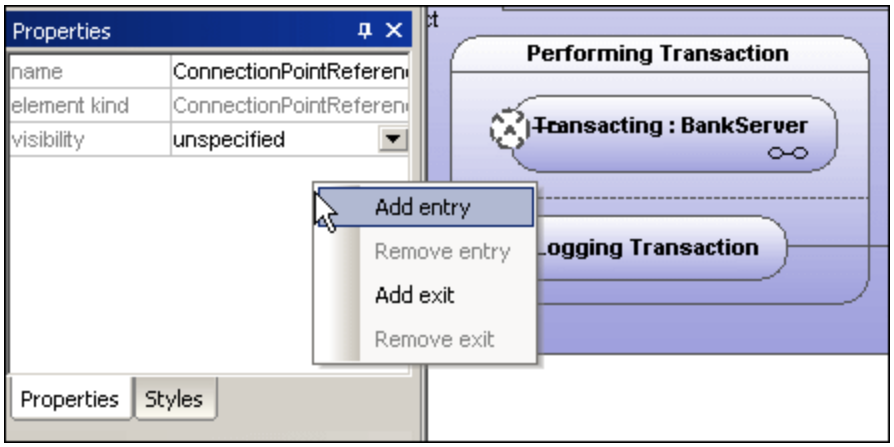
To add entry / exit points to a submachine state:

- The state which the point is connected to, must itself reference a submachine State Machine (visible in the Properties tab).
- This submachine must contain one or more Entry and Exit points

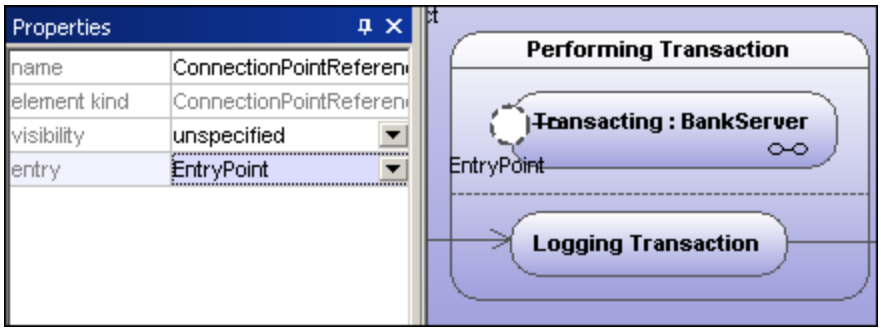
1. Click the **ConnectionPointReference** icon in the title bar, then click the submachine state that you want to add the entry/exit point to.



2. Right-click in the **Properties** tab and select **Add entry**. Please note that another Entry, or Exit Point has to exist elsewhere in the diagram to enable this pop-up menu.



This adds an EntryPoint row to the Properties tab, and changes the appearance of the ConnectionPointReference element.



3. Use the same method to insert an ExitPoint, by selecting "Add exit" from the context menu.

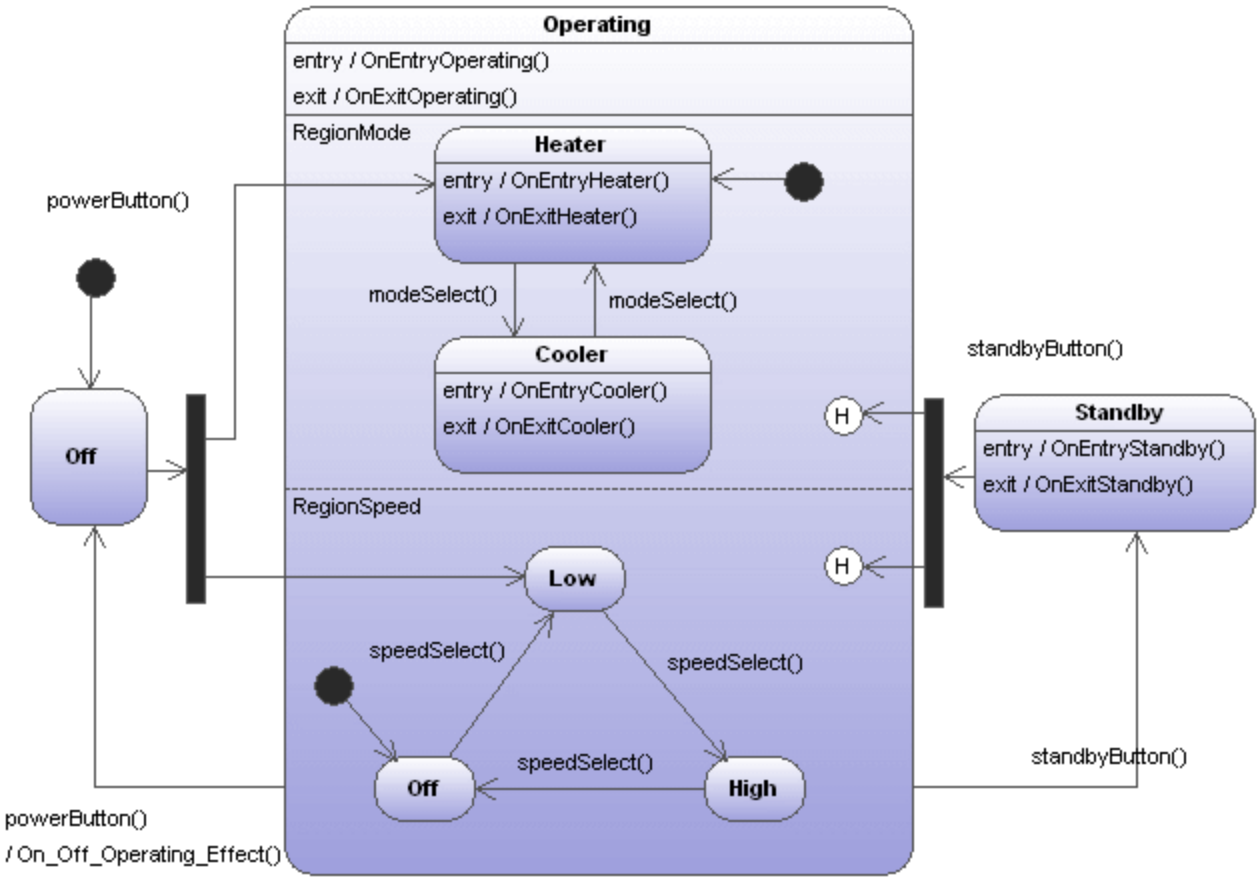
8.1.2.4 Generating code from State Machine diagrams

UModel can generate executable code from State Machine diagrams (C#, Java, VB.NET). Almost all of the State Machine diagram elements and features are supported:

- State
- CompositeState, with any hierarchical level
- OrthogonalState, with any number of regions
- Region
- InitialState
- FinalState
- Transition
- Guard
- Trigger
- Call-Event
- Fork
- Join
- Choice
- Junction
- DeepHistory
- ShallowHistory
- Entry/exit/do actions
- Effects

State Machine code generation is integrated into the "normal" round-trip engineering process. This means that State Machine code can be automatically updated on every forward-engineering process.

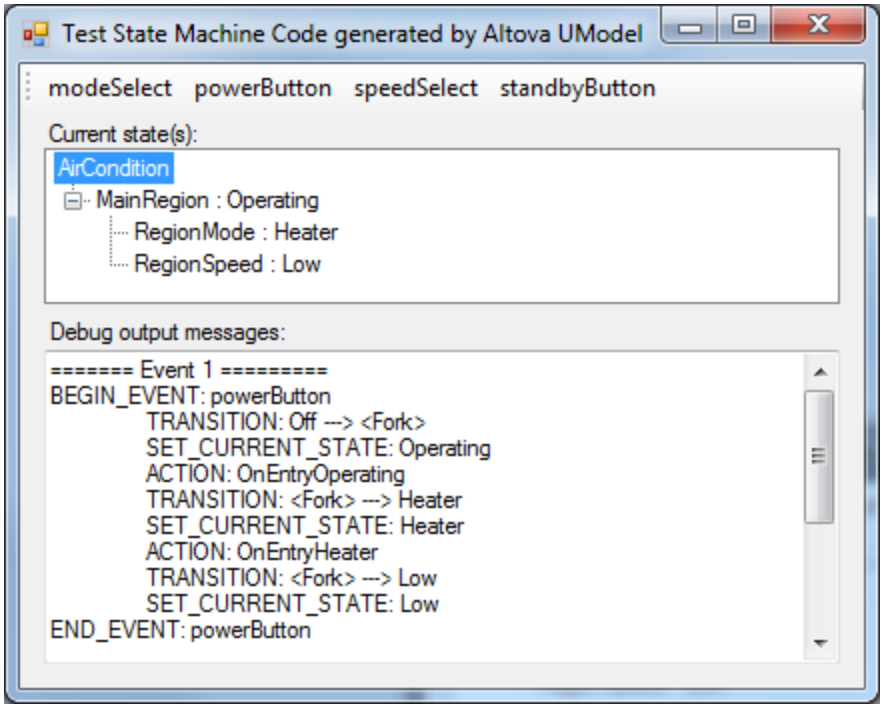
© 2019-2025 Altova GmbH



The screenshot above shows the AirCondition State Machine diagram which is available in the **..\StateMachineCodeGeneration** directory under **...\UModelExamples**. A separate directory exists for each of the code generation languages supported by UModel.

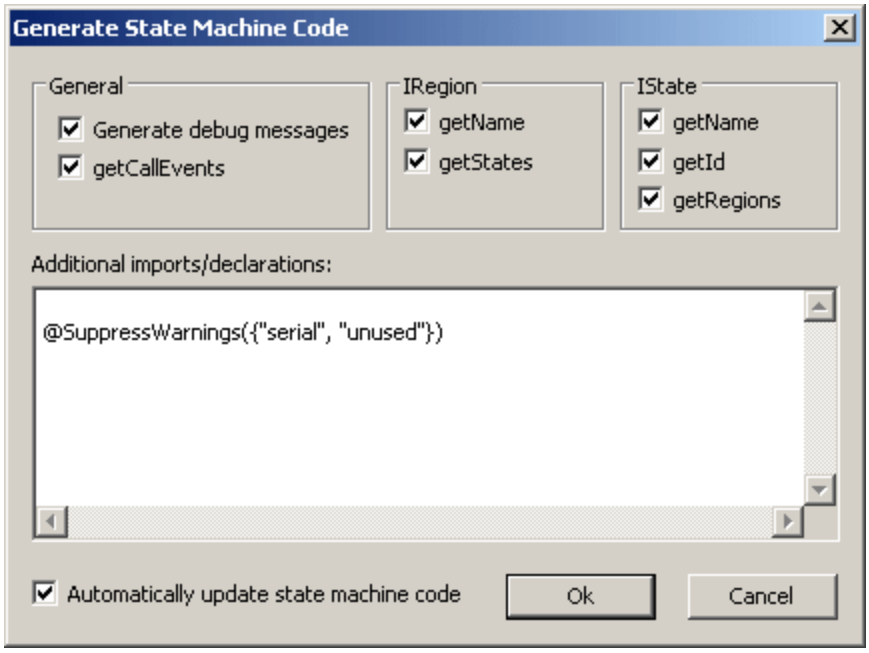
Each directory contains an AirCondition and Complex folder, which contains the respective UModel project, programming language project files, as well as the generated source files. The Complex.ump project file contains almost all of the modeling elements and functionality that UModel supports when generating code from State Machine diagrams.

Each directory also contains a test application, e.g. TestSTMAirCondition.sln for C#, allowing you to work with the generated source files immediately.



To generate code from a State Machine diagram:

- Right-click in the State Machine diagram and select "Generate State Machine code", or
- Select the menu option **Project | Generate State Machine Code**

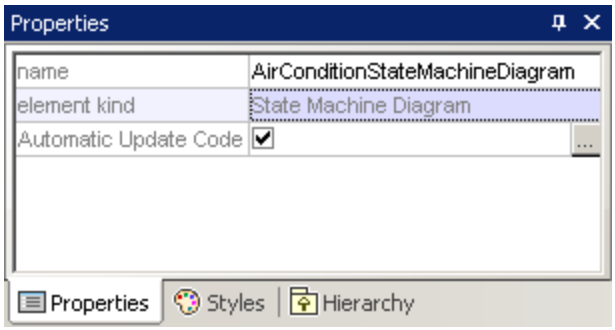


The default settings are shown above. Click OK to generate the code.

© 2019-2025 Altova GmbH

State Machine code is automatically updated when you start the forward engineering process. You can however change this setting by clicking on the State Machine diagram background and clicking the "Automatic Update Code" check box.

Changes should not be made manually in the generated code, as these changes are not reflected in the State Machine diagram during the reverse-engineering process.



Clicking the ... icon of the Automatic Update field, opens the Generate State Machine Code dialog box, allowing you to change the code generation settings.

To perform a syntax check on a State Machine diagram:

- Right-click the diagram and selecting **Check State Machine Syntax** from the context menu.

8.1.2.5 Working with state machine code

The parent class of the state machine (i.e. the "controller class", or "context class") is the one, and only, "interface" between the state machine user and the state machine implementation.

The controller class provides methods which can be used from "outside" to change the states (e.g. after external events occur).

The state machine implementation however, calls controller class methods ("callbacks") to inform the state machine user about state changes (OnEntry, OnExit, ...), transition effects, and the possibility to override and implement methods for conditions (guards).

UModel can automatically create simple operations (without any parameter) for entry/exit/do behaviors, transition effects, ... when the corresponding option is turned on (also see Creating states, activities and transitions 316). These methods can be changed to whatever you want in UModel (add parameters, set them as abstract, etc.).

A state machine (i.e. its controller class) can be instantiated several times. All instances work independently of each other.

- The UML State machine execution is designed for the "Run-to-completion execution model".
- UML state machines assume that processing of each event is completed before the next event is processed.

- This also means no entry/exit/do action or transition effect may directly trigger a new transition/state change.

Initialization

- Every region of a state machine has to have an initial state.
- The code generated by UModel automatically initializes all regions of the state machine (or when the `Initialize()` method of the controller class is called).
- If OnEntry events are not wanted during initialization, you can call the `Initialize()` method manually and ignore OnEntry events during the startup.

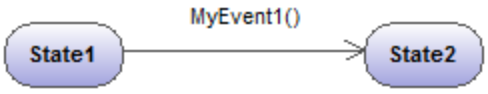
Getting the current state(s)

UModel supports composite states as well as orthogonal states, so there is not just one current state—every region (in any hierarchy level) can have one current state.

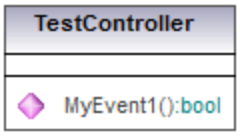
The **AirCondition.ump** example shows how to walk through the regions to the current state(s):

```
TreeNode rootNode = m_CurrentStateTree.Nodes.Add(m_STM.getRootState().getName());
UpdateCurrentStateTree(m_STM.getRootState(), rootNode);

private void UpdateCurrentStateTree(AirCondition.AirConditionController.IState state,
TreeNode node)
{
    foreach (AirCondition.AirConditionController.IRegion r in state.getRegions())
    {
      TreeNode childNode = node.Nodes.Add(r.getName() + " : " +
r.getCurrentState().getName());
      UpdateCurrentStateTree(r.getCurrentState(), childNode);
    }
}
```

Example 1 - a simple transition



The corresponding operation is automatically generated in UModel



Generated method in code:

```
private class CTestStateMachine : IState
{
       …
```

 © 2019-2025 Altova GmbH

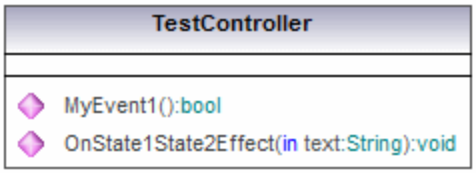
```
        public bool MyEvent1()
        {
                …
        }
}
```

Notes:

- The state machine user should call the generated method "MyEvent1" when the corresponding event occurs (outside the state machine).
- The return parameter of these event-methods provides information about whether the event caused a state change (i.e. if it had any effect on the state machine) or not. For example, if "State1" is active and event "MyEvent1()" occurs, the current state changes to "State2" and "MyEvent1()" returns true. If "State2" is active and "MyEvent1()" occurs, nothing changes in the state machine and MyEvent1() returns false.

Example 2 - a simple transition with an effect

The corresponding operation is automatically generated in UModel



Generated method in code:

```
private class CTestStateMachine : IState
{
        …
        // Override to handle entry/exit/do actions, transition effects,...:
        public virtual void OnState1State2Effect() {}
}
```

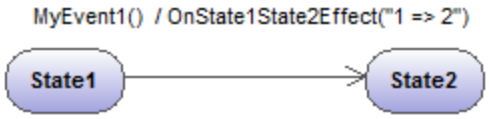
Notes:

- "OnState1State2Effect()" will be called by the state machine implementation, whenever the transition between "State1" and "State2" is fired.
- To react to this effect, "OnState1State2Effect()" should be overridden in a derived class of "CTestStateMachine".
- "CTestStateMachine:: OnState1State2Effect()" can also be set to abstract, and you will get compiler errors until the method is overridden.

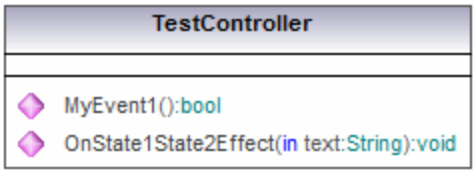
- When "OnState1State2Effect()" is not abstract, and the "Generate debug messages" option is active, UModel will generate following debug output:

```
// Override to handle entry/exit/do actions, transition effects,...:
public virtual void OnState1State2Effect() {OnDebugMessage("ACTION:
OnState1State2Effect");}
```

Example 3 - a simple transition with an effect and parameter



The corresponding operation is automatically generated in UModel



Generated method in code:

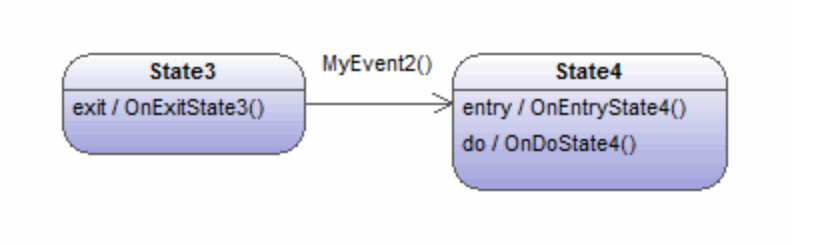
```
private class CTestStateMachine : IState
{
        …
        // Additional defined operations of the controller class:
        public virtual void OnState1State2Effect(String text)
        {
        }
}
```

Notes:

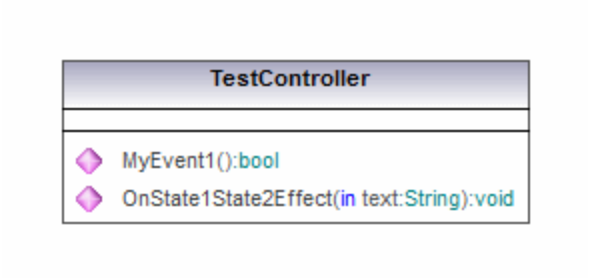
- To effect operations (automatically created by UModel) parameters can be added manually (UModel cannot know the required type).
- In this sample, the parameter "text:String" has been added to the Effect method in TestController. A proper argument has to be specified when calling this method (here: "1 => 2").
- Another possibility would be: e.g. to call static methods ("MyStatic.OnState1State2Effect("1 => 2")"), or methods of singletons ("getSingleton().OnState1State2Effect("1 => 2")").

© 2019-2025 Altova GmbH

Example 4 - entry/exit/do actions



The corresponding operations are automatically generated in UModel



Generated method in code:

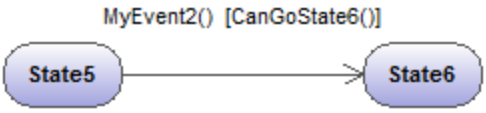
```
private class CTestStateMachine : IState
{
        …
        // Override to handle entry/exit/do actions, transition effects,...:
        public virtual void OnExitState3() {}
        public virtual void OnEntryState4() {}
        public virtual void OnDoState4() {}
}
```

Notes:

- States can have entry/exit/do behaviors. UModel automatically creates the corresponding operations to handle them.
- When "MyEvent2()" occurs in the sample above, the state machine implementation calls "OnExitState3()". If "MyEvent2" would have an Effect, it would be subsequently called, then "OnEntryState4" and "OnDoState4" would be called.
- Normally, these methods should be overridden. When they are not abstract and the "Generate debug messages" option is active, UModel provides default debug output as described in Example 2.
- These methods can also have parameters as shown in Example 3.

Example 5 - guards

Transitions can have guards, which determine if the transition really can fire.



The corresponding operation is automatically generated in UModel

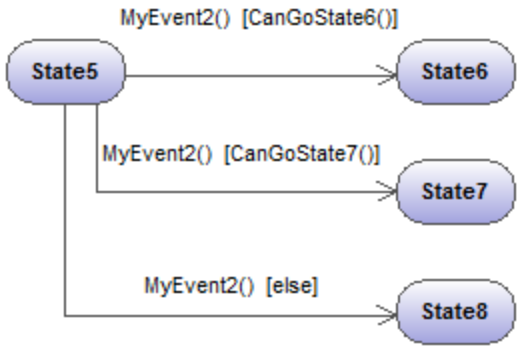
Generated method in code:

```
private class CTestStateMachine : IState
{
        …
        // Additional defined operations of the controller class:
        public virtual bool CanGoState6()
        {
            return true; // Override!
        }
}
```

Notes:

- If "State5" is the active state and "MyEvent2" occurs, the state machine implementation will call "CanGoState6" and, depending on its result, the transition will fire or not.
- Normally, these methods should be overridden. When they are not abstract and the "Generate debug messages" option is active, UModel provides default debug output as described in Example 2.
- These methods also can have parameters as shown in Example 3.
- Multiple transitions with the same event, but having different guards, are possible. The order in which the different guards are polled is undefined. If a transition does not have a guard, or the guard is "else", it will be considered as the last (i.e., only when all other transition guards return false, will this one will fire). For example, in the diagram below, it is undefined whether `CanGoState6()` or `CanGoState7()` is called first. The third transition will only fire if `CanGoState6()` and `CanGoState7()` return false.

© 2019-2025 Altova GmbH



Additional constructs and functionality can be found in the **AirCondition.ump** and **Complex.ump** samples.

8.1.2.6 State Machine Diagram elements

InitialState (pseudostate)

The beginning of the process.

FinalState

The end of the sequence of processes.

EntryPoint (pseudostate)

The entry point of a state machine or composite state.

ExitPoint (pseudostate)

The exit point of a state machine or composite state.

Choice

This represents a dynamic conditional branch, where mutually exclusive guard triggers are evaluated (OR operation).

Junction (pseudostate)

This represents an end to the OR operation defined by the Choice element.

Terminate (pseudostate)

The halting of the execution of the state machine.

Fork (pseudostate)

Inserts a vertical Fork bar. Used to divide sequences into concurrent subsequences.

Fork horizontal (pseudostate)

Inserts a horizontal Fork bar. Used to divide sequences into concurrent subsequences.

Join (pseudostate)

Joins/merges previously defined subsequences. All activities have to be completed before progress can continue.

Join horizontal (pseudostate)

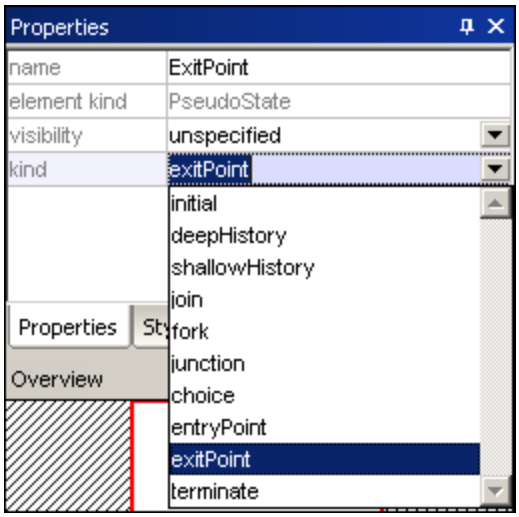
Joins/merges previously defined subsequences. All activities have to be completed before progress can continue.

DeepHistory

A pseudostate that restores the previously active state within a composite state.

ShallowHistory

A pseudostate that restores the initial state of a composite state. All pseudostate elements can be changed to a different "type", by changing the **kind** combo box entry in the Properties tab.



© 2019-2025 Altova GmbH

ConnectionPointReference

A connection point reference represents a usage (as part of a submachine state) of an entry/exit point defined in the state machine reference by the submachine state.

To add Entry or Exit points to a connection point reference:

- The state which the point is connected to, must itself reference a submachine State Machine (visible in the **Properties** tab).
- This submachine must contain one or more Entry and Exit points

Transition

A direct relationship between two states. An object in the first state performs one or more actions and then enters the second state depending on an event and the fulfillment of any guard conditions. Transitions have an event trigger, guard condition(s), an action (behavior), and a target state. The supported event subelements are:

- ReceiveSignalEvent
- SignalEvent
- SendSignalEvent
- ReceiveOperationEvent
- SendOperationEvent
- ChangeEvent.

Toggle automatic creation of operations in target by typing operation names

Activating the "Toggle automatic creation of operations in target by typing operation names" icon, automatically creates the corresponding operation in the referenced class, when creating a transition and entering a name `myOperation()`.

Note: Operations can only be created automatically when the state machine is inside a class or interface.

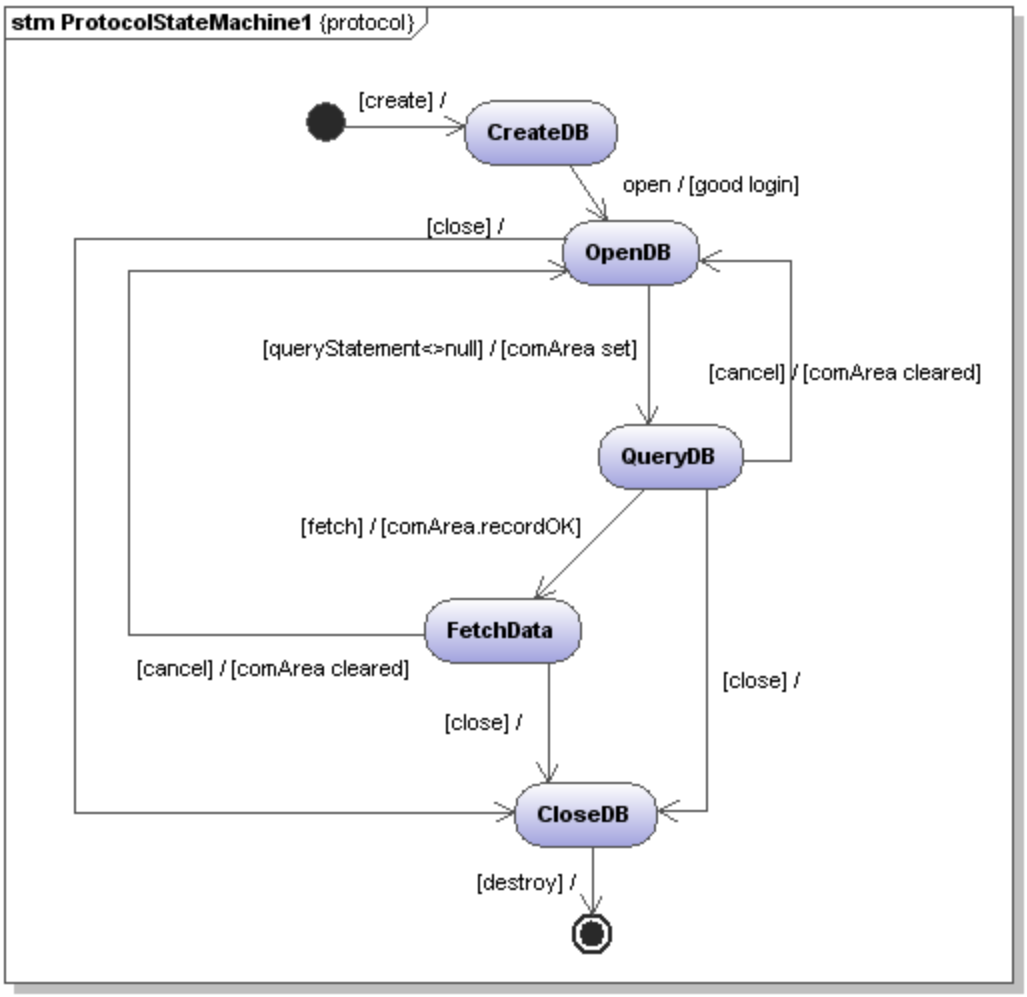
8.1.3 Protocol State Machine

Altova website: UML Protocol State Machine diagrams

Protocol State Machines are used to show a **sequence** of events that an object responds to, without having to show the specific behavior. The required sequence of events, and the resulting changes in the state of the object, are modeled in this diagram.

Protocol State Machines are most often used to describe complex protocols, e.g. database access through a specific interface, or communication protocols such as TCP/IP.

Protocol State Machines are created in the same way as State Machine diagrams, but have fewer modeling elements. Protocol-Transitions between states can have pre- or post conditions which define what must be true for a transition to another state to occur, or what the resulting state must be, once the transition has taken place.



8.1.3.1 Inserting Protocol State Machine elements

Using the toolbar icons:

1. Click the Protocol State Machine icon in the toolbar.
2. Click in the Protocol State Machine Diagram to insert the element. To insert multiple elements of the selected type, hold down the **Ctrl** key and click in the diagram window.

© 2019-2025 Altova GmbH

Dragging existing elements into the Protocol State Machine diagram

Most elements occurring in other Protocol State Machine diagrams, can be inserted into an existing diagram.

1. Locate the element you want to insert in the **Model Tree** tab (you can use the search function text box, or press **Ctrl+F** to search for any element).
2. Drag the element(s) into the Protocol State Machine diagram.

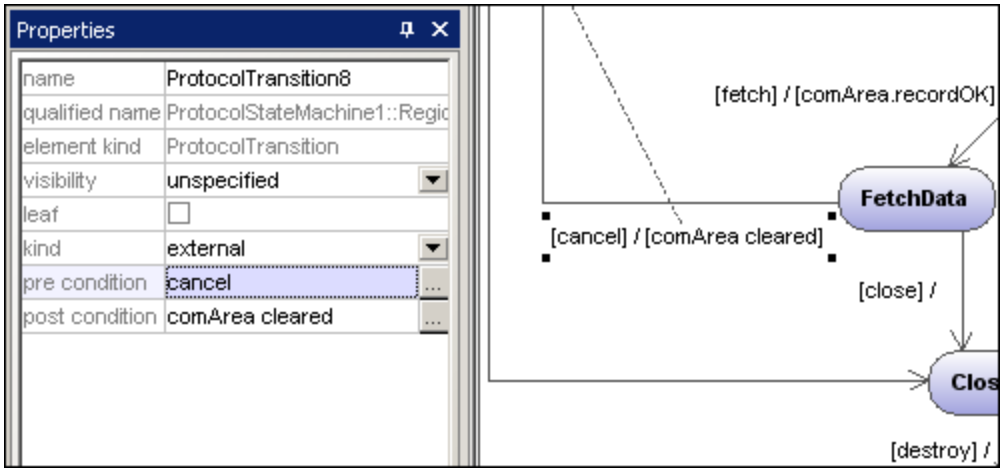
To insert a simple state:

1. Click the **State** icon in the icon bar and click in the Protocol State Machine diagram to insert it.
2. Enter the name of the state and press **Enter** to confirm. Simple states do not have any regions or any other type of substructure.

To create a Protocol Transition between two states:

1. Click the Transition handle of the source state (on the right of the element), or use the Protocol Transition icon in the icon bar.
2. Drag-and-drop the transition arrow onto the target state. The text cursor is automatically set for you to enter the pre and/or post condition. Please make sure to use the square brackets [] and slash character when entering the conditions directly.

   Entering the pre/post conditions in the Properties window automatically inserts the square brackets and slash character into the diagram.



For information about how to create and insert composite state elements and submachine states, see Composite states 323

8.1.3.2 Protocol State Machine Diagram elements

State

A simple state element with one compartment.

Composite state

This type of state contains a second compartment comprised of a single region. Any number of states may be placed within this region.

Orthogonal state

This type of state contains a second compartment comprised of two or more regions, where the separate regions indicate concurrency. Right clicking a state and selecting **New | Region** allows you add new regions.

Submachine state

This state is used to hide details of a state machine. This state does not have any regions but is associated to a separate state machine.

InitialState (pseudostate)

The beginning of the process.

FinalState

The end of the sequence of processes.

EntryPoint (pseudostate)

The entry point of a state machine or composite state.

ExitPoint (pseudostate)

The exit point of a state machine or composite state.

Choice

This represents a dynamic conditional branch, where mutually exclusive guard triggers are evaluated (OR operation).

© 2019-2025 Altova GmbH

Junction (pseudostate)

This represents an end to the OR operation defined by the Choice element.

Terminate (pseudostate)

The halting of the execution of the state machine.

Fork (pseudostate)

Inserts a vertical Fork bar. Used to divide sequences into concurrent subsequences.

Fork horizontal (pseudostate)

Inserts a horizontal Fork bar. Used to divide sequences into concurrent subsequences.

Join (pseudostate)

Joins/merges previously defined subsequences. All activities have to be completed before progress can continue.

Join horizontal (pseudostate)

Joins/merges previously defined subsequences. All activities have to be completed before progress can continue.

ConnectionPointReference

A connection point reference represents a usage (as part of a submachine state) of an entry/exit point defined in the state machine reference by the submachine state.

To add Entry or Exit points to a connection point reference:

- The state which the point is connected to, must itself reference a submachine State Machine (visible in the Properties tab).
- This submachine must contain one or more Entry and Exit points

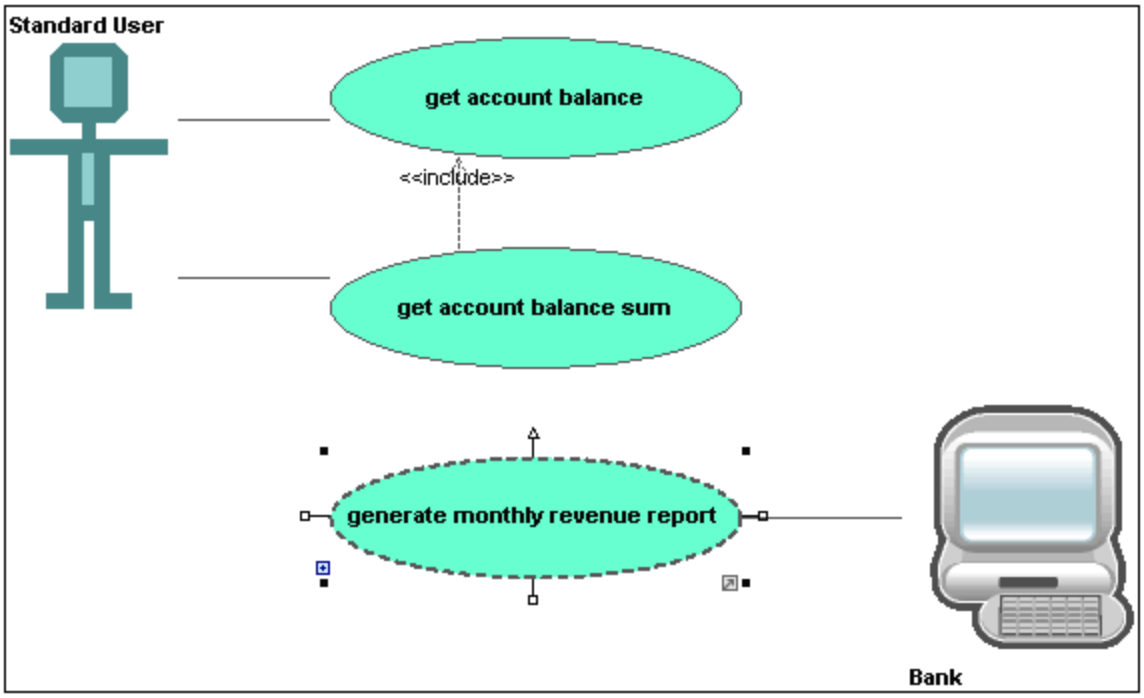
Protocol Transition

A direct relationship between two states. An object in the first state performs one or more operations and then enters the second state depending on an event and the fulfillment of any pre- or post conditions.

Please see Inserting Protocol State Machine elements 339 for more information.

8.1.4 Use Case Diagram

Please see the Use Cases [22] section in the tutorial for more information on how to add use case elements to the diagram.



8.1.5 Communication Diagram

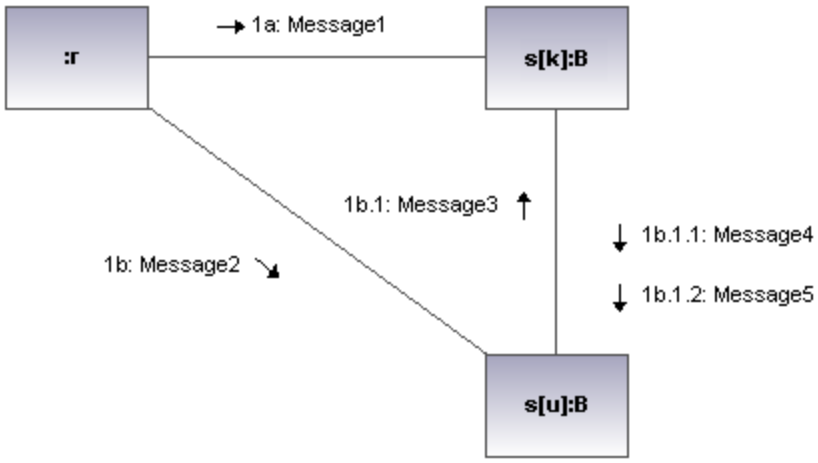
Altova website: UML Communication diagrams

Communication diagrams display the interactions i.e. message flows, between objects at run-time, and show the relationships between the interacting objects. Basically, they model the dynamic behavior of use cases.

Communication diagrams are designed in the same way as sequence diagrams, except that the notation is laid out in a different format. Message numbering is used to indicate message sequence and nesting.

UModel allows you to generate Communication diagrams from Sequence diagrams and vice versa, in one simple action see "Generating Sequence diagrams [345]" for more information.

© 2019-2025 Altova GmbH



8.1.5.1 Inserting Communication Diagam elements

Using the toolbar icons:

1. Click the specific communication icon in the Communication Diagram toolbar.

2. Click in the Communication diagram to insert the element. To insert multiple elements of the selected type, hold down the **Ctrl** key and click in the diagram window.

Dragging existing elements into the Communication Diagram

Elements occurring in other diagrams, e.g. classes, can be inserted into a Communication diagram.

1. Locate the element you want to insert in the **Model Tree** tab (you can use the search function text box, or press **Ctrl+F** to search for any element).
2. Drag the element(s) into the Communication diagram.

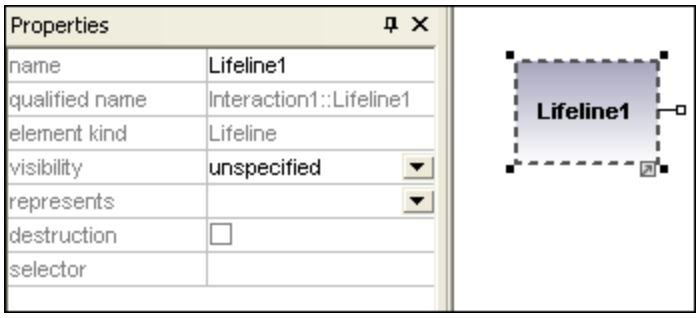
Lifeline

The lifeline element is an individual participant in an interaction. UModel allows you to insert other elements into the sequence diagram, e.g. classes. Each of these elements then appear as a new lifeline. You can redefine the lifeline colors/gradient using the "Header Gradient" combo boxes in the Styles tab.

To create a **multiline** lifeline, press **Ctrl+Enter** to create a new line.

To insert a Communication lifeline:

1. Click the Lifeline icon in the title bar, then click in the Communication diagram to insert it.



2. Enter the lifeline name to change it from the default name, Lifeline1, if necessary.

Messages

A Message is a modeling element that defines a specific kind of communication in an interaction. A communication can be e.g. raising a signal, invoking an Operation, creating or destroying an instance. The message specifies the type of communication as well as the sender and the receiver.

Message (Call)

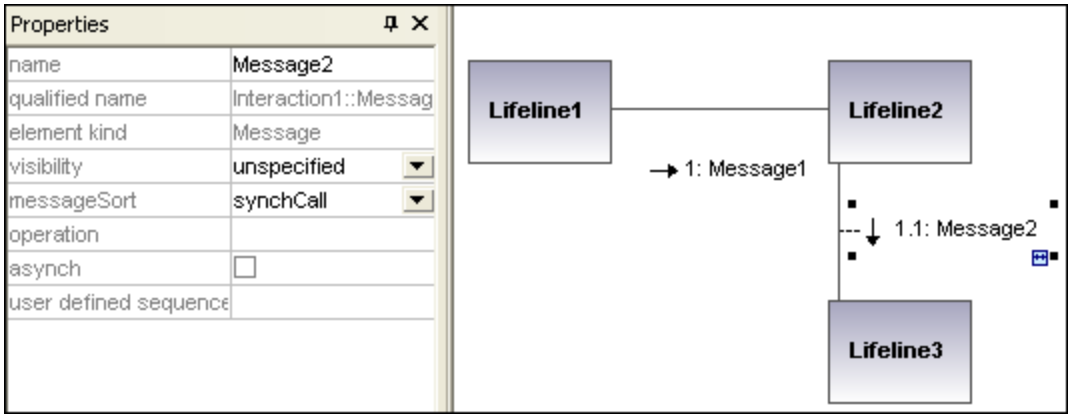
Message (Reply)

Message (Creation)

Message (Destruction)

To insert a message:

1. Click the specific message icon in the toolbar.
2. Drag and drop the message line onto the receiver objects.
   Lifelines are highlighted when the message can be dropped.

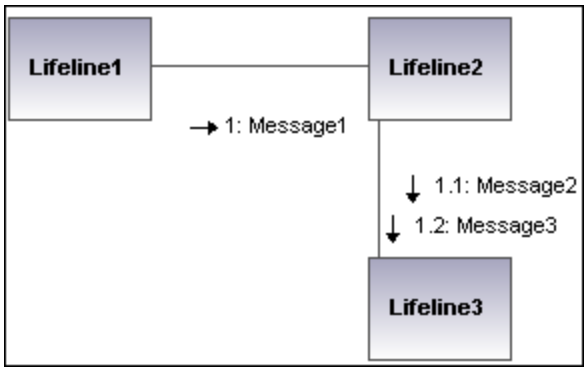


© 2019-2025 Altova GmbH

Note: Holding down the **Ctrl** key allows you to insert a message with each click.

To insert additional messages:

1. Right-click an existing communication link and select **New | Message**.



- The direction in which you drag the arrow defines the message direction. Reply messages can point in either direction.
- Having clicked a message icon and holding down **Ctrl** allows you to insert multiple messages by repeatedly clicking and dragging in the diagram tab.

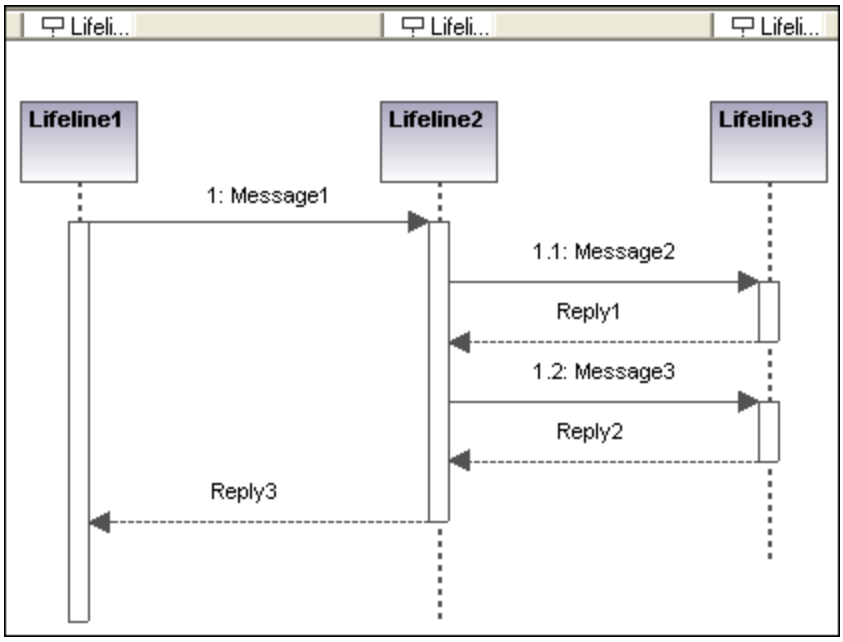
Message numbering

The Communication diagram uses the decimal numbering notation, which makes it easy to see the hierarchical structure of the messages in the diagram. The sequence is a dot-separated list of sequence numbers followed by a colon and the message name.

Generating Sequence diagrams from Communication diagrams

UModel allows you to generate Communication diagrams from Sequence diagrams and vice versa, in one simple action:

- Right-click anywhere in a Communication diagram and select **Generate Sequence Diagram** from the context menu.

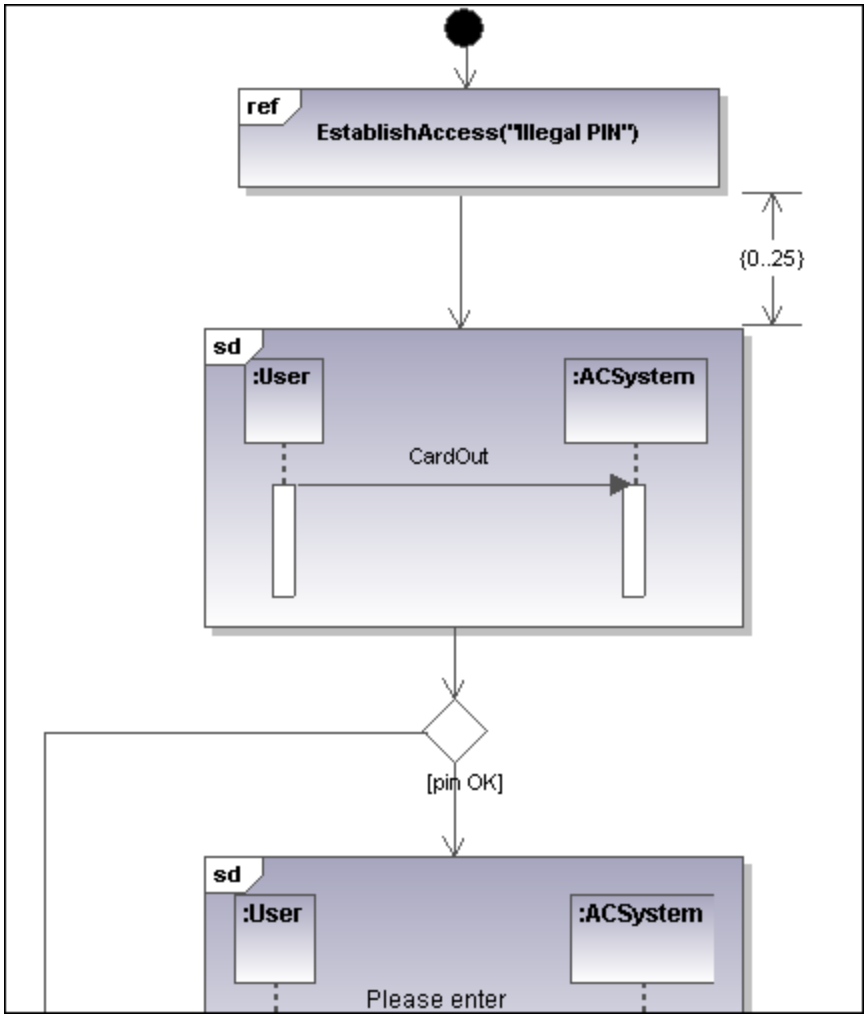


8.1.6 Interaction Overview Diagram

Altova website: UML Interaction Overview diagrams

Interaction Overview Diagrams are a variant of Activity diagrams and give an overview of the interaction between other interaction diagrams such as Sequence, Activity, Communication, or Timing diagrams. The method of constructing a diagram is similar to that of Activity diagram and uses the same modeling elements: start/end points, forks, joins etc.

© 2019-2025 Altova GmbH



Two types of interaction elements are used instead of activity elements: Interaction elements and Interaction use elements.

Interaction elements are displayed as iconized versions of a Sequence, Communication, Timing, or Interaction Overview diagram, enclosed in a frame with the "SD" keyword displayed in the top-left frame title space.

Interaction occurrence elements are references to existing Interaction diagrams with "Ref" enclosed in the frame's title space, and the occurrence's name in the frame.

8.1.6.1 Inserting Interaction Overview elements

Using the toolbar icons

1. Click the specific icon in the Interaction Overview Diagram toolbar.

2. Click in the diagram to insert the element. To insert multiple elements of the selected type, hold down the **Ctrl** key and click in the diagram window.

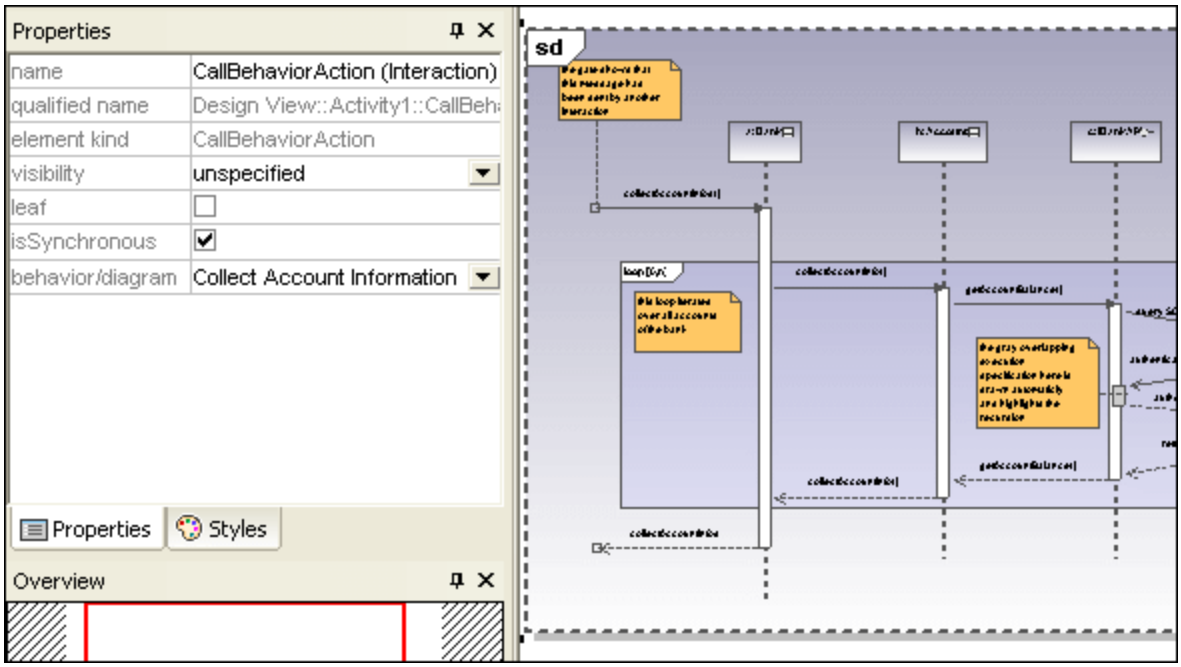
Dragging existing elements into the Interaction Overview Diagram

Elements occurring in other diagrams, e.g. Sequence, Activity, Communication, or Timing diagrams can be inserted into a Interaction Overview diagram.

1. Locate the element you want to insert in the Model Tree tab (you can use the search function text box, or press Ctrl+F, to search for any element).
2. Drag the element(s) into the diagram.

Inserting an Interaction element

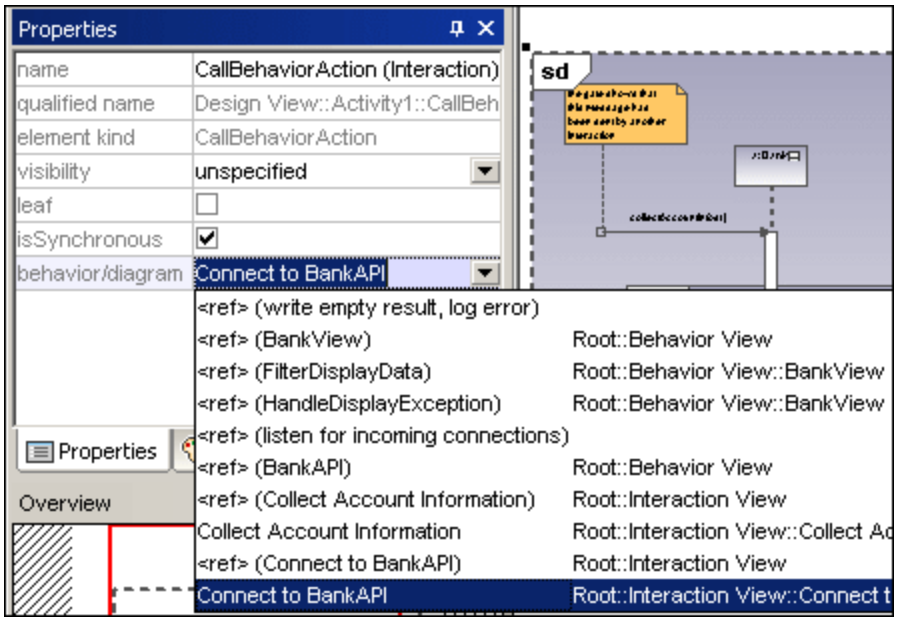
1. Click the **CallBehaviorAction (Interaction)** icon in the icon bar, and click in the Interaction Overview diagram to insert it.



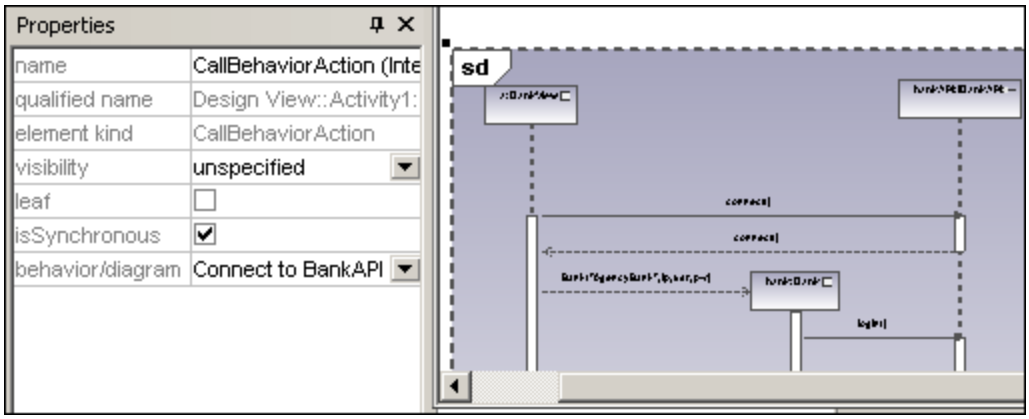
The Collect Account Information sequence diagram is automatically inserted if you are using the **Bank_Multilanguage.ump** example file from the ...**UModelExamples** folder. The first sequence diagram, found in the model tree, is selected by default.

2. To change the default interaction element: Click the **behavior/diagram** combo box in the Properties tab. A list of all the possible elements that can be inserted is presented.

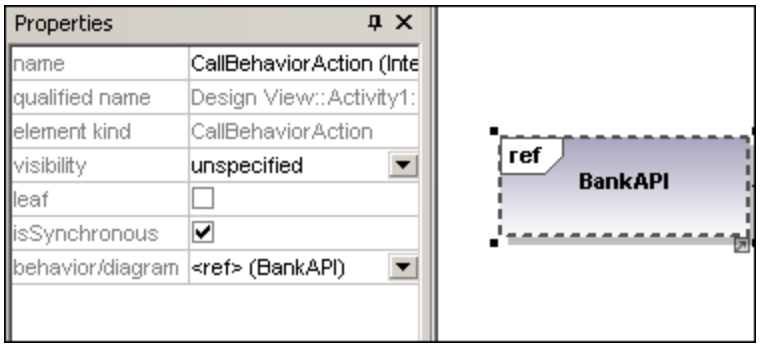
© 2019-2025 Altova GmbH



3. Click the element you want to insert to e.g. Connect to BankAPI.



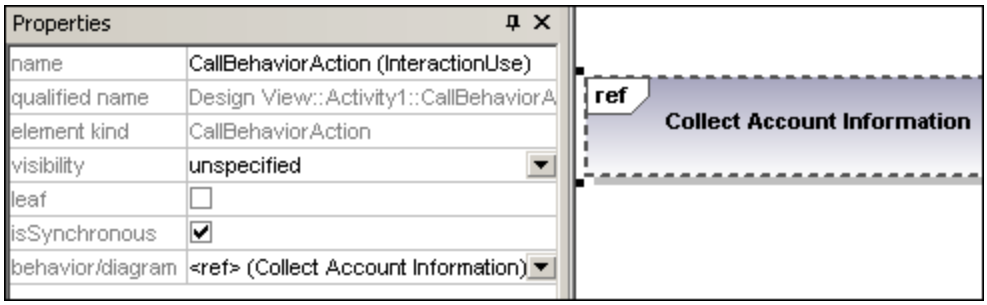
As this is also a sequence diagram, the Interaction element appears as an iconized version of the sequence diagram.
If you select **<ref> BankAPI**, then the Interaction element occurrence is displayed.



Inserting an Interaction element occurrence

1. Click the **CallBehaviorAction (InteractionUse)** icon in the icon bar, and click in the Interaction Overview diagram to insert it.

   Collect Account Information is automatically inserted as a Interaction occurrence element, if you are using the **Bank_Multilanguage.ump** example file from the ...**UModelExamples** folder. The first existing sequence diagram is selected per default.



2. To change the Interaction element, double-click the **behavior** combo box in the **Properties** tab. A list of all the possible elements that can be inserted is presented.
3. Select the occurrence you want to insert.

Note: All elements inserted using this method appear in the form shown in the screenshot above i.e. with "ref" in the frame's title space.

DecisionNode

Inserts a Decision Node which has a single incoming transition and multiple outgoing guarded transitions. Please see "Creating a branch 301" for more information.

MergeNode

Inserts a Merge Node which merges multiple alternate transitions defined by the Decision Node. The Merge Node does not synchronize concurrent processes, but selects one of the processes.

InitialNode

The beginning of the activity process. An interaction can have more than one initial node.

ActivityFinalNode

The end of the interaction process. An interaction can have more that one final node, all flows stop when the "first" final node is encountered.

© 2019-2025 Altova GmbH

ForkNode

Inserts a vertical Fork node. Used to divide flows into multiple concurrent flows.

ForkNode (Horizontal)

Inserts a horizontal Fork node. Used to divide flows into multiple concurrent flows.

JoinNode

Inserts a vertical Fork node. A Join node synchronizes multiple flows defined by the Fork node.

Join Node (horizontal)

Inserts a horizontal Fork node. A Join node synchronizes multiple flows defined by the Fork node.

AddDurationConstraint

A Duration defines a ValueSpecification that denotes a duration in time between a start and endpoint. A duration is often an expression representing the number of clock ticks, which may elapse during this duration.

ControlFlow

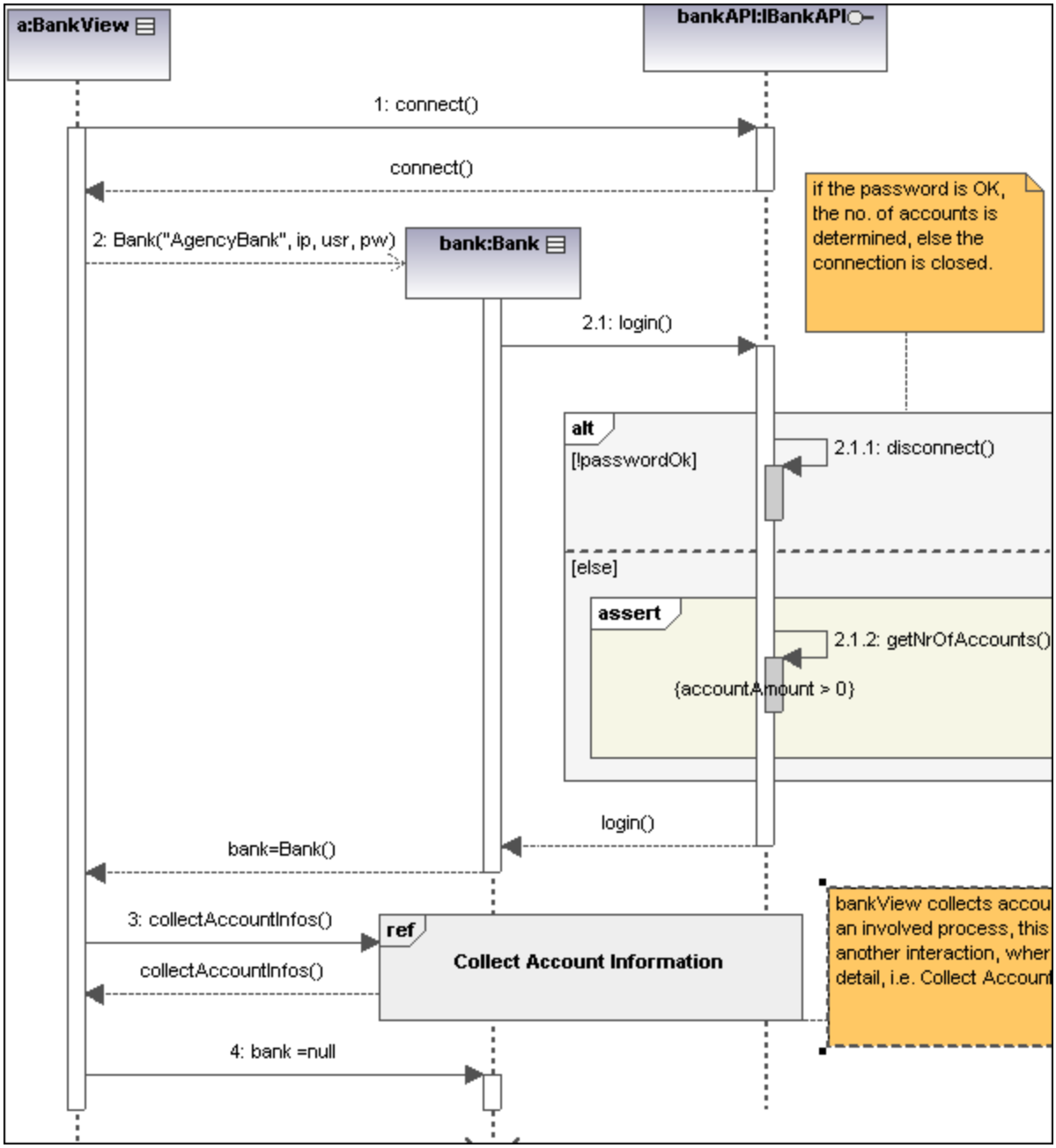
A Control Flow is an edge, i.e. an arrowed line, that connects two behaviours, and starts an interaction after the previous one has been completed.

8.1.7 Sequence Diagram

Altova website: UML Sequence diagrams

UModel supports the standard Sequence diagram defined by UML, and allows easy manipulation of objects and messages to model use case scenarios. The sequence diagrams shown in the following sections are available in the **Bank_Java.ump**, **Bank_CSharp.ump** and **Bank_MultiLanguage.ump** samples, in the **...\UModelExamples** folder supplied with UModel.

You can model sequence diagrams manually, or, alternatively, generate them from reverse-engineered source code, as described in Generating Sequence Diagrams from Source Code 366. The UModel API also provides means to generate or model a sequence diagram programmatically, see How to Create Sequence Diagrams 759.



8.1.7.1 Inserting Sequence Diagram Elements

A sequence diagram models runtime dynamic object interactions, using messages. Sequence diagrams are generally used to explain individual use case scenarios.

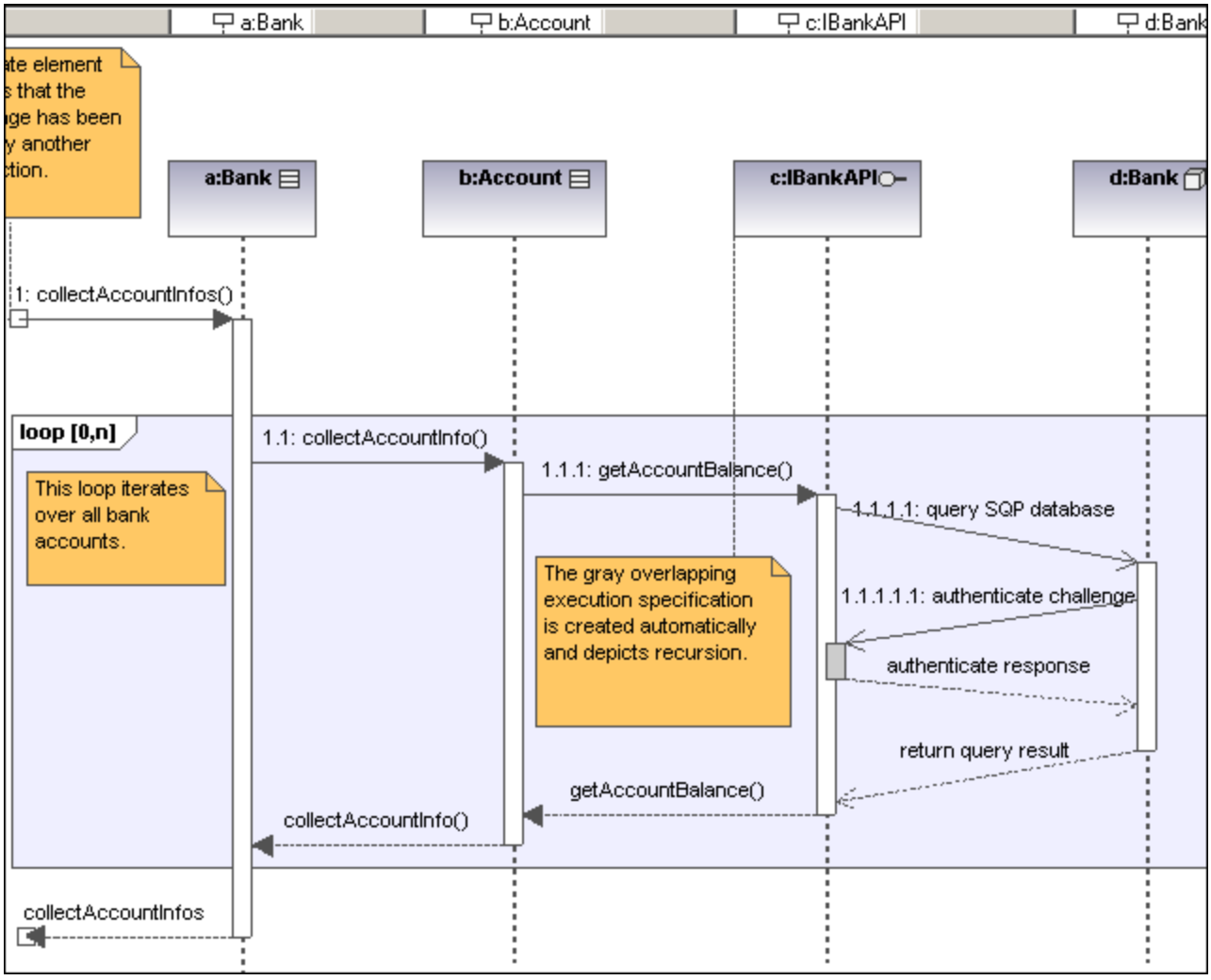
© 2019-2025 Altova GmbH

- *Lifelines* are the horizontally aligned boxes at the top of the diagram, together with a dashed vertical line representing the object's life during the interaction. Messages are shown as arrows between the lifelines of two or more objects.
- *Messages* are sent between sender and receiver objects, and are shown as labeled arrows. Messages can have a sequence number and various other optional attributes: argument list etc. Conditional, optional, and alternative messages are all supported.

See also:

- Lifeline 354
- Combined Fragment 356
- Interaction Use 359
- Gate 359
- State Invariant 360
- Messages 360

Sequence diagram and other UModel elements, can be inserted into a sequence diagram using several methods.



Using the toolbar icons

1. Click the specific sequence diagram icon in the Sequence Diagram toolbar.
2. Click in the Sequence diagram to insert the element. To insert multiple elements of the selected type, hold down the **Ctrl** key and click in the diagram window.

Dragging existing elements into the sequence diagram

Most classifier types, as well as elements occurring in other sequence diagrams, can be inserted into an existing sequence diagram.

1. Locate the element you want to insert in the **Model Tree** tab (you can use the search function text box, or press **Ctrl+F** to search for any element).
2. Drag the element(s) into the sequence diagram.

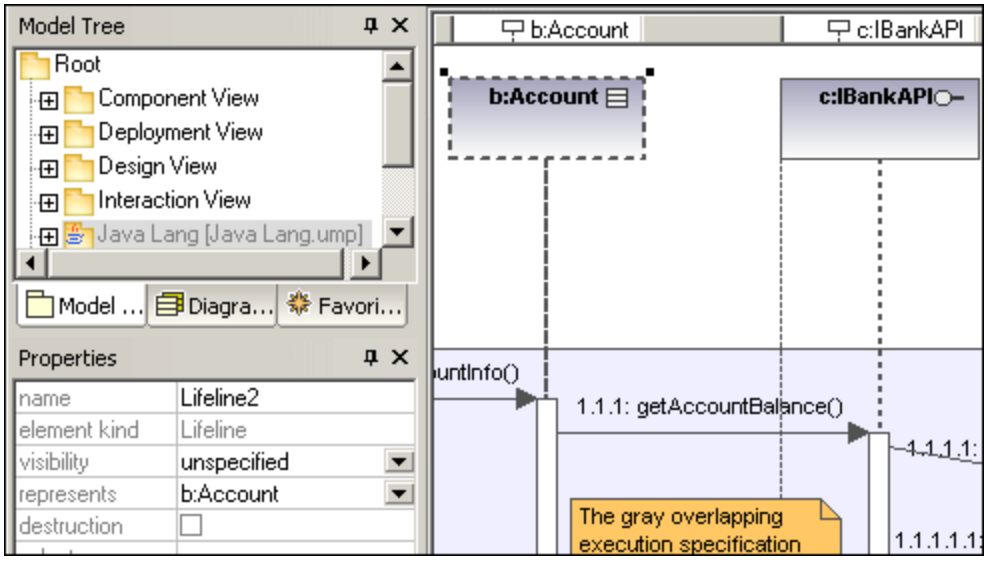
8.1.7.1.1 Lifeline

The **lifeline** element is an individual participant in an interaction. UModel also allows you to insert other elements into the sequence diagram, e.g. classes and actors. Each of these elements appear as a new lifeline once they have been dragged into the diagram pane from the **Model Tree** tab.

The "lifeline" label appears in a bar at the top of the sequence diagram. Labels can be repositioned and resized in the bar, with changes taking immediate effect in the diagram tab. You can also redefine the label colors/gradient using the "Header Gradient" combo boxes in the **Styles** tab.

To create a **multiline** lifeline, press **Ctrl+Enter** to create a new line.

Most classifier types can be inserted into the sequence diagram. The "represents" field in the Properties tab displays the element type that is acting as the lifeline. Dragging **typed** properties onto a sequence diagram also creates a lifeline.



 © 2019-2025 Altova GmbH

Execution Specification (Object activation)

An execution specification (activation) is displayed as a box (rectangle) on the object lifeline. An activation is the execution of a procedure and the time needed for any nested procedures to execute. Activation boxes are automatically created when a message is created between two lifelines.

A recursive, or self message (one that calls a different method in the same class) creates stacked activation boxes.

To display/hide activation boxes:

- Click the **Styles** tab and scroll to the bottom of the list.

The "**Show Execution Specifications**" combo box allows you to show/hide the activation boxes in the sequence diagram.

Lifeline attributes

The **destruction** check box allows you to add a destruction marker, or stop, to the lifeline without having to use a destruction message.

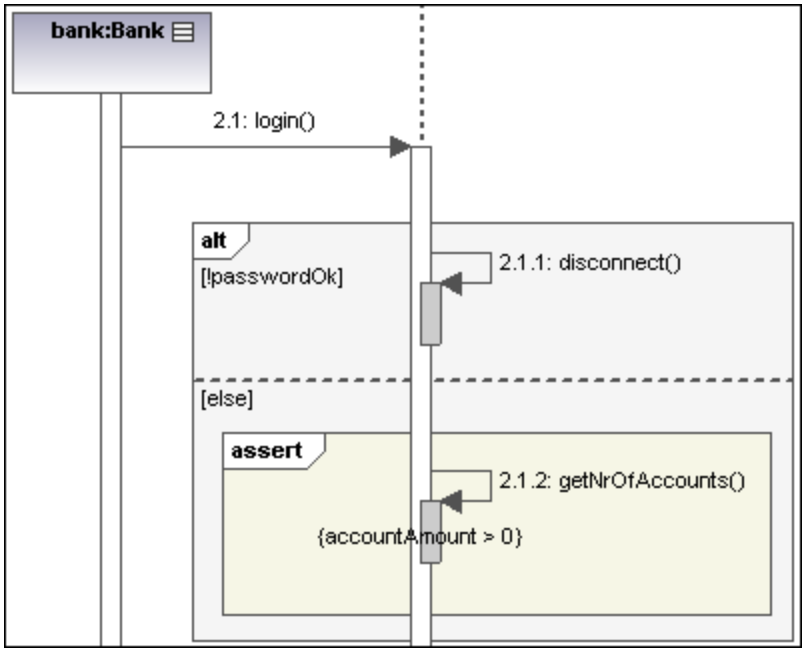
The **selector** field allows you to enter an expression that specifies the particular part represented by the lifeline, if the ConnectableElement is multivalued, i.e. has a multiplicity greater than one.

Goto lifeline element

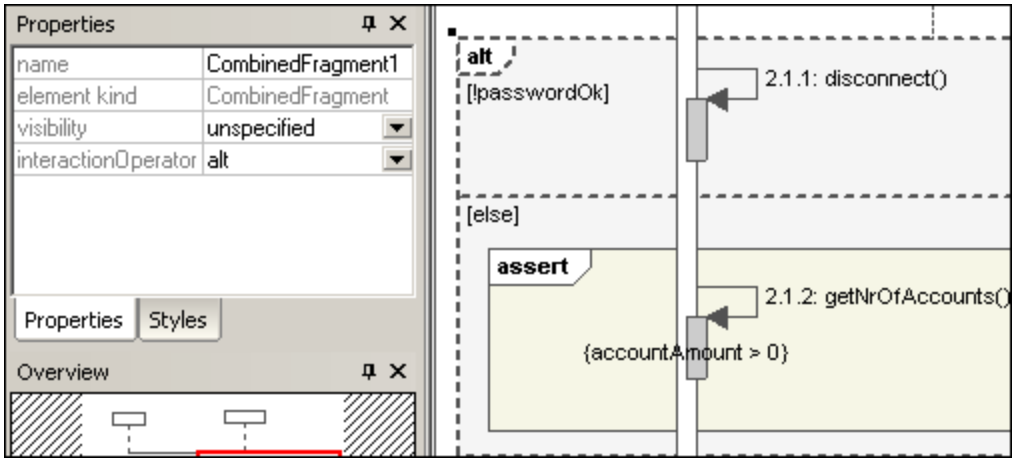
Right clicking a lifeline allows you to select Goto XXX, where XXX is the specific lifeline type that you clicked. The element will then be visible in the Model Tree window.

8.1.7.1.2 Combined Fragment

Combined fragments are subunits, or sections of an interaction. The **interaction operator** visible in the pentagon at top left, defines the specific kind of combined fragment. The constraint thus defines the specific fragment, e.g. loop fragment, alternative fragment etc. used in the interaction.

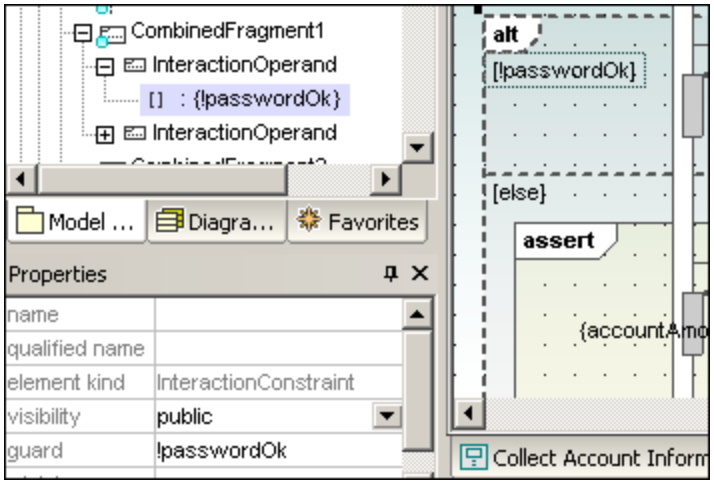


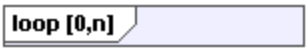
The combined fragment icons in the icon bar allow you to insert a specific combined fragment: seq, alt or loop. Clicking the **interactionOperator** combo box also allows you to define the specific interaction fragment.



© 2019-2025 Altova GmbH

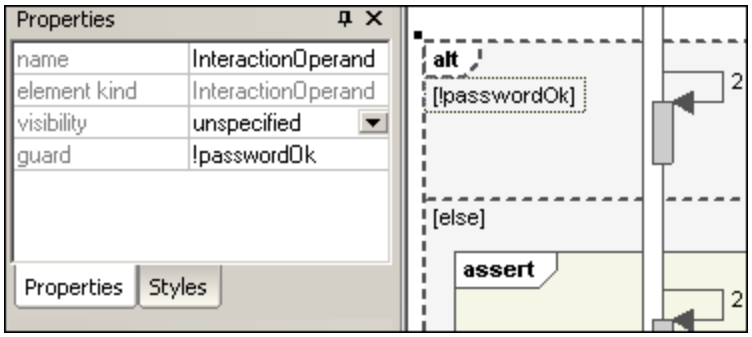
InteractionOperators

Weak sequencing	**seq**	 The combined fragment represents weak sequencing between the behaviours of the operands.
Alternatives	**alt**	Only one of the defined operands will be chosen, the operand must have a guard expression that evaluates to true. If one of the operands uses the guard "else", then this operand is executed if all other guards return false. The guard expression can be entered immediately upon insertion, will appear between the two square brackets.  The **InteractionConstraint** is actually the guard expression between the square brackets.
Option	**opt**	Option represents a choice where either the sole operand is executed, or nothing happens.
Break	**break**	The break operator is chosen when the guard is true, the rest of the enclosing fragment is ignored.
Parallel	**par**	Indicates that the combined fragment represents a parallel merge of operands.
Strict sequencing	**strict**	The combined fragment represents a strict sequencing between the behaviours of the operands.

Loop	**loop**	The loop operand will be repeated by the number of times defined in the guard expression.  Having selected this operand, you can directly edit the expression (in the loop pentagon) by double clicking.
Critical Region	**critical**	The combined fragment represents a critical region. The sequence(s) may not be interrupted/interleaved by any other processes.
Negative	**neg**	Defines that the fragment is invalid, and all others are considered to be valid.
Assert	**assert**	Designates the valid combined fragment, and its sequences. Often used in combination with consider, or ignore operands.
Ignore	**ignore**	Defines which messages should be ignored in the interaction. Often used in combination with assert, or consider operands.
Consider	**consider**	Defines which messages should be considered in the interaction.

Adding InteractionOperands to a combined fragment

1. Right-click the combined fragment and select **New | InteractionOperand**. The text cursor is automatically set for you to enter the guard condition.
2. Enter the guard condition for the InteractionOperand e.g. **!passwordOK** and press Enter to confirm. Use **Ctrl+Enter** to create a **multi-line** InteractionOperand.



3. Use the same method to add the second interaction operand with the guard condition "else". Dashed lines separate the individual operands in the fragment.

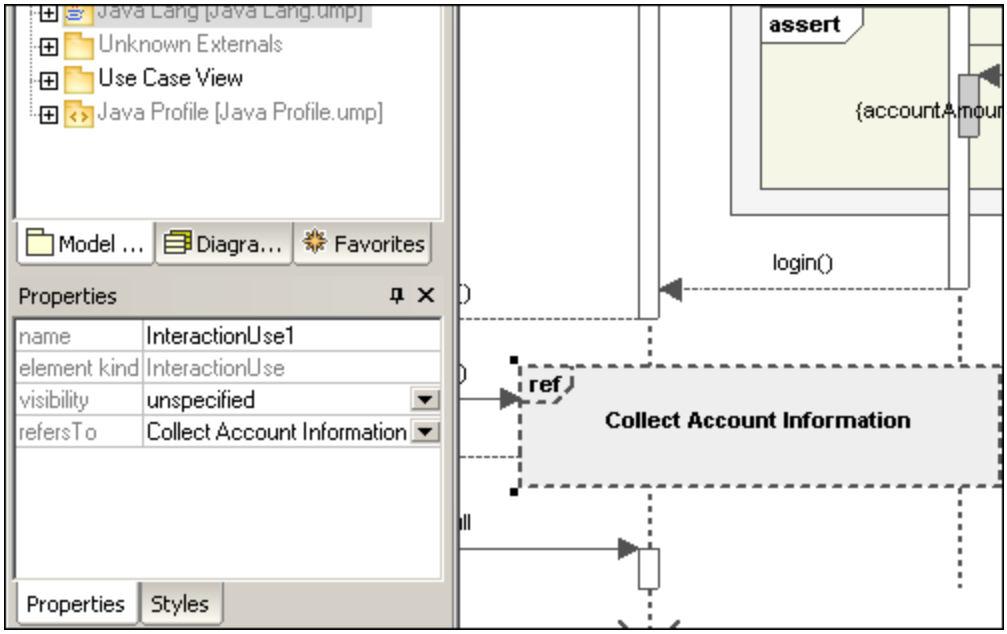
Deleting InteractionOperands

1. Double-click the guard expression in the combined fragment element, of the diagram (not in the **Properties** tab).
2. Delete the guard expression completely, and press Enter to confirm. The guard expression/interaction operand is removed and the combined fragment is automatically resized.

© 2019-2025 Altova GmbH

8.1.7.1.3 Interaction Use

The **InteractionUse** element is a reference to an interaction element. This element allows you to share portions of an interaction between several other interactions.



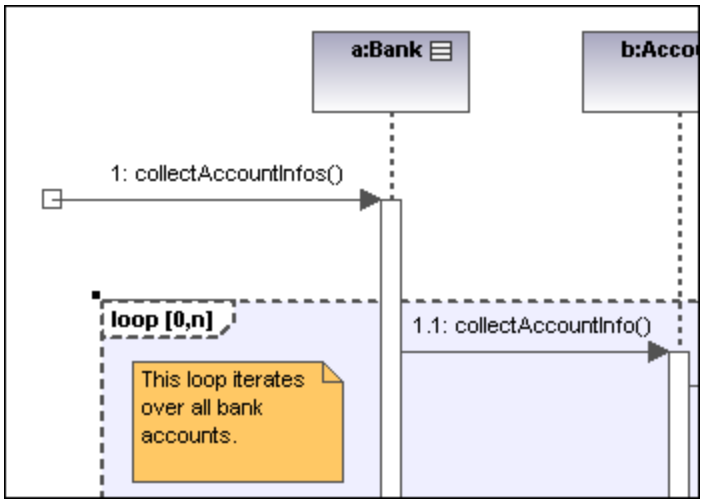
Clicking the "refersTo" combo box, allows you to select the interaction that you want to refer to. The name of the interaction use you select appears in the element.

Note: You can also drag an existing Interaction Use element from the Model Tree into the diagram tab.

8.1.7.1.4 Gate

A **gate** is a connection point which allows messages to be transmitted into, and out of, interaction fragments. Gates are connected using messages.

1. Insert the gate element into the diagram.
2. Create a new message and drag from the gate to a lifeline, or drag from a lifeline and drop onto a gate. This connects the two elements. The square representing the gate is now smaller.

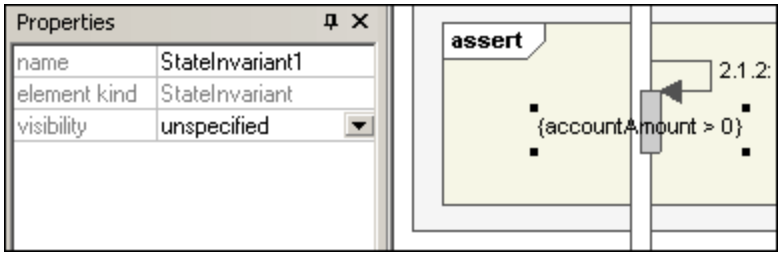


8.1.7.1.5 State Invariant

A **StateInvariant** is a condition, or constraint applied to a lifeline. The condition must be fulfilled for the lifeline to exist.

To define a StateInvariant:

1. Click the **State invariant** icon, then click a lifeline, or an object activation to insert it.
2. Enter the condition/constraint you want to apply, e.g. `accountAmount > 0`, and press **Enter** to confirm.



8.1.7.1.6 Messages

Messages are sent between sender and receiver lifelines, and are shown as labeled arrows. Messages can have a sequence number and various other optional attributes: argument list etc. Messages are displayed from top to bottom, i.e. the vertical axis is the time component of the sequence diagram.

- A **call** is a synchronous, or asynchronous communication which invokes an operation that allows control to return to the sender object. A call arrow points to the **top** of the activation that the call initiates.

© 2019-2025 Altova GmbH

- Recursion, or calls to another operation of the same object, are shown by the stacking of activation boxes (Execution Specifications).

To insert a message:

1. Click the specific message icon in the Sequence Diagram toolbar.
2. Click the lifeline, or activation box of the sender object.
3. Drag and drop the message line onto the receiver objects lifeline or activation box. Object lifelines are highlighted when the message can be dropped.

- The direction in which you drag the arrow defines the message direction. Reply messages can point in either direction.
- Activation box(es) are automatically created, or adjusted in size, on the sender/receiver objects. You can also manually size them by dragging the sizing handles.
- Depending on the message numbering settings you have enabled, the numbering sequence is updated.
- Having clicked a message icon and holding down **Ctrl** key, allows you to insert multiple messages by repeatedly clicking and dragging in the diagram tab.

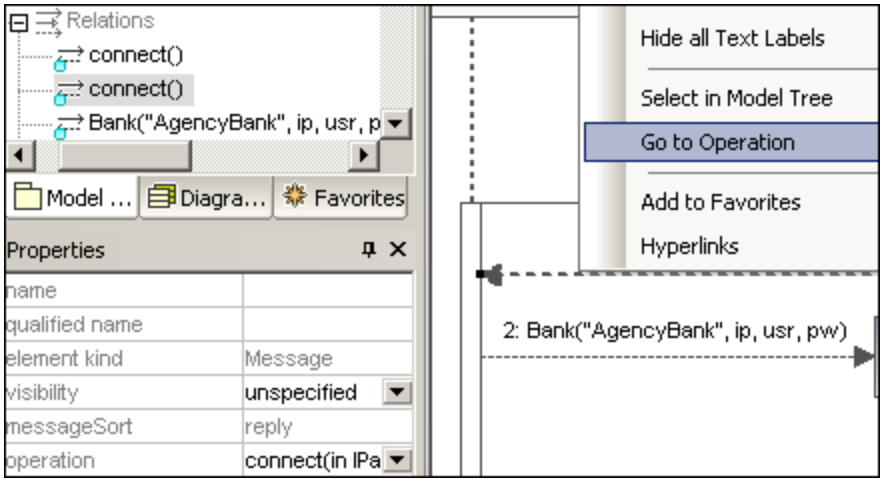
To delete a message:

1. Click the specific message to select it.
2. Press the **Del.** key to delete it from the model, or right click it and select "Delete from diagram". The message numbering and activation boxes of the remaining objects are updated.

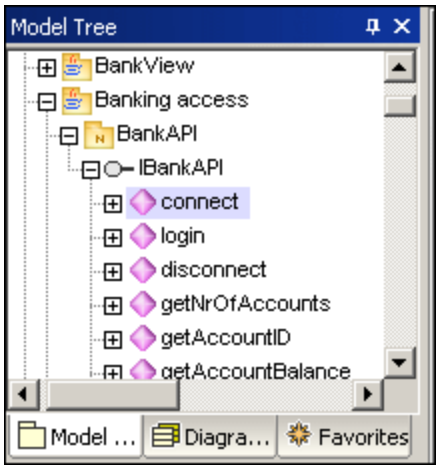
"Go to operation" for call messages:

The operations referenced by call messages can be found in sequence and communication diagrams.

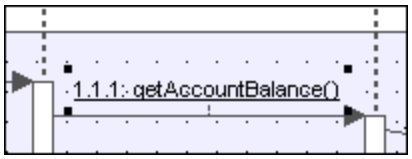
1. Right-click a call message and select "Go to Operation".



The display changes and the connect operation is displayed in the Model Tree tab.



Note: Static operation names are shown as underlined in sequence diagrams.



To position dependent messages:

- Click the respective message and drag vertically to reposition it.

The default action when repositioning messages is to move all dependent messages related to the active one. Using **Ctrl+Click** allows you to select multiple messages.

To position messages individually:

1. Click the **Toggle dependent message movement** icon to deselect it.
2. Click the message you want to move and drag to move it.

Only the selected message moves during dragging. You can position the message anywhere in the vertical axis between the object lifelines.

To automatically create reply messages:

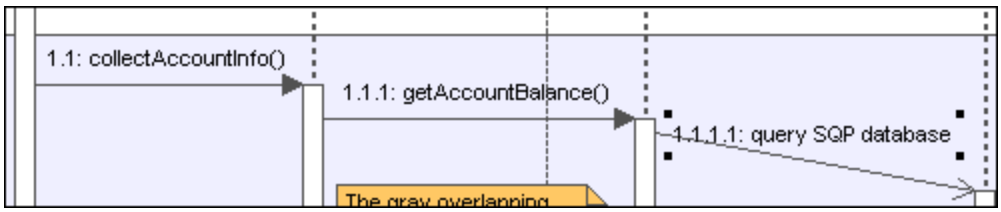
1. Click the "**Toggle automatic creation of replies for messages**" icon .
2. Create a new message between two lifelines. A reply message is automatically inserted for you.

Message numbering

UModel supports different methods of message numbering: nested, simple and none.

© 2019-2025 Altova GmbH

- **None** removes all message numbering.
- **Simple** assigns a numerical sequence to all messages from top to bottom i.e. in the order that they occur on the time axis.
- **Nested** uses the decimal notation, which makes it easy to see the hierarchical structure of the messages in the diagram. The sequence is a dot-separated list of sequence numbers followed by a colon and the message name.



There are two methods of selecting the numbering scheme:

- Click the respective icon in the icon bar.
- Use the **Styles** tab to select the scheme.

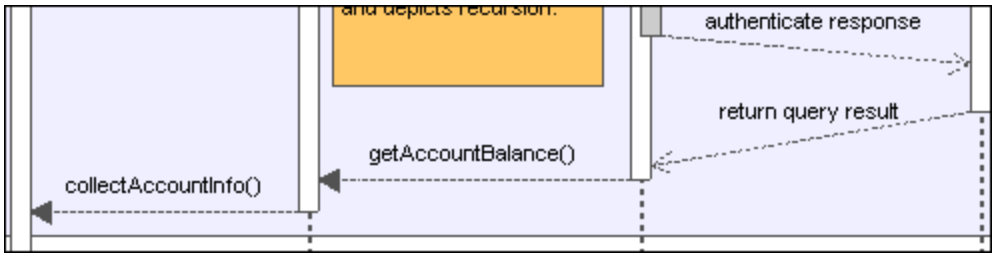
To select the numbering scheme using the Styles tab:

1. Click the **Styles** tab and scroll down to the **Show Message Numbering** field.
2. Click the combo box and select the numbering option you want to use. The numbering option you select is immediately displayed in the sequence diagram.

Note: The numbering scheme might not always correctly number all messages, if ambiguous traces exist. If this happens, adding return messages will probably clear up any inconsistencies.

Message replies

Message reply icons are available to create reply messages, and are displayed as dashed arrows.

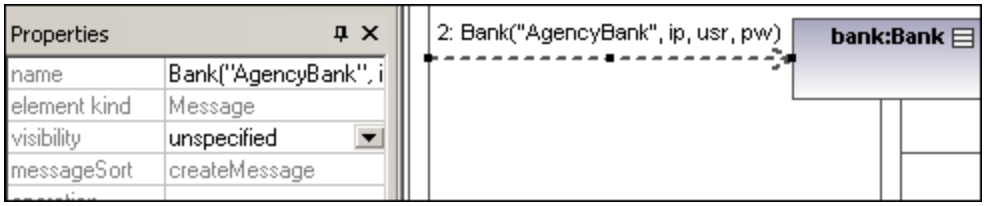


Reply messages are also generally implied by the bottom of the activation box when activation boxes are present. If activation boxes have been disabled (**Styles tab | Show Execution Specifics=false**), then reply arrows should be used for clarity.

Activating the "toggle reply messages" icon, automatically creates syntactically correct reply messages when creating a call message between lifelines/activations boxes.

Creating objects with messages

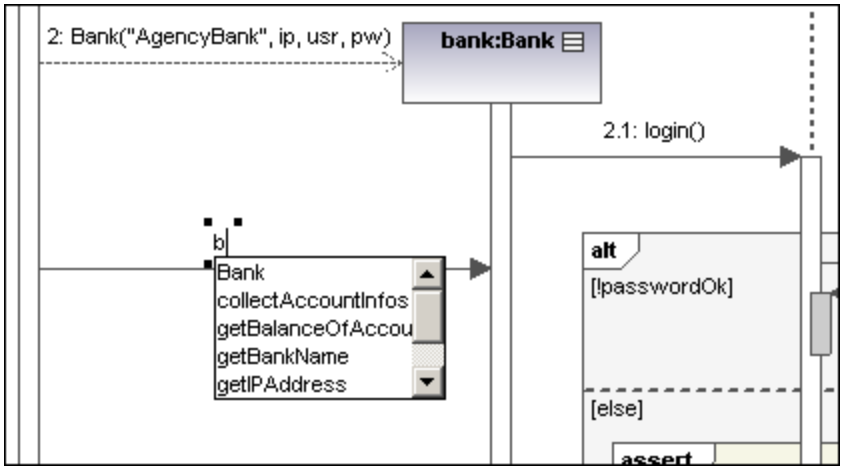
1. Messages can create new objects. This is achieved using the **Message Creation** icon .
2. Drag the message arrow to the lifeline of an existing object to create that object. This type of message ends in the middle of an object rectangle, and often repositions the object box vertically.



Sending messages to specific class methods/operations in sequence diagrams

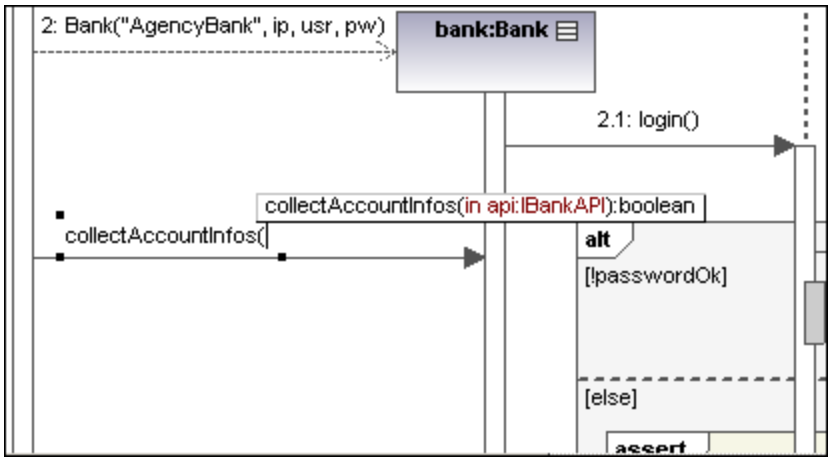
Having inserted a class from the Model Tree into a sequence diagram, you can then create a message from a lifeline to a specific method of the receiver class (lifeline) using UModel's syntax help and autocompletion functions.

1. Create a message between two lifelines, the receiving object being a class lifeline (Bank). As soon as you drop the message arrow, the message name is automatically highlighted.
2. Enter a character using the keyboard e.g. "b". A pop-up window containing a list of the existing class methods is opened.



3. Select an operation from the list, and press **Enter** to confirm e.g. `collectAccountInfos`.
4. Press the space bar and press **Enter** to select the parenthesis character that is automatically supplied. A syntax helper now appears, allowing you to enter the parameter correctly.

© 2019-2025 Altova GmbH



Creating operations in referenced classes

Activating the **Toggle automatic creation of operations in target by typing operation names** icon, automatically creates the corresponding operation in the referenced class, when creating a message and entering a name e.g. `myOperation()`.

Note: Operations can only be created automatically when the lifeline references a class or interface.

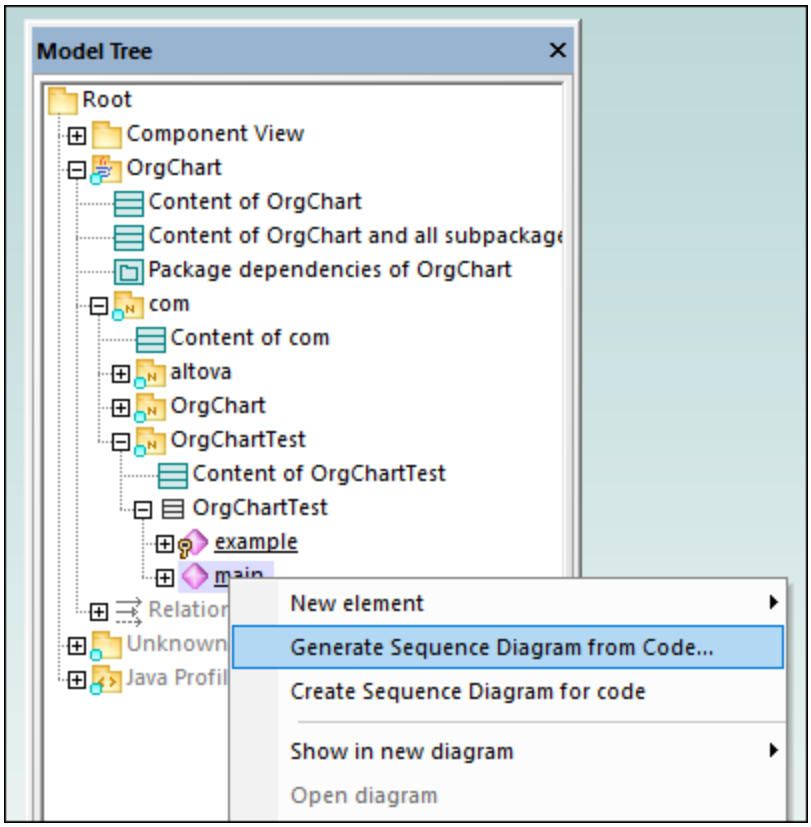
Message icons

- Message (Call)
- Message (Reply)
- Message (Creation)
- Message (Destruction)
- Asynchronous Message (Call)
- Asynchronous Message (Reply)
- Asynchronous Message (Destruction)
- Toggle dependent message movement
- Toggle automatic creation of replies for messages
- Toggle automatic creation of operations in target by typing operation names

8.1.7.2 Generate Sequence Diagrams from Source Code

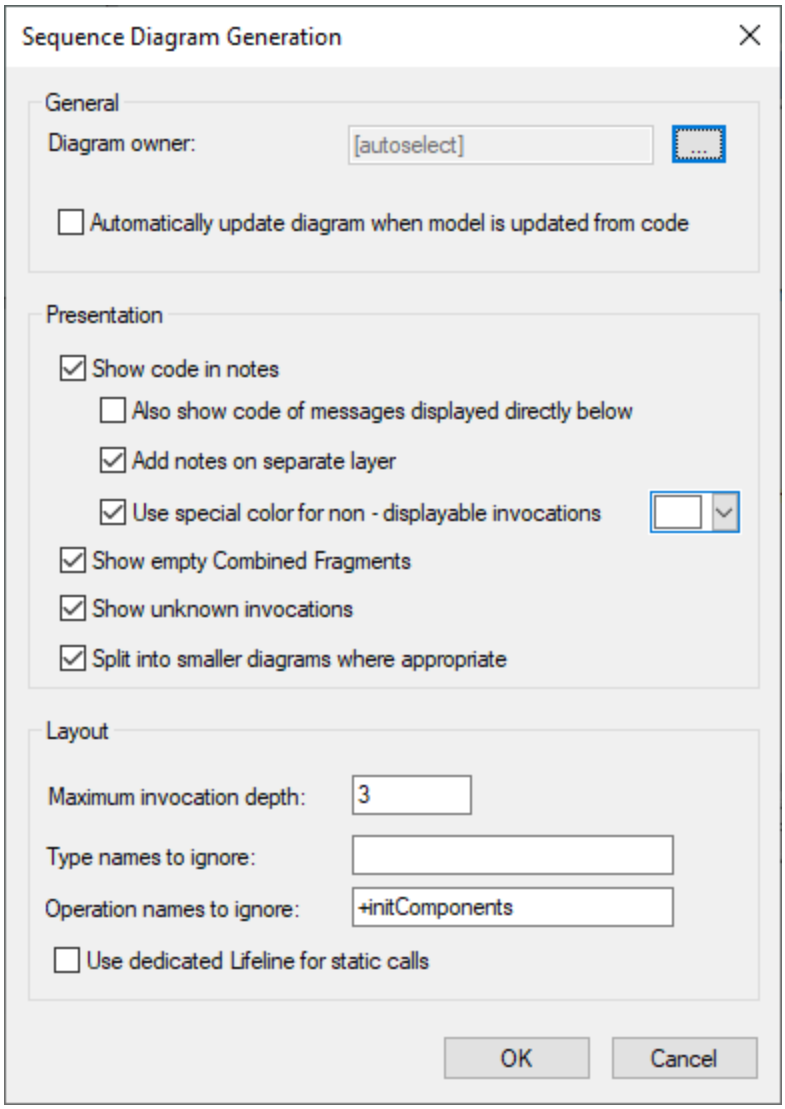
This example shows you how to generate a Sequence diagram from a method. The project containing this method will be reverse-engineered from Java source code. You can find the Java source code at the following path: **C:\Users\<user>\Documents\Altova\UModel2025\UModelExamples\OrgChart.zip**. First, unzip the **OrgChart.zip** archive to the same location (for example, right-click the archive in Windows Explorer and select **Extract All**).

1. On the **Project** menu, click **Import Source Directory**, and select the directory unzipped previously.
2. Go through the wizard steps to import the source code as a Java project. For more information about this step, see Reverse Engineering (from Code to Model)[73].
3. Having imported the code, right-click the `main` method of the `OrgChartTest` class in the Model Tree and select **Generate Sequence Diagram from Code...** from the context menu.

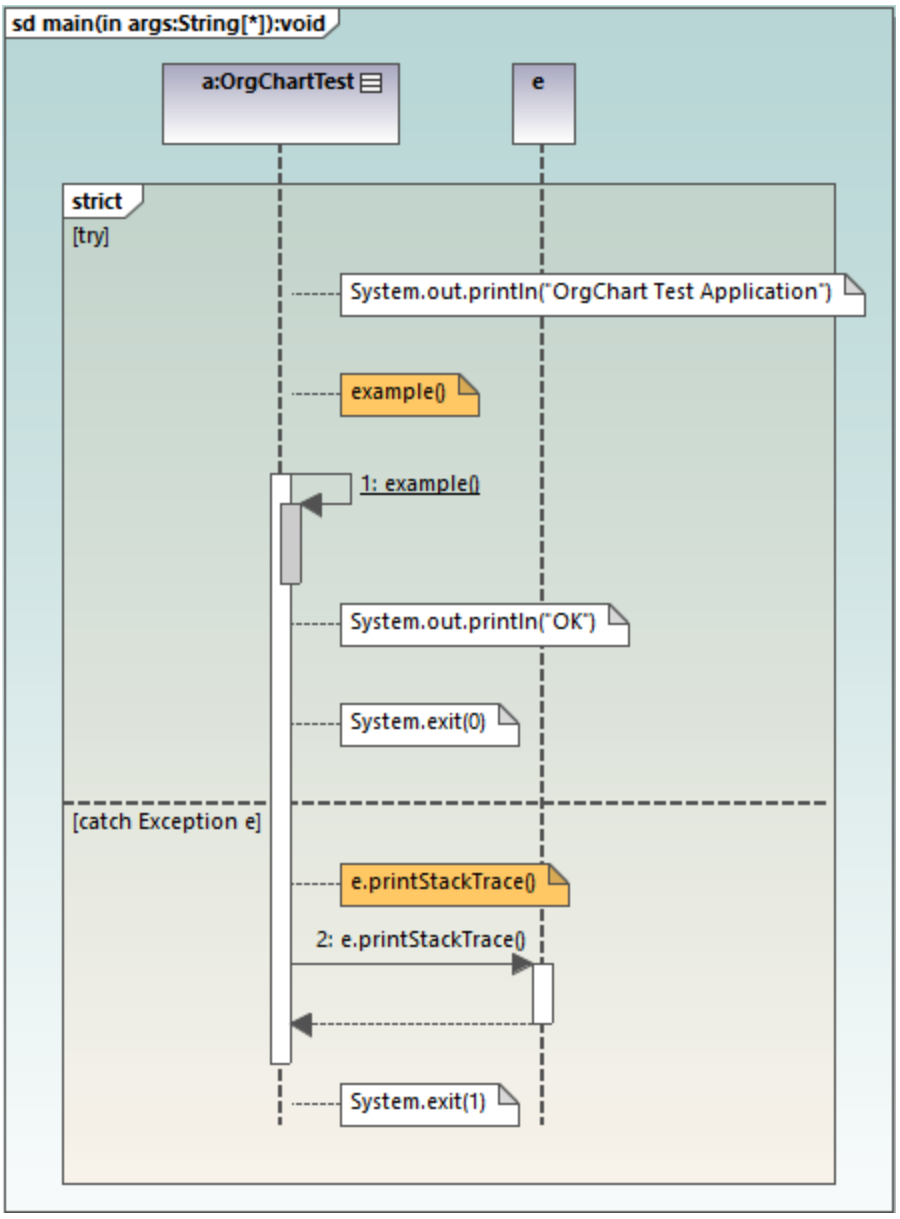


This opens the Sequence Diagram Generation dialog box in which you define the generation settings.

© 2019-2025 Altova GmbH



4. Select the presentation and layout options, and then click **OK** to generate the diagram. The settings shown above produce the sequence diagram below.

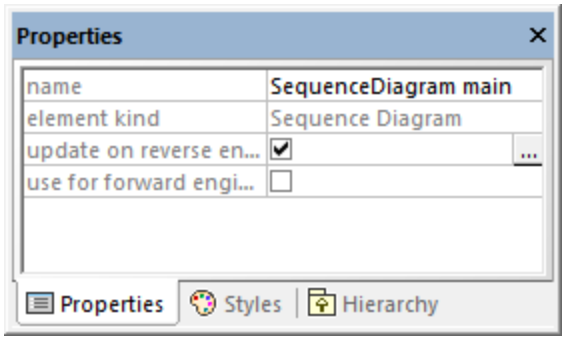


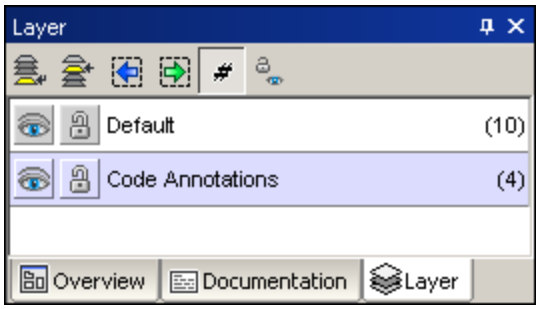
Sequence diagram generation options

The table below lists the generation options pertaining to Sequence diagrams.

Option	Purpose
Diagram owner	You can set this option when generating a diagram for the first time. For existing diagrams, this information is read-only.

© 2019-2025 Altova GmbH

Option	Purpose
	Click the **Ellipsis** button to select the owner package of the diagram. Otherwise, the option **[autoselect]** places the diagram in the default package.
Automatically update diagram when model is updated from code	When you perform reverse engineering (from code to model), sequence diagrams are re-generated automatically in the model, provided that you have selected the option **Automatically update diagram when model is updated from code** when generating the diagram for the first time. For existing diagrams, you can change this option as follows: 1. Select the Sequence diagram in the Model Tree or in the Diagram Tree. 2. In the Properties window, select the **update on reverse engineering** check box.  If you select the **use for forward engineering** check box, the synchronization from model to code will generate code based on the sequence diagram, when you perform forward engineering (from model to code), see also Generate Code from Sequence Diagram 372. If the two "engineering" check boxes are missing, it is likely that this diagram is just a fragment of a bigger diagram, or perhaps you have created the diagram from a non reverse-engineered operation.
Show code in notes	Select this check box to generate the diagram with notes (callouts) that contain program code.
Also show code of messages displayed directly below	Even when it is possible to show a piece of code as UML Message on the diagram, this option still displays the code of that message as a note.
Add notes on separate layer	Assigns code notes to a "Code Annotations" layer.

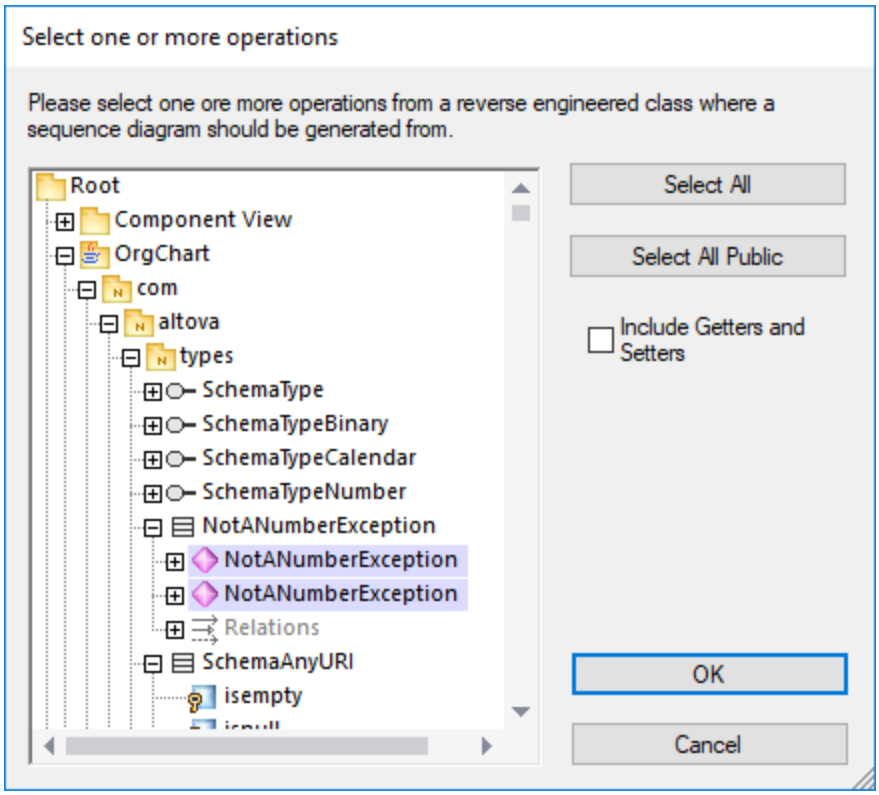
Option	Purpose
	
Use special color for non-displayable invocations	Assigns a color of your choice to non-displayable invocations.
Show empty Combined Fragments	Keeps the Combined Fragment 356 blocks on the diagram, even if they don't contain anything.
Shown unknown invocations	When selected, this option also displays messages for operations or constructors which could not be resolved (that is, not found in the model).
Split into smaller diagrams where appropriate	Automatically splits sequence diagrams into smaller sub-diagrams, and automatically generates hyperlinks between them for easy navigation.
Maximum invocation depth	Defines the call depth to be used in the diagram. For example, if `method1()` calls `method2()` which calls `method3()`, and the invocation depth is set to **2**, then only `method2` is shown, and `method3` is no longer shown.
Type names to ignore	Lets you define a comma delimited list of types that should not appear in the sequence diagram when it is generated.
Operation names to ignore	Lets you define a comma delimited list of operations that should not appear in the generated sequence diagram. Adding the operation names to the list causes the complete operation to be ignored. Prepending a "+" character to the operation in the list (for example, **+InitComponent**) causes the operation calls to be shown in the diagram, but without their content.
Use dedicated Lifeline for static calls	If there are static methods calls, and if there is already an instance of that object on the diagram, messages are normally drawn to that existing lifeline. With this option enabled, the diagram generator uses a dedicated new lifeline just for static method calls for that classifier.

© 2019-2025 Altova GmbH

8.1.7.2.1 Generate Multiple Sequence Diagrams

You can also create multiple sequence diagram models from multiple operations, as follows:

1. Select the menu option **Project | Generate Sequence diagrams from Code**.



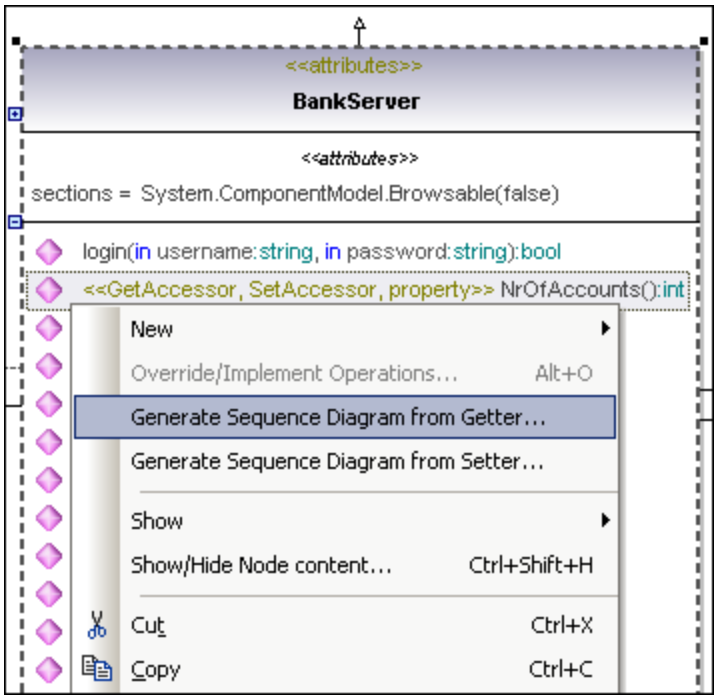
2. Select the operations that you want to generate a sequence diagram for and click **OK**. (Use the **Select All Public** and **Select All** buttons where necessary.)
3. Optionally, select the **Include Getters and Setters** check box to generate sequence diagrams for C#/VB.NET getters and setters.
4. Click **OK**. This opens a dialog box where you can specify the sequence diagram generation options 368 .
5. Click **OK**. A sequence diagram is generated for each selected operation, and UModel automatically opens it.

Creating multiple Sequence diagrams will likely take longer if your project is large. Note that only the first 10 diagrams will be opened automatically by UModel; all the rest will be generated without being opened.

8.1.7.2.2 Generate Sequence Diagrams from Getters/Setters

You can also generate a sequence diagram from getter/setter properties (in C#, VB .NET), as follows:

1. Right-click an Operation with a `GetAccessor/SetAccessor` stereotype.



2. Select **Generate Sequence Diagram from Code (Getter/Setter)** from the context menu. This opens a dialog box where you can specify the sequence diagram generation options [368].
3. Click **OK** to generate the Sequence Diagram.

8.1.7.3 Generate Code from Sequence Diagram

UModel can create code from a sequence diagram which is linked to at least one operation. Code generation from sequence diagrams is available for:

- VB.NET, C# and Java
- UModel standalone, Eclipse, and Visual Studio editions
- All three UModel editions

Creating code from Sequence diagrams is possible by either:

- Starting from a reverse engineered operation, see Generating Sequence Diagrams from source code [366],
- By creating a **new** sequence diagram from scratch, which is linked to an operation, by right-clicking the operation (in the Model Tree) and selecting Create sequence diagram for code [375].

When using a reverse engineered sequence diagram as basis, ensure that the option "Show code in notes" is selected when reverse engineering the code, so you do not lose any code when you start the forward-engineering process again. This is due to the fact that UML is not able to display all the language features of VB.NET, Java and C# on the sequence diagram, and those code sections are therefore shown as code notes.

© 2019-2025 Altova GmbH

To add plain text as code when creating a sequence diagram:

1. Attach a note to a sequence diagram lifeline.
2. Type in the code which should be written into the final source code. Click the **Is Code** check box (in the **Properties** pane) for that note, to make it accessible.

See Adding code to sequence diagrams 375 for an example.

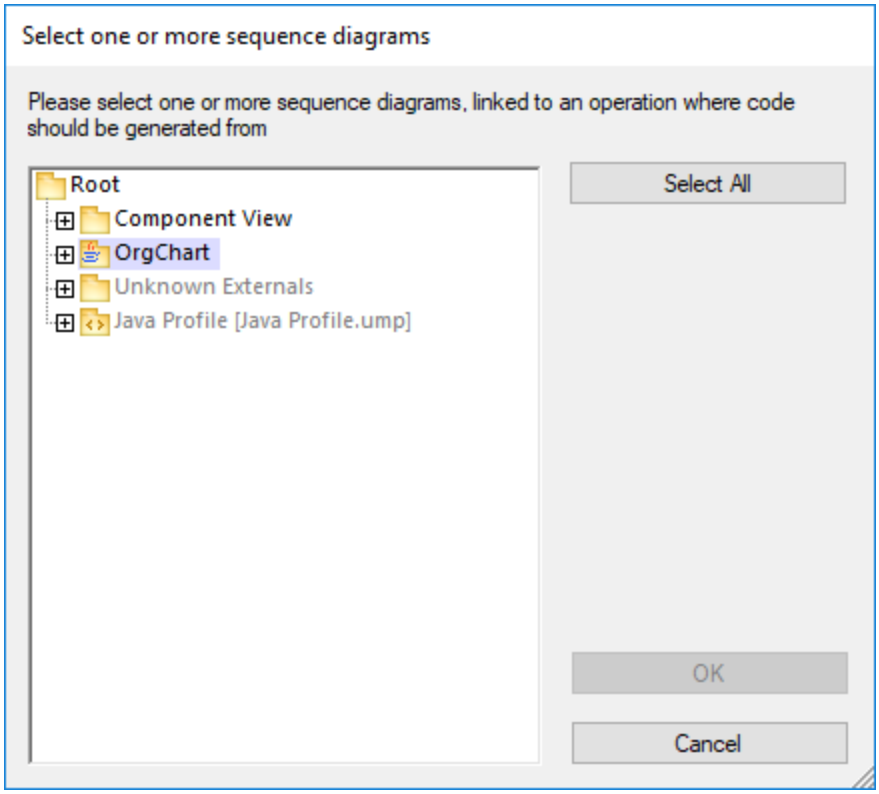
If a Sequence Diagram is to be used for code engineering automatically every time code engineering is started:

1. Select the diagram in the Model Tree or Diagram Tree window.
2. Select the **Use for forward engineering** check box in the **Properties** window.

Old code will always be lost when forward engineering code from a sequence diagram, because it will be overwritten with the new code.

To generate code using the Project menu:

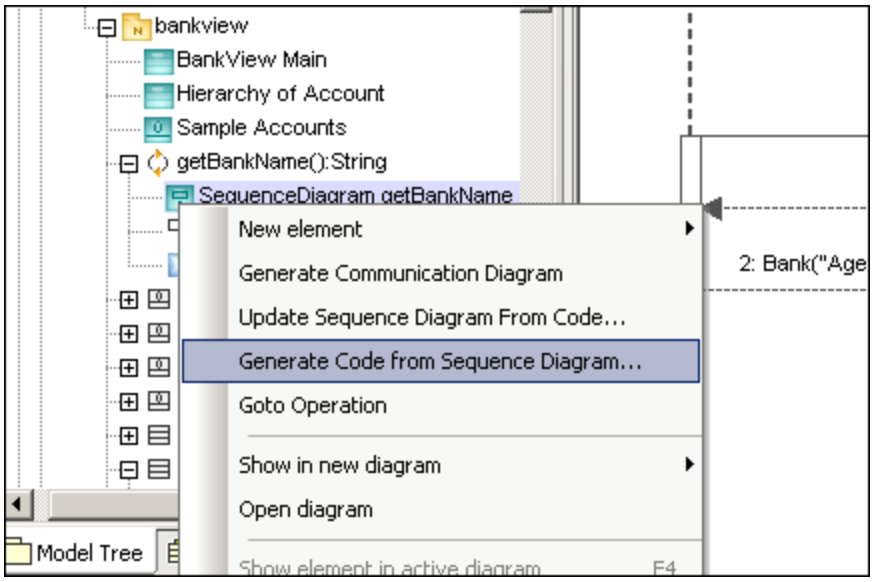
1. Select the menu option **Project | Generate Code from Sequence Diagrams**. You are now prompted to select the specific Sequence Diagram(s). Clicking the "Select All" button selects all the Sequence Diagrams in the UModel project.



2. Click OK to generate the code. The Messages window shows the status of the code generation process.

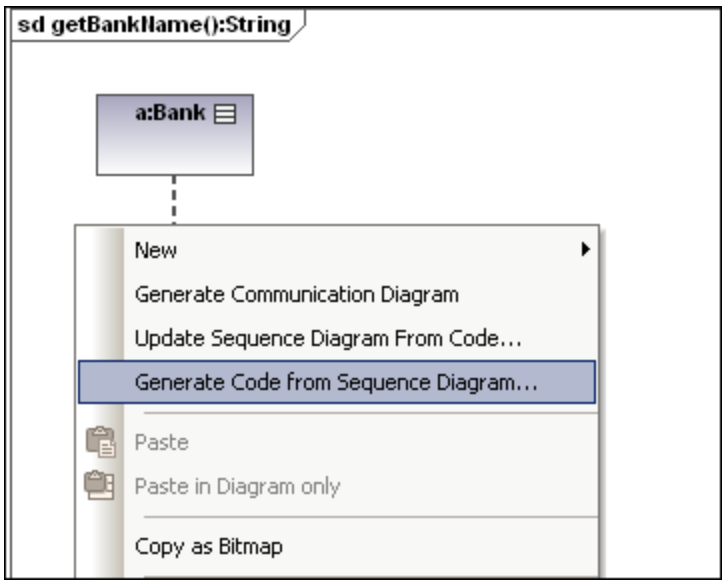
To generate code using the Model Tree:

- Right click a Sequence Diagram and select **Generate Code from Sequence diagram**.



To generate a Sequence Diagram containing code of an operation:

1. Click into the empty space of the Sequence Diagram, that contains code of an operation.
2. Select **Generate Code from Sequence diagram**.

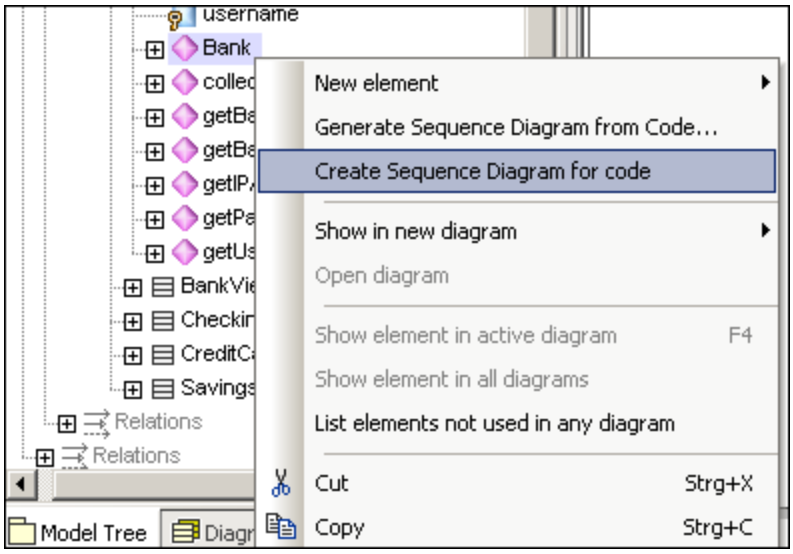


This command starts the forward-engineering process at this point.

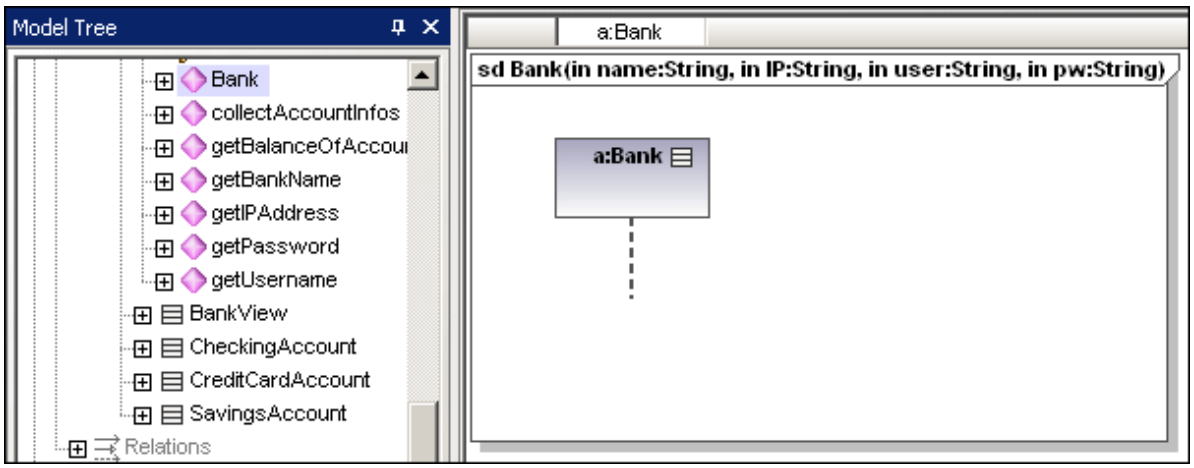
© 2019-2025 Altova GmbH

To create a Sequence diagram for code (engineering):

- In the Model Tree, right-click an operation and select **Create Sequence diagram for code**.



You will then be prompted if you want to use the new diagram for forward engineering.



The result is a new Sequence Diagram containing the lifeline of that class.

8.1.7.3.1 Adding code to sequence diagrams

Program code can be generated from new, and reverse-engineered sequence diagrams, but only for a sequence diagram linked to the "main operation".

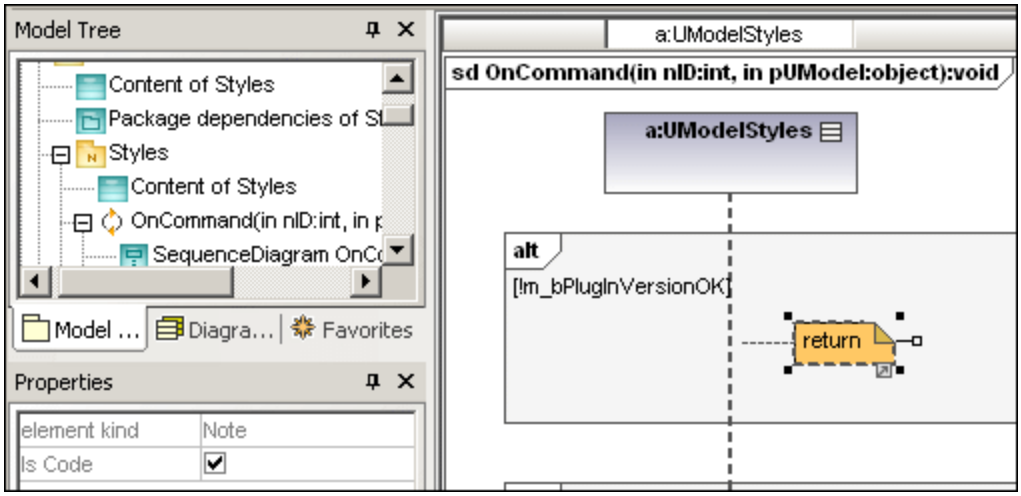
When reverse-engineering code, standard sequence diagram elements, e.g. CombinedFragments, are "mapped/assigned" to coding elements (e.g. "if" statements, loops, etc.).

For those programming statements that have no corresponding sequence diagram elements, e.g. "i = i+1", UModel makes use of "code" notes to add code to diagrams. These notes must then be linked to the lifeline.

Note that UModel does not check, or parse, these code fragments. It is up to you to make sure that the code fragments are correct and will compile.

To add code to a sequence diagram:

1. Click the **Note** icon then click the model element where you want to insert it, e.g. CombinedFragment.
2. Enter the code fragment, e.g. return.
3. Click the Node Link handle of the inserted note and drop the cursor on the lifeline.
4. Activate the "Is Code" check box in the Properties tab to include this code fragment when generating code.



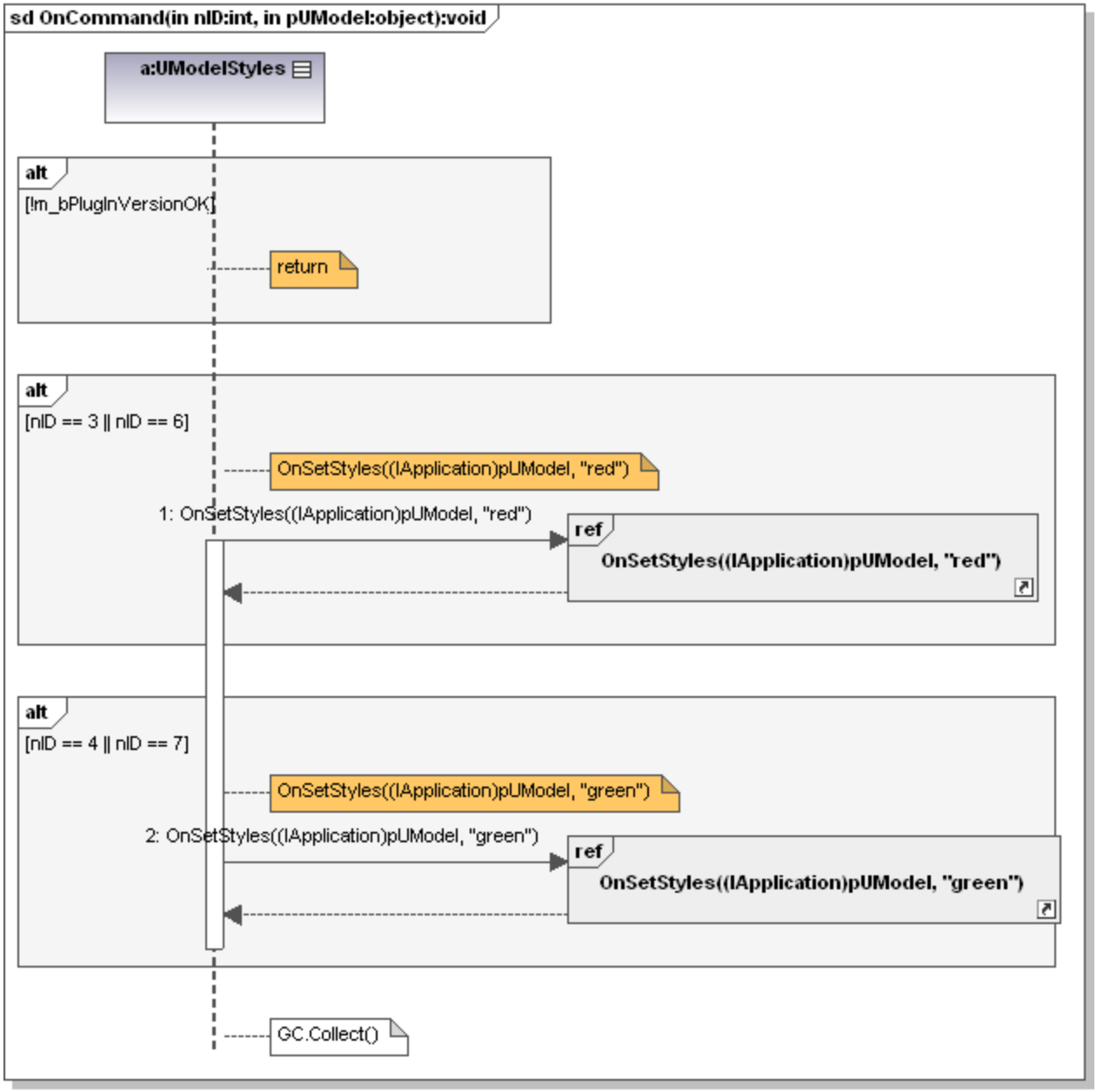
When selecting a note on a sequence diagram, which **can** be used for code generation, the property "is code" is available in the Properties window. Clicking the check box, allows you to switch between "ordinary" notes and code generation notes.

Ordinary notes:	
Code generation notes	- shown with a darker dog-ear

Code updates occur automatically on every forward engineering process if the "Use for forward engineering" check box is active. If changes were made to the sequence diagram, the code of the operation is always overwritten.

The sequence diagram shown below was generated by right clicking the **OnCommand** operation and selecting **Generate sequence diagram from code**. The C# code of this example is available in the **C:\Users\<user>\Documents\Altova\UModel2025\UModelExamples\IDEPlugIn\Styles** folder. Use the option **Project | Import Source Project**, to import the project.

© 2019-2025 Altova GmbH



The code shown below is generated from the sequence diagram.

```
Public void OnCommand(int nID, object pUModel)
{
   //Generated by UModel. This code will be overwritten when you re-run code generation.

   if (!m_bPlugINVersionOK)
   {
   return;
   }
```

```
    if (nID == 3 || nID == 6)
    {
    OnSetStyles((IApplication)pUModel, "red");
    }

    if (nID == 4 || nID == 7)
    {
    OnSetStyles((IApplication)pUModel, "green");
    }
    GC.Collect();

}
```

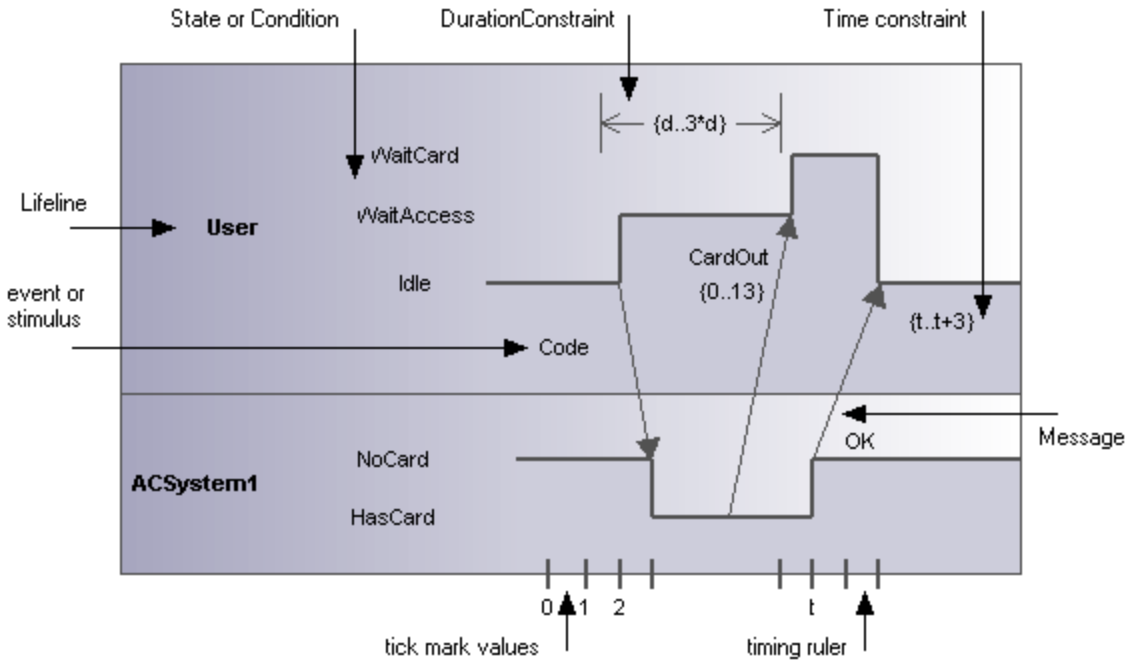
8.1.8 Timing Diagram

Altova website: UML Timing diagrams

Timing diagrams depict the changes in state, or condition, of one or more interacting objects over a given period of time. States, or conditions, are displayed as timelines responding to message events, where a lifeline represents a Classifier Instance or Classifier Role.

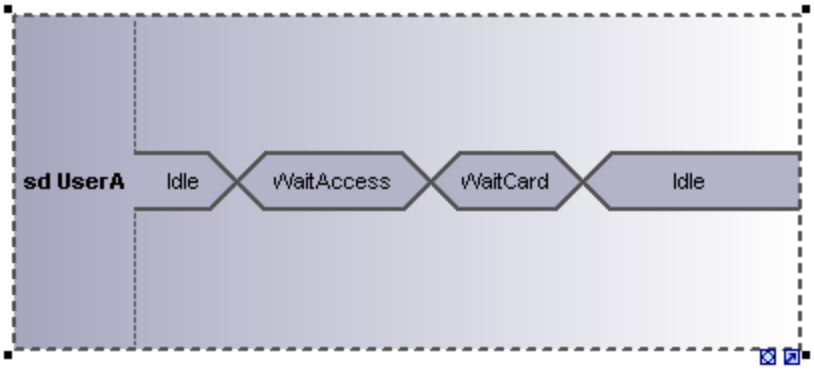
A Timing diagram is a special form of a sequence diagram. The difference is that the axes are reversed i.e. time increases from left to right, and lifelines are shown in separate vertically stacked compartments.

Timing diagrams are generally used when designing embedded software or real-time systems.



There are two different types of timing diagram: one containing the State/Condition timeline as shown above, and the other, the General value lifeline, shown below.

© 2019-2025 Altova GmbH



8.1.8.1 Inserting Timing Diagram elements

Using the toolbar icons

1. Click the specific timing icon in the Timing Diagram toolbar.

2. Click in the Timing Diagram to insert the element. To insert multiple elements of the selected type, hold down the **Ctrl** key and click in the diagram window.

Dragging existing elements into the timing machine diagram

Elements occurring in other diagrams, e.g. classes, can be inserted into an Timing Diagram.

1. Locate the element you want to insert in the **Model Tree** tab (you can use the search function text box, or press **Ctrl+F** to search for any element).
2. Drag the element(s) into the state diagram.

8.1.8.2 Lifeline

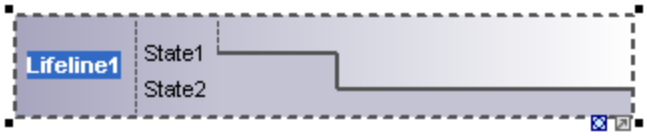
The **lifeline** element is an individual participant in an interaction, and is available in two different representations:

1. State/Condition lifeline
2. General Value lifeline

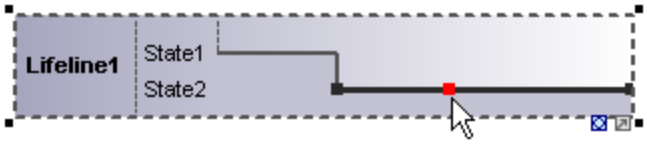
To create a **multiline** lifeline, press **Ctrl+Enter** to create a new line.

To insert a State Condition (StateInvariant) lifeline and define state changes:

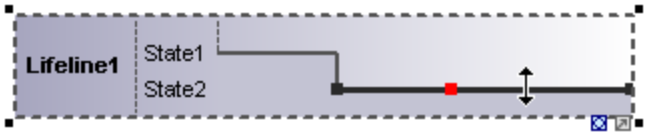
1. Click the **Lifeline (State/Condition)** icon in the title bar, then click in the Timing Diagram to insert it.



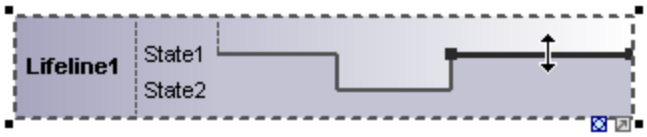
2. Enter the lifeline name to change it from the default name, Lifeline1, if necessary.
3. Place the mouse cursor over a section of one of the timelines and click left. This selects the line.
4. Move the mouse pointer to the position you want a state change to occur, and click again. Note that you will actually see the double headed arrow when you do this. A red box appears at the click position and divides the line at this point.



5. Move the cursor to the right hand side of the line and drag the line upwards.



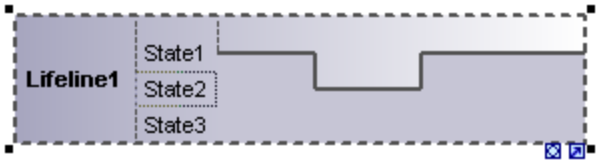
Note that lines can only be moved between existing states of the current lifeline.



Any number of state changes can be defined per lifeline. Once the red box appears on a line, clicking anywhere else in the diagram deletes it.

To add a new state to the lifeline:

- Right-click the lifeline and select **New | State/Condition (StateInvariant)**. A new State e.g. State3 is added to the lifeline.



© 2019-2025 Altova GmbH

To move a state within a lifeline:

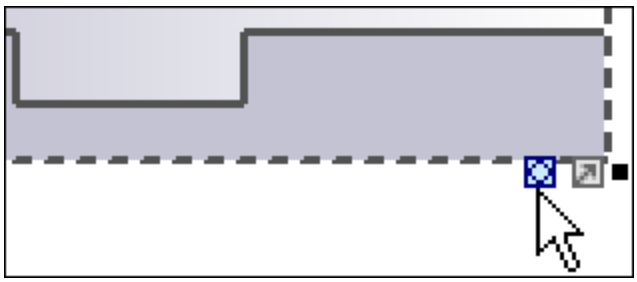
1. Click the state label that you want to move.
2. Drag it to a different position in the lifeline.

To delete a state from a lifeline:

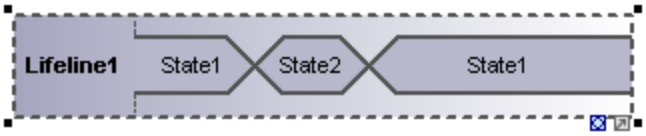
- Click the state and press the **Del.** key, or alternatively, right click and select **Delete**.

To switch between timing diagram types:

- Click the "toggle notation" icon at the bottom right of the lifeline.



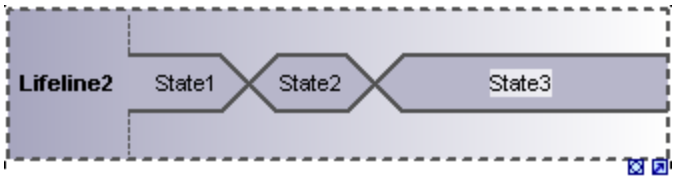
This changes the display to the General Value lifeline, the cross-over point represents a state/value change.



Note: Clicking the **Lifeline (General Value)** icon inserts the lifeline as shown above. You can switch between the two representations at any time.

To add a new state to the General value lifeline:

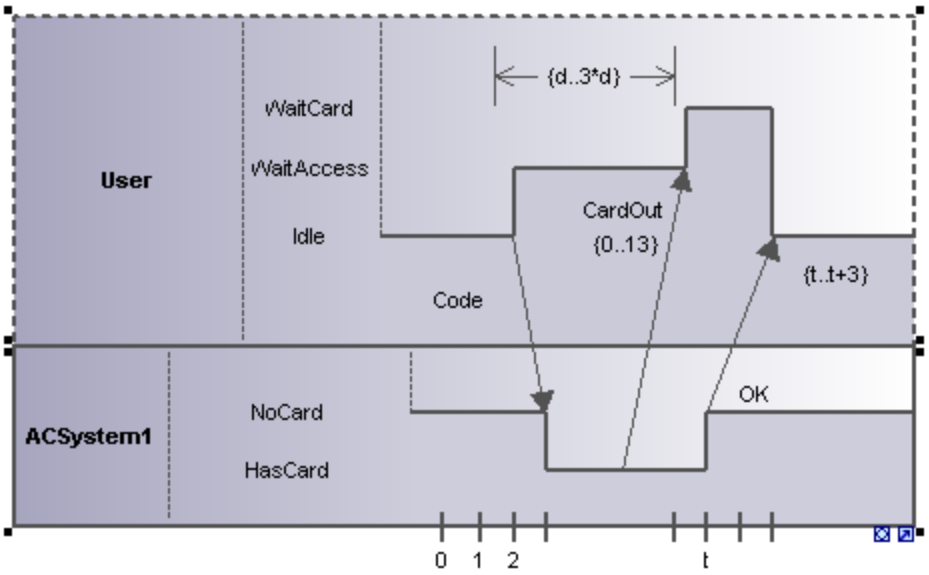
1. Right-click the lifeline and select **New | State/Condition (StateInvariant)**.
2. Edit the new name e.g. `State3`, and press **Enter** to confirm.



A new State is added to the lifeline.

Grouping lifelines

Placing or stacking lifelines automatically positions them correctly and preserves any tick marks that might have been added. Messages can also be created between separate lifelines by dragging the respective message object.

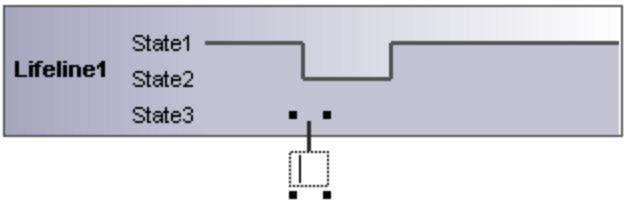


8.1.8.3 Tick Mark

The **TickMark** is used to insert the tick marks of a timing ruler scale onto a lifeline.

To insert a TickMark:

1. Click the tick mark icon and click on the lifeline to insert it.

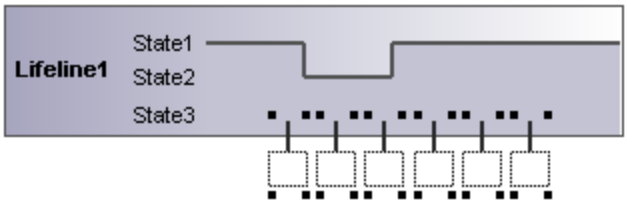


2. Insert multiple tick marks by holding down the **Ctrl** key and repeatedly clicking at different positions on the lifeline border.
3. Enter the tick mark label in the field provided for it. Drag tick marks to reposition them on the lifeline.

© 2019-2025 Altova GmbH

To evenly space tick marks on a lifeline:

1. Use the marquee, by dragging in the main window, to mark the individual tick marks.
2. Click the **Space Across** icon in the icon bar.

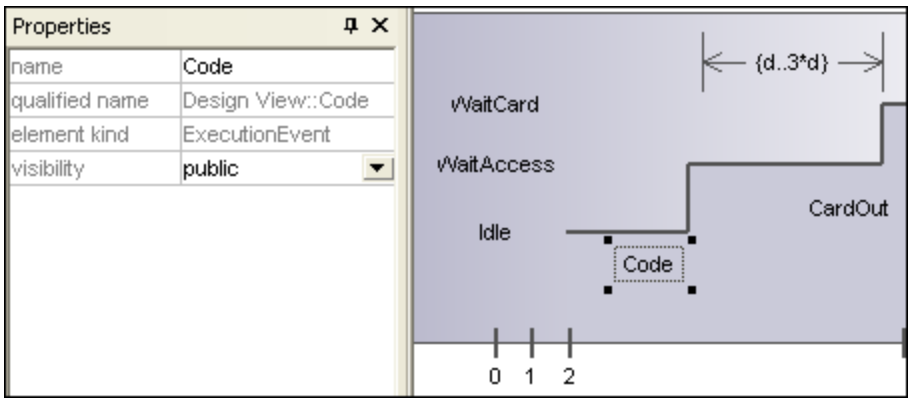


8.1.8.4 Event/Stimulus

The **Event/Stimulus** ExecutionEvent is used to show the change in state of an object caused by the respective event or stimulus. The received events are annotated to show the event causing the change in condition or state.

To insert an Event/Stimulus:

1. Click the Event/Stimulus icon, then click the specific position in the timeline where the state change takes place.



2. Enter a name for the event, in this example the event is "Code".

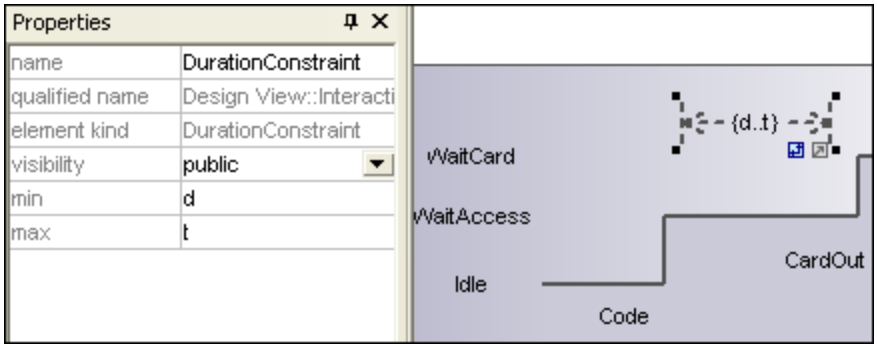
Note that the event properties are visible in the Properties tab.

8.1.8.5 DurationConstraint

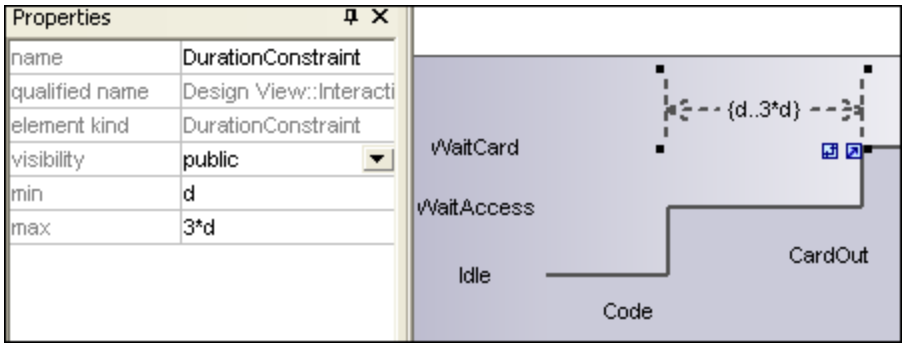
A **DurationConstraint** defines a ValueSpecification that denotes a duration in time between a start and endpoint. A duration is often an expression representing the number of clock ticks, which may elapse during this duration.

To insert an DurationConstraint:

1. Click the **DurationConstraint** icon, then click the specific position on the lifeline where the constraint is to be displayed. The default minimum and maximum values, "d..t", are automatically supplied. These values can be edited by double clicking the time constraint, or by editing the values in the Properties window.



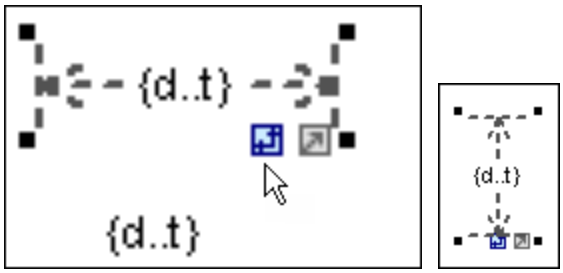
2. Use the handles to resize the object if necessary.



To change the orientation of the DurationConstraint:

- Click the "Flip" icon to orient the constraint vertically.

© 2019-2025 Altova GmbH

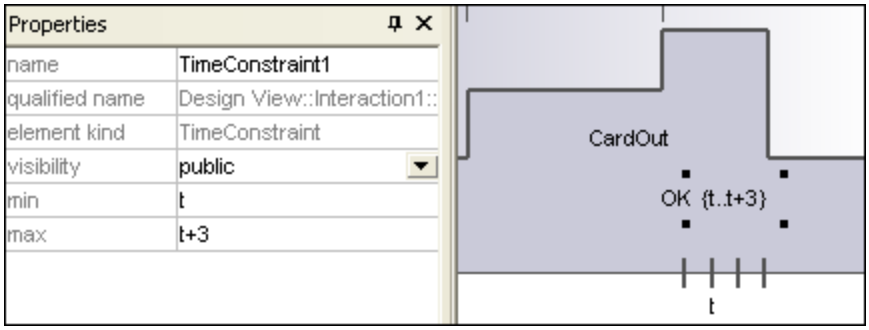


8.1.8.6 TimeConstraint

A **TimeConstraint** is generally shown as graphical association between a TimeInterval and the construct that it constrains. Typically, this is graphical association between an EventOccurrence and a TimeInterval.

To insert a TimeConstraint:

- Click the **TimeConstraint** icon, then click the specific position on the lifeline where the constraint is to be displayed.



The default minimum and maximum values are automatically supplied, "d..t" respectively. These values can be edited by double clicking the time constraint, or by editing the values in the Properties window.

8.1.8.7 Message

A Message is a modeling element that defines a specific kind of communication in an Interaction. A communication can be e.g. raising a signal, invoking an Operation, creating or destroying an Instance. The Message specifies the type of communication defined by the dispatching ExecutionSpecification, as well as the sender and the receiver.

Use the following toolbar buttons to add specific message types:

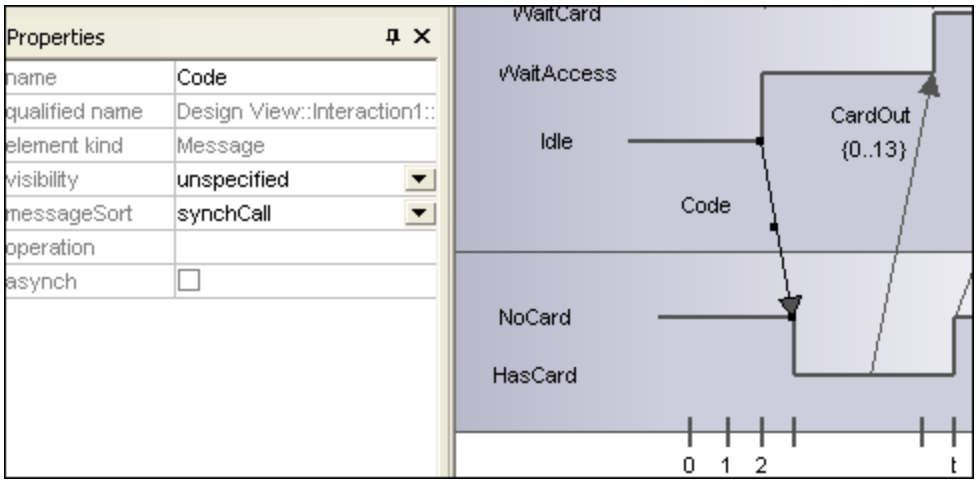
Message (Call)
Message (Reply)

Async message (Call)

Messages are sent between sender and receiver timelines, and are shown as labeled arrows.

To insert a message:

1. Click the specific message icon in the toolbar.
2. Click anywhere on the timeline sender object e.g. **Idle**.
3. Drag and drop the message line onto the receiver objects timeline e.g. **NoCard**. Lifelines are highlighted when the message can be dropped.



Notes:

- The direction in which you drag the arrow defines the message direction. Reply messages can point in either direction.
- Having clicked a message icon and holding down **Ctrl** key, allows you to insert multiple messages by repeatedly clicking and dragging in the diagram tab.

To delete a message:

1. Click the specific message to select it.
2. Press the **Del** key to delete it from the model, or right click it and select "Delete from diagram".

© 2019-2025 Altova GmbH

8.2 Structural Diagrams

These diagrams depict the structural elements that make up a system or function. Both the static, e.g. Class diagram, and dynamic, e.g. Object diagram, relationships are presented.

Class Diagram

Component Diagram

Composite Structure Diagram

Deployment Diagram

Object Diagram

Package Diagram

Profile Diagram 411

8.2.1 Class Diagram

This section includes tasks and concepts applicable to Class Diagrams, as follows:

- Customizing Class Diagrams 387
- Overriding Base Class Operations and Implementing Interface Operations 394
- Creating Getter and Setter Methods 394
- Ball and Socket Notation 396
- Adding Raised Exceptions to Methods of a Class 397
- Adding Receptions to a Class 398
- Generating Class Diagrams 399

For a basic introduction to Class Diagrams, see Class Diagrams 31 in the tutorial section of this documentation.

8.2.1.1 Customizing Class Diagrams

Expanding / hiding class compartments in a UML diagram

There are several methods of expanding the various compartments of class diagrams.

- Click on the **+** or **-** buttons of the currently active class to expand/collapse the specific compartment.
- Use the marquee (drag on the diagram background) to mark **multiple** classes, then click the expand/hide button. You can also use **Ctrl+Click** to select multiple classes.
- Press **Ctrl+A** to select **all classes**, then click the expand/collapse button, on one of the classes, to expand/collapse the respective compartments.

Expanding / collapsing class compartments in the Model Tree

In the Model Tree classes are subelements of packages and you can affect either the packages or the classes.

- Click the package / class you want to **expand** and:
  - Press the ***** key to expand the current package/class and all sub-elements
  - Press the **+** key to open the current package/class.

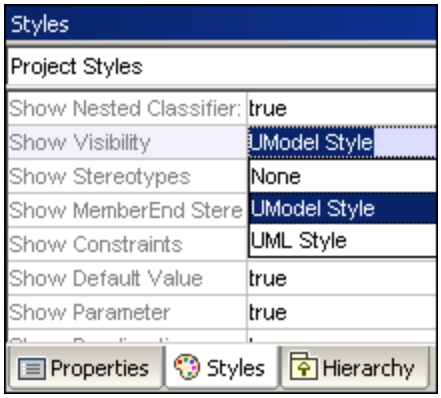
To **collapse** the packages/classes, press the **-** keyboard key.

Note that you can use the standard keyboard keys, or the numeric keypad keys to achieve this.

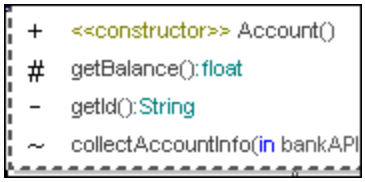
Changing the visibility type icons

Clicking the **visibility icon** to the left of an operation, or property, opens a drop-down list enabling you to change the visibility status. You can also change the type of visibility symbols that you want to see.

- Click a class in the diagram window, click the **Styles** tab and scroll down the list until you find the **Show Visibility** entry.



You can choose between the UModel type shown above, or the UML conformant symbols shown below.



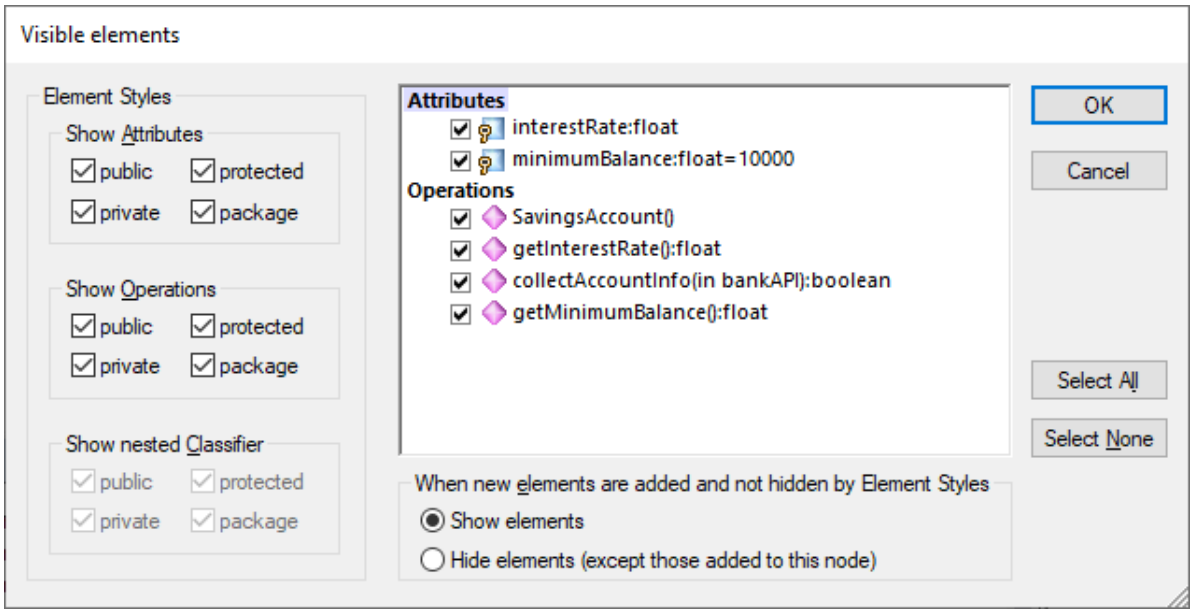
Showing or hiding node content (class attributes, operations, slots)

In class diagrams, you can show or hide specific members of a class, such as attributes or operations. You can show or hide not only individual members but also multiple members of the same type according to their visibility. For example, you can hide only those class attributes that have private visibility. Showing or hiding is also supported for object slots (`InstanceSpecifications`) in Object diagrams.

To show or hide class members or object slots:

1. Right-click a class (for example, `SavingsAccount` from the example **Bank_MultiLanguage.ump** project) and select **Show/Hide Node content** from the context menu.
2. Select or clear the check box next to the members you want to show or hide, respectively.

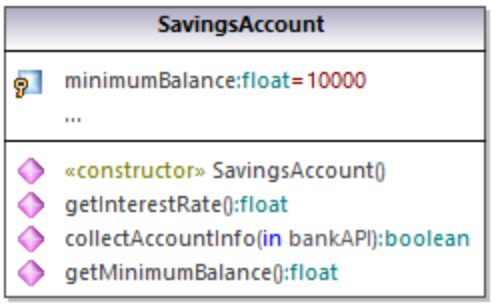
© 2019-2025 Altova GmbH



To show or hide multiple members based on their visibility, use the check boxes in the **Element Styles** group. For example, clearing the **protected** check box in the **Show Attributes** group hides all protected attributes of the class.

Note: Tagged values of hidden elements are also hidden when you select the hide option.

After you confirm your preferences with **OK** and close the dialog box, any hidden members on the diagram are replaced by the ellipsis `...` symbol. To open the dialog box again, double-click the ellipsis.



The **When new elements are added and not hidden by Element Styles** option allows you to define what will be made visible when new elements are added to the class. This applies not only to elements added manually in the diagram or in the Model Tree, but also to those added automatically during the code engineering process. The valid values for this option are as follows:

Show elements	When a new member is added to the class, show it on the diagram. Nevertheless, if any of the options set under "Element styles" dictate that the element must be hidden, hide it.

Hide elements (except those added to this node)	Here, the term "node" refers to the current instance of the class on the diagram. (Recall that the same class can be added multiple times on the same diagram, see Renaming, Moving, and Copying Elements 112.) When two or more instances of the same class exist on the diagram, and when a new member is added to *this instance* of the class, then hide the member in all instances of the class but show it for the current instance.

For an example of how the options above are useful, open the **Bank_MultiLanguage.ump** example project, and find the "Hierarchy of Account" class diagram.

Next, create a new instance of the `SavingsAccount` class, as follows:

1. Right-click the `SavingsAccount` class in the diagram and select **Copy**.
2. Right-click an empty area in the same diagram and select **Paste in this diagram only** from the context menu.

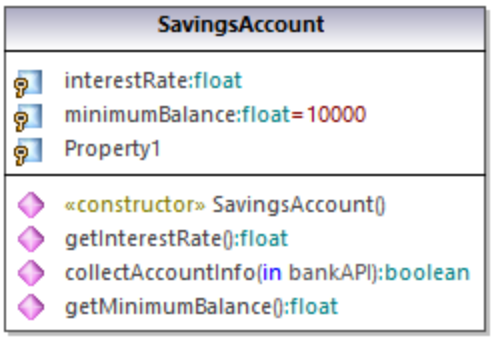
There are now two instances of the `SavingsAccount` class on the diagram.

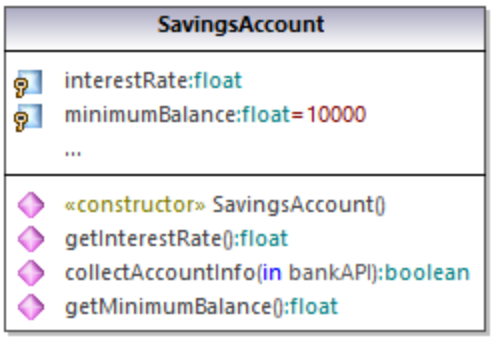
Next, set different visibility options in each of the instances:

1. Right-click the left instance of the class, select **Show/Hide Node content**, and then select the **Show elements** option.
2. Right-click the right instance of the class, select **Show/Hide Node content**, and then select the **Hide elements (except those added to this node)** option.

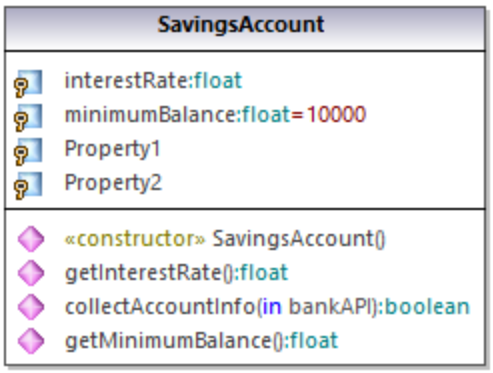
Next, add a new property to the left instance (select the class and press **F7**). As illustrated below, the new property (`Property1`) is visible in the left instance but not visible in the right instance. This happens because the right-side instance of the class has the the **Hide elements (except those added to this node)** option enabled.

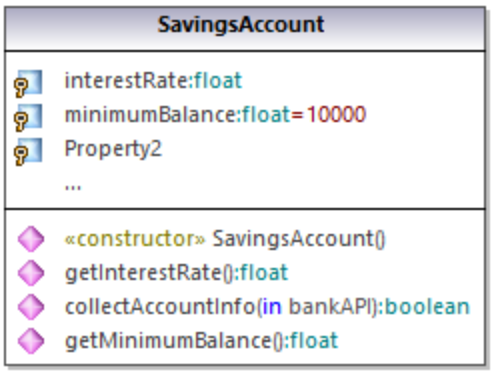
© 2019-2025 Altova GmbH





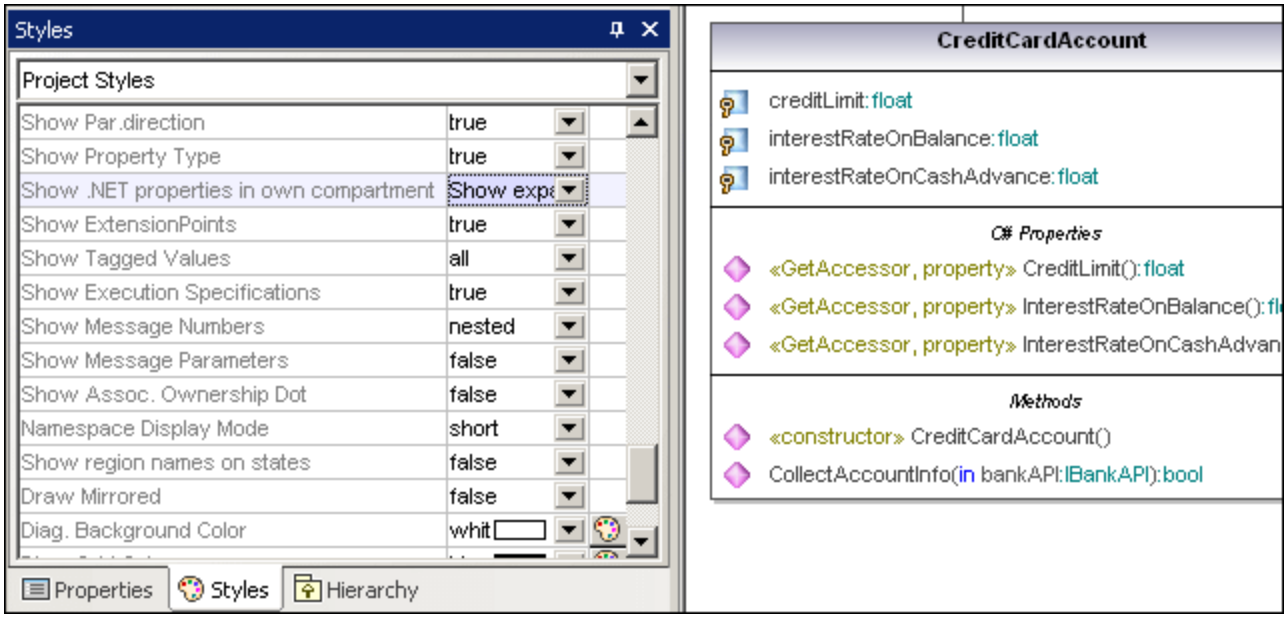
Finally, add a new property to the right-side instance of the class. As illustrated below, the new property (`Property2`) is visible in both instances. This happens because the left-side instance is configured to show new elements, while the right-side instance is the *current instance* where the property is added, so the new property is shown unconditionally.





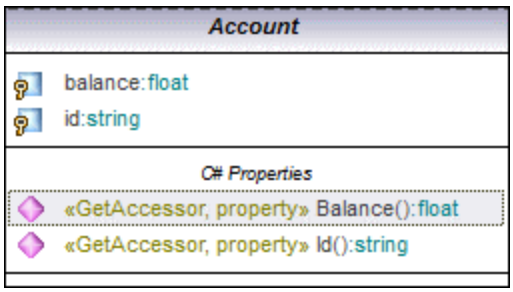
Showing or hiding .NET compartments

To display .NET properties in their own compartment, select the "Show .NET properties in own compartment" option in the **Styles** tab.



Showing .NET properties as associations

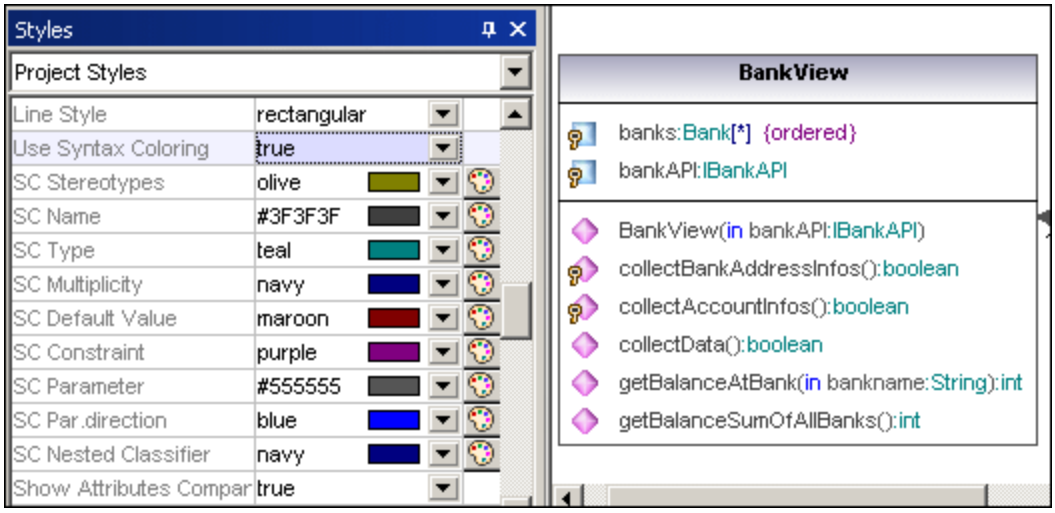
To display .NET properties as associations, right-click a C# property as shown below, and select **Show | All .NET Properties as Associations** from the context menu.



Changing the syntax coloring of operations/properties

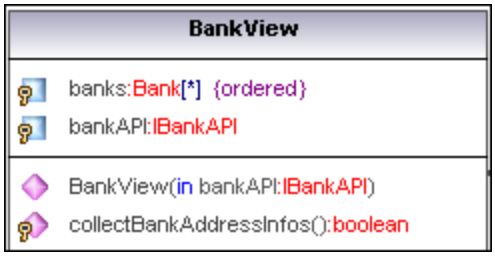
UModel automatically enables syntax coloring, but lets you customize it to suit your needs. The default settings are shown below.

© 2019-2025 Altova GmbH



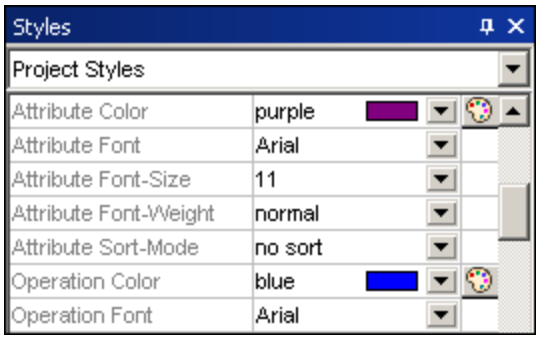
To change the default syntax coloring options (shown below):

1. Switch to the **Styles** tab and scroll the **SC** prefixed entries.
2. Change one of the "SC color" entries e.g. "SC Type" to "red".



To disable syntax coloring:

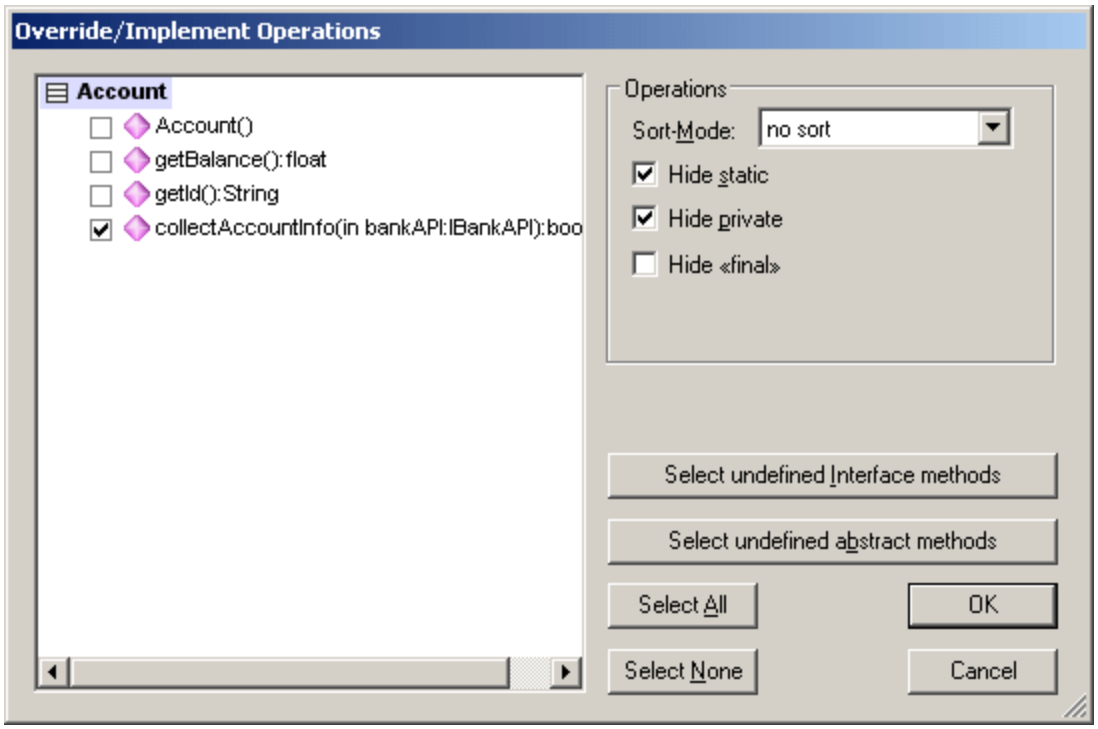
1. Switch to the **Styles** tab and change the **Use Syntax Coloring** entry to **false**.
2. Use the **Attribute Color**, or **Operation Color** entries in the **Styles** tab to customize these items in the class.



8.2.1.2 Overriding Base Class Operations and Implementing Interface Operations

UModel gives you the ability to override the base-class operations, or implement interface operations of a class. This can be done from the Model Tree, Favorites tab, or in Class diagrams.

1. Right-click one of the derived classes in the class diagram, e.g. `CheckingAccount`, and select **Override/Implement Operations**. This opens the dialog box shown below.



2. Select the Operations that you want to override and confirm with **OK**. The "Select undefined..." buttons select those method types in the window at left.

Note: When the dialog box is opened, operations of base classes and implemented interfaces that have the same signature as existing operations, are automatically checked (i.e. active).

8.2.1.3 Creating Getter and Setter Methods

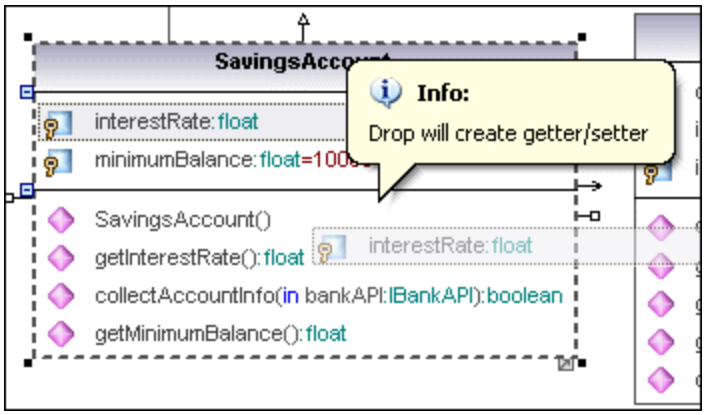
During the modeling process it is often necessary to create get/set methods for existing attributes. UModel supplies you with two separate methods to achieve this:

- Drag and drop an attribute into the operation compartment
- Use the context menu to open a dialog box allowing you to manage get/set methods

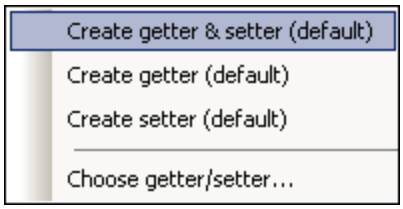
© 2019-2025 Altova GmbH

To create getter/setter methods using drag and drop:

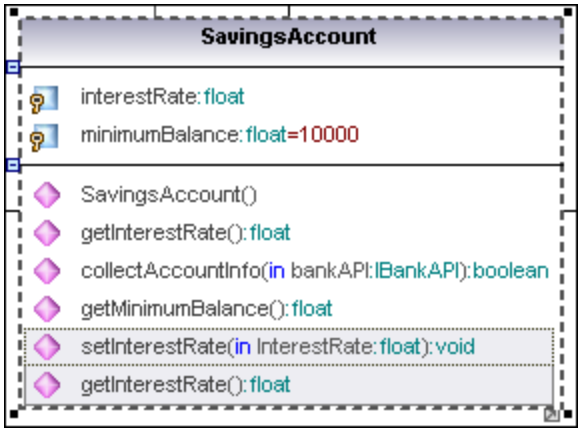
- Drag an attribute from the Attribute compartment and drop it in the Operations compartment.



A pop-up menu appears at this point allowing you to decide what type of get/set method you want to create.

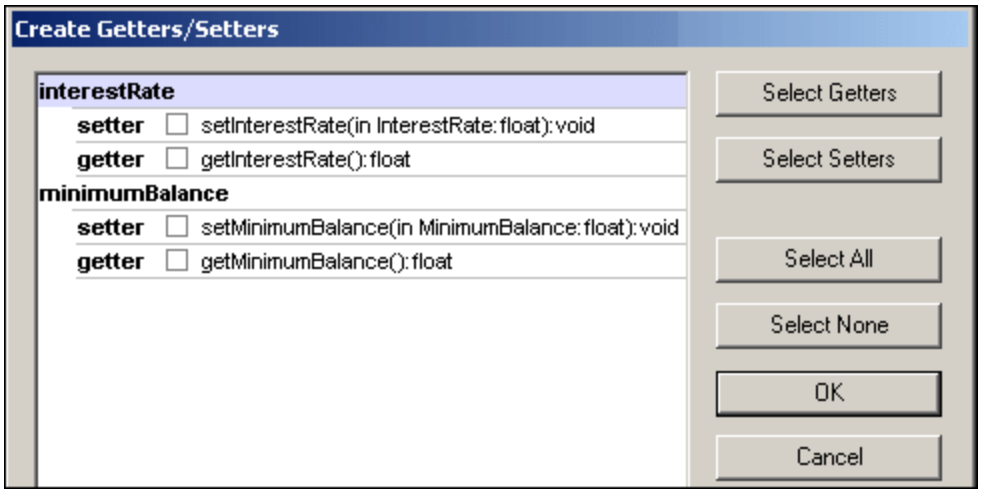


Selecting the first item creates a get and set method for interestRate:float.



To create getter/setter methods using the context menu:

1. Right-click the class title, e.g. `SavingsAccount`, and select the context menu option **Create Getter/Setter Operations**. The Create Getters/Setters dialog box opens displaying all attributes available in the currently active class.

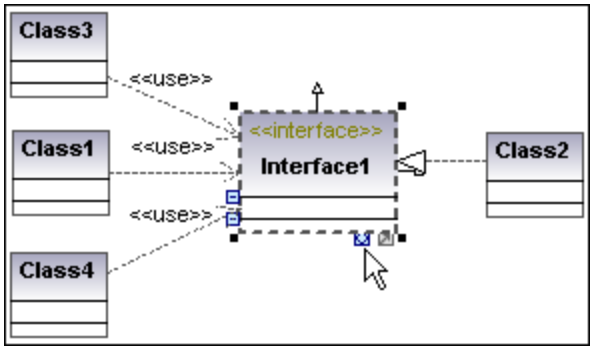


2. Use the buttons to select the items as a group, or click the getter/setter check boxes individually.

Note: You can also right-click a single attribute and use the same method to create an operation for it.

8.2.1.4 Ball and Socket Notation

UModel supports the ball and socket notation of UML. Classes that require an interface display a "socket" and the interface name, while classes that implement an interface display the "ball".

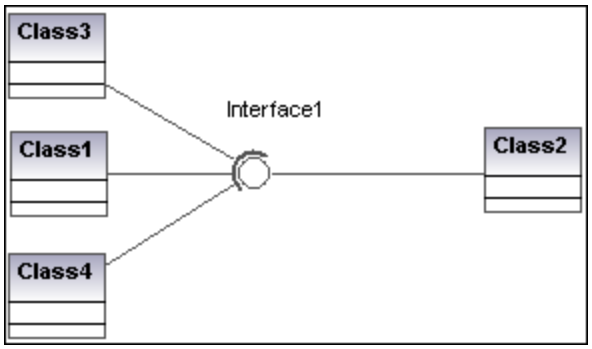


In the shots shown above, Class2 realizes Interface1, which is used by classes 1, 3, and 4. The usage icons were used to create the usage relationship between the classes and the interface.

To switch between the standard and ball-and-socket view:

- Click the Toggle Interface notation icon at the base of the interface element.

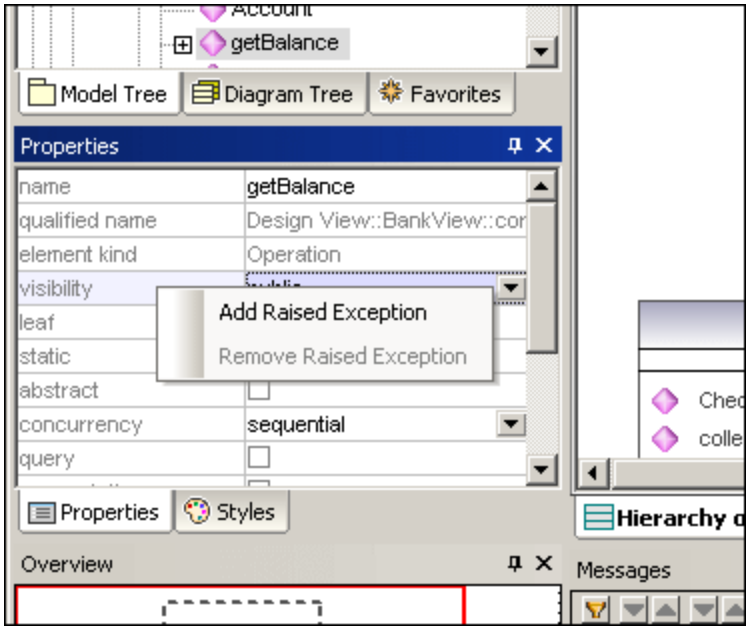
© 2019-2025 Altova GmbH



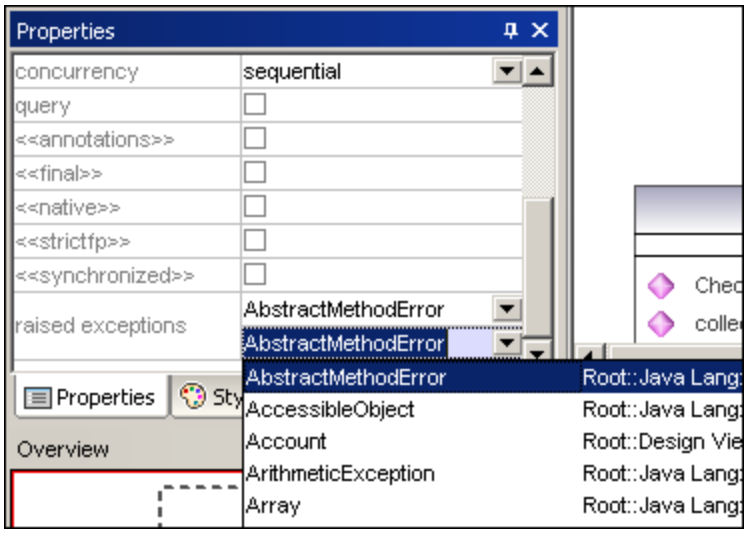
8.2.1.5 Adding Raised Exceptions to Methods of a Class

To add raised Exceptions to methods of a class:

1. Click the method of the class you want to add the raised exception to in the Model Tree window, e.g. `getBalance` of the `Account` class.
2. Right-click the Properties window and select **Add Raised Exception** from the pop-up menu. This adds the **raised exceptions** field to the Properties window, and automatically selects the first entry in the list.



3. Select an entry from the list, or enter your own into the field.



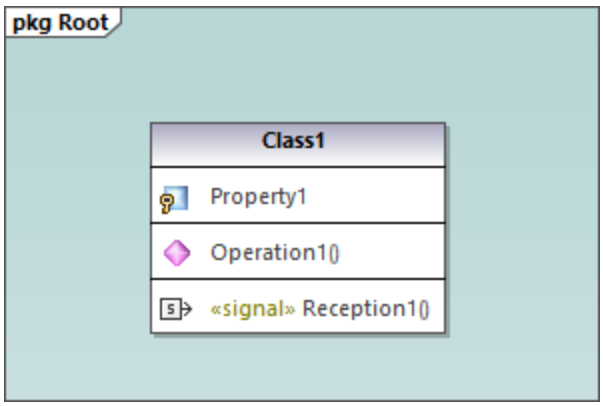
8.2.1.6 Adding Receptions to a Class

In addition to operations and properties, you can add Reception elements to a class.

To add a Reception to a class:

- Right-click the class on the diagram and select **New | Reception** from the context menu.

Receptions appear in a separate compartment on the Class diagram, similar to properties and operations, for example:



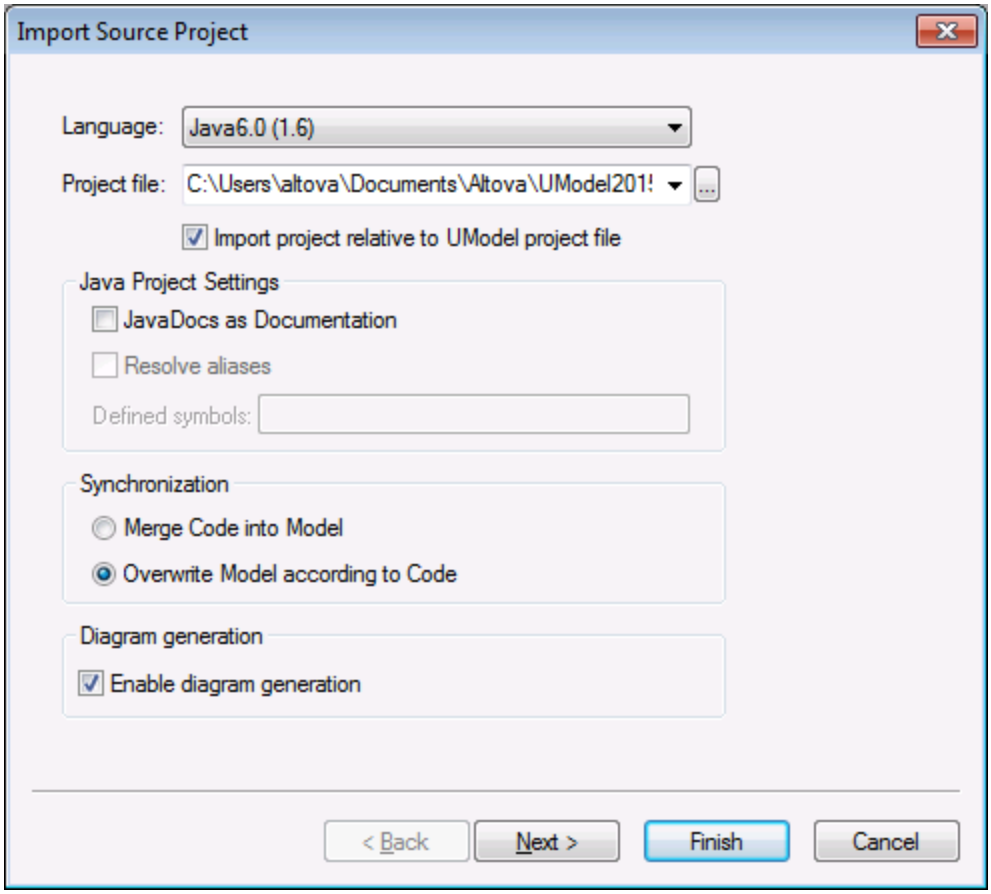
Receptions share the same styles as operations. This means that, whenever you change the style of operations, the changes affect Receptions also. For more information, see Changing the Style of Elements 122.

© 2019-2025 Altova GmbH

8.2.1.7 Generating Class Diagrams

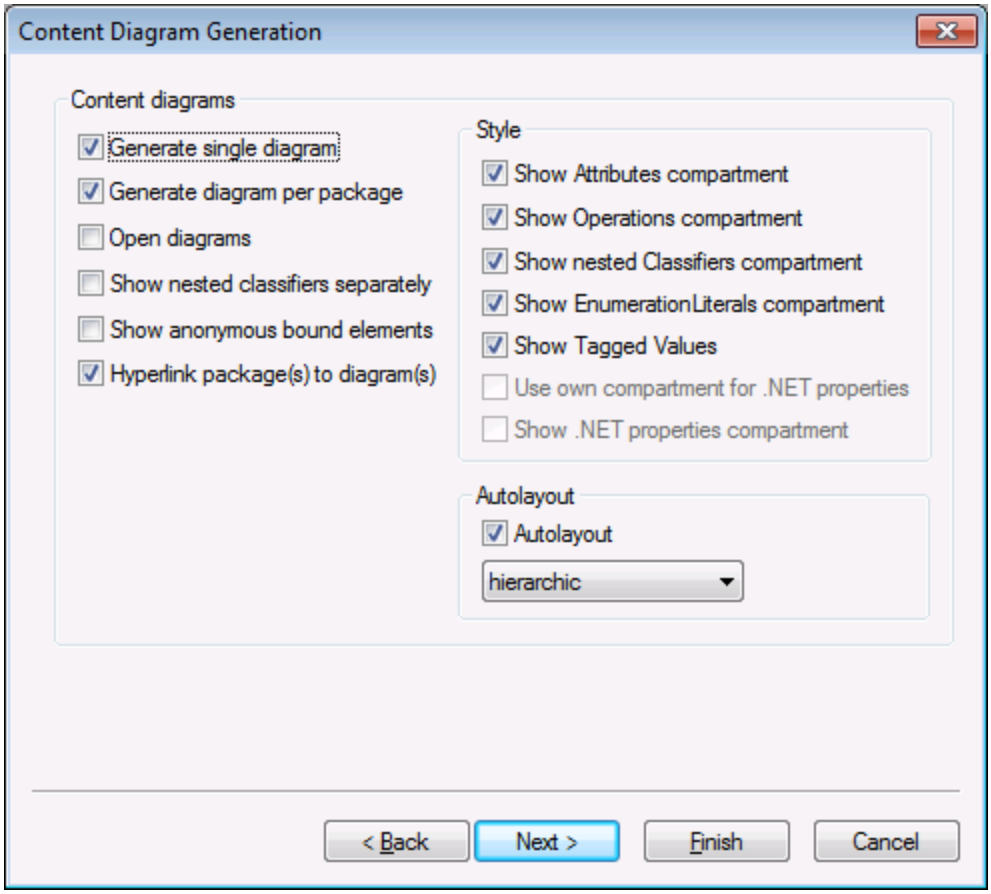
As an alternative to designing class diagrams directly in UModel, you can generate them automatically when importing source code or binaries into UModel projects (see Importing Source Code 192 and Importing Java, C# and VB.NET Binaries 204). When following the import wizard, make sure that:

1) The **Enable diagram generation** check box is selected on the "Import Source Project", "Import Binary Types", or "Import Source Directory" dialog box.



Import Source Project dialog box

2) The **Generate single diagram** and/or the **Generate diagram per package** options are selected on the "Content Diagram Generation" dialog box.



Content Diagram Generation dialog box

Once the import operation is finished, any generated class diagrams are available under "Class Diagrams" in the Diagram Tree.

© 2019-2025 Altova GmbH

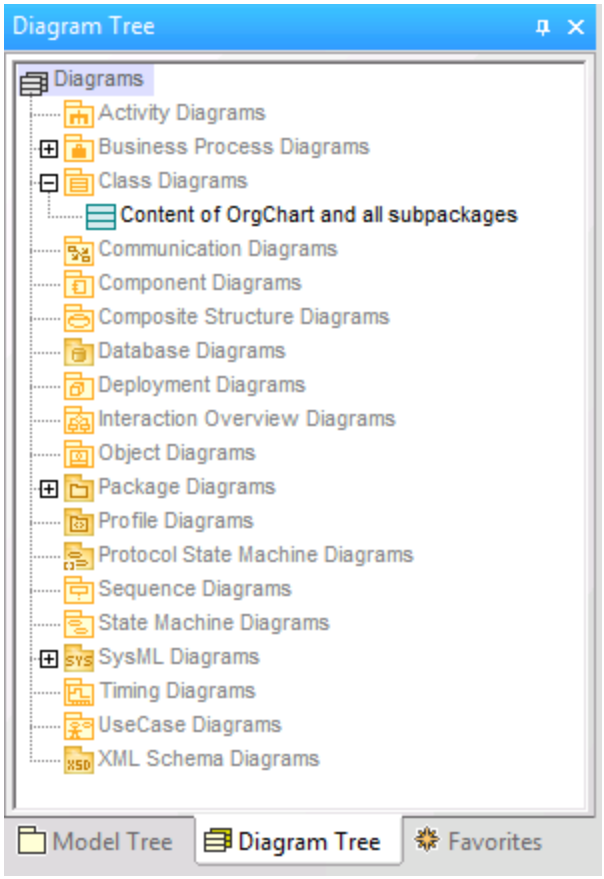
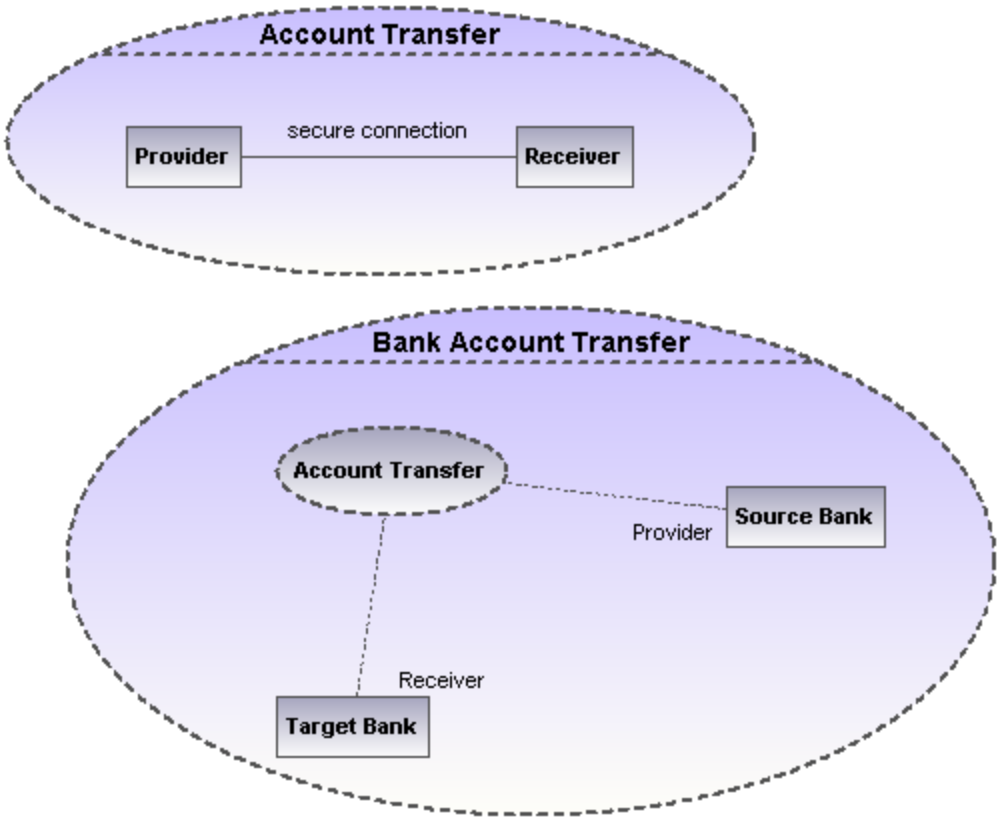


Diagram Tree

8.2.2 Composite Structure Diagram

Altova website: UML Composite Structure diagrams

The Composite Structure Diagram has been added in UML 2.0 and is used to show the internal structure, including parts, ports and connectors, of a structured classifier, or collaboration.



8.2.2.1 Inserting Composite Structure Diagram elements

Using the toolbar icons

1. Click the specific Composite Structure diagram icon in the toolbar.

2. Click in the Composite Structure diagram to insert the element. To insert multiple elements of the selected type, hold down the **Ctrl** key and click in the diagram window.

Dragging existing elements into the Composite Structure diagram

Most elements occurring in other Composite Structure diagrams, can be inserted into an existing Composite Structure diagram.

1. Locate the element you want to insert in the Model Tree tab (you can use the search function text box, or press **Ctrl+F** to search for any element).
2. Drag the element(s) into the Composite Structure diagram.

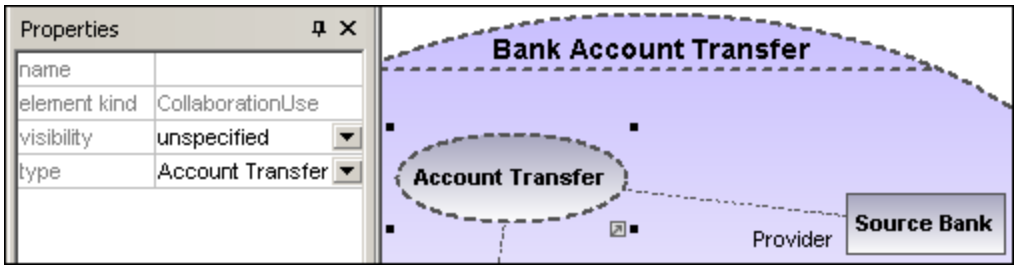
© 2019-2025 Altova GmbH

Collaboration

Inserts a collaboration element which is a kind of classifier/instance that communicates with other instances to produce the behavior of the system.

CollaborationUse

Inserts a Collaboration use element which represents one specific use of a collaboration involving specific classes or instances playing the role of the collaboration. A collaboration use is shown as a dashed ellipse containing the name of the occurrence, a colon, and the name of the collaboration type.



When creating dependencies between collaboration use elements, the "type" field must be filled to be able to create the role binding, and the target collaboration must have at least one part/role.

Part (Property)

Inserts a part element which represents a set of one or more instances that a containing classifier owns. A Part can be added to collaborations and classes.

Port

Inserts a port element which defines the interaction point between a classifier and its environment, and can be added on parts with a defined type.

Class

Inserts a Class element, which is the actual classifier that occurs in that particular use of the collaboration.

Connector

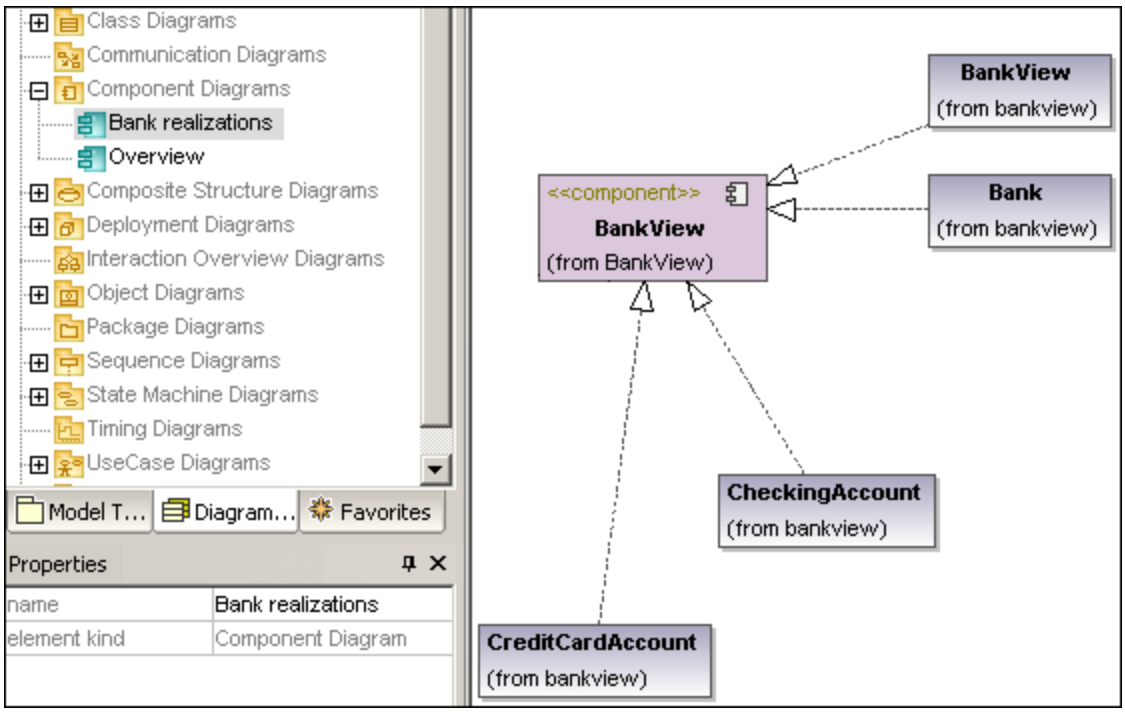
Inserts a Connector element which can be used to connect two or more instances of a part, or a port. The connector defines the relationship between the objects and identifies the communication between the roles.

Dependency (Role Binding)

Inserts the Dependency element, which indicates which connectable element of the classifier or operation, plays which role in the collaboration.

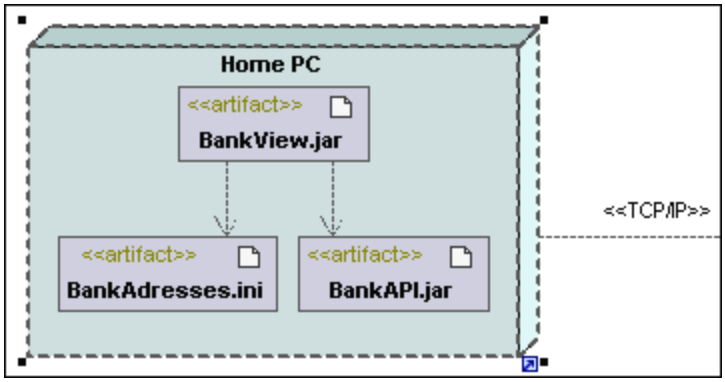
8.2.3 Component Diagram

Please see the Component Diagrams [53] section in the tutorial for more information on how to add component elements to the diagram.



8.2.4 Deployment Diagram

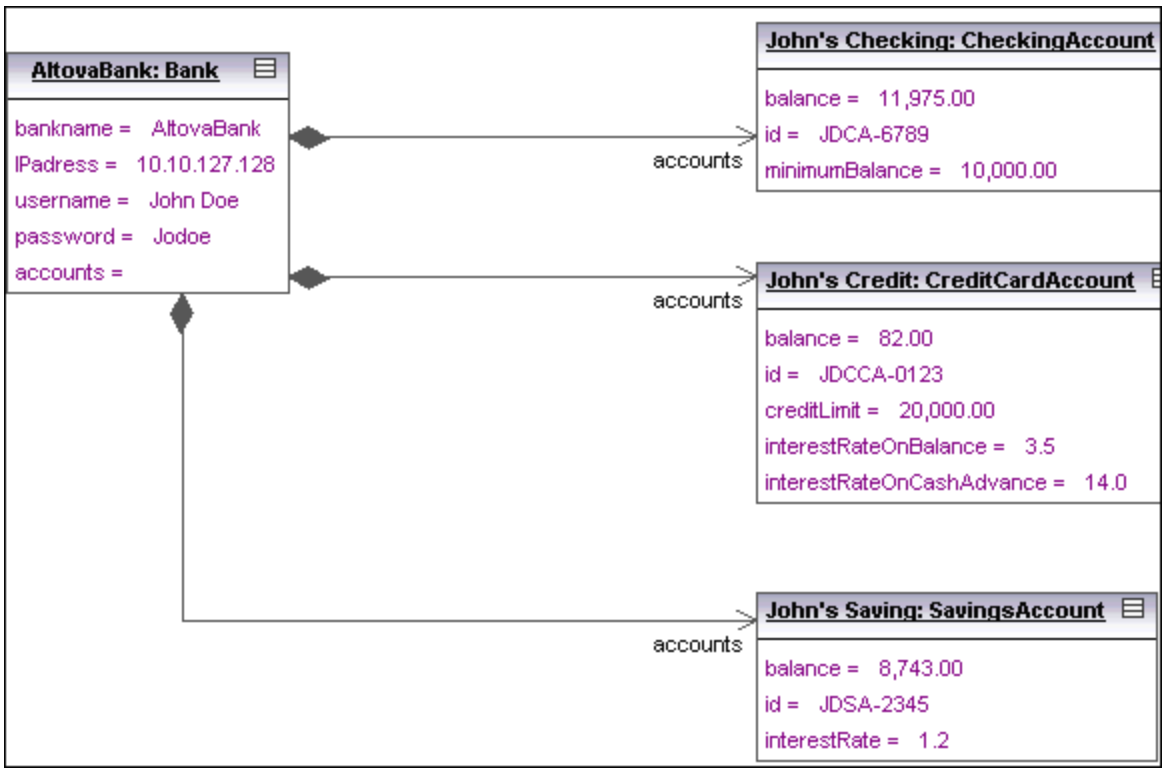
Please see the Deployment Diagrams [59] section in the tutorial for more information on how to add nodes and artifacts to the diagram.



© 2019-2025 Altova GmbH

8.2.5 Object Diagram

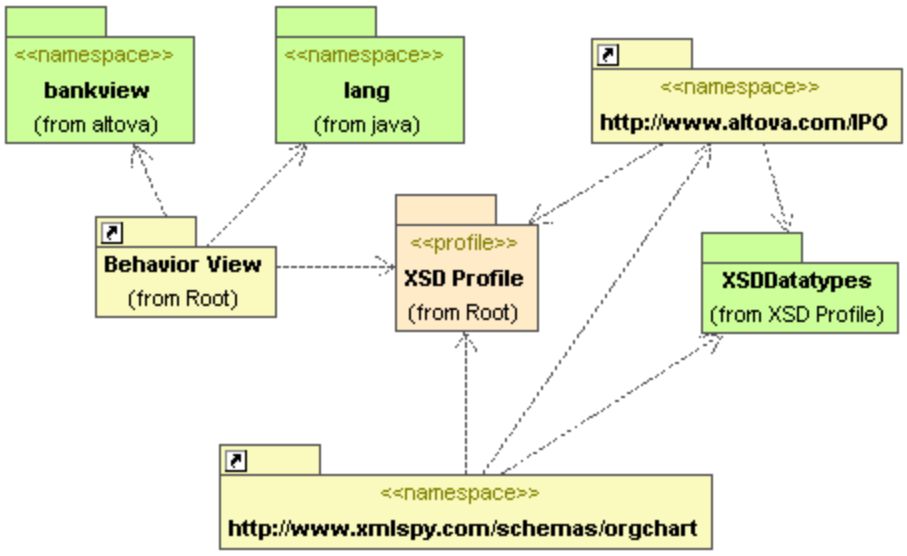
Please see the Object Diagrams 46 section in the tutorial for more information on how to add new objects/instances to the diagram.



8.2.6 Package Diagram

Package diagrams display the organization of packages and their elements, as well as their corresponding namespaces. UModel additionally allows you to create a hyperlink and navigate to the respective package content.

Packages are depicted as folders and can be used on any of the UML diagrams, although they are mainly used on use-case and class diagrams.



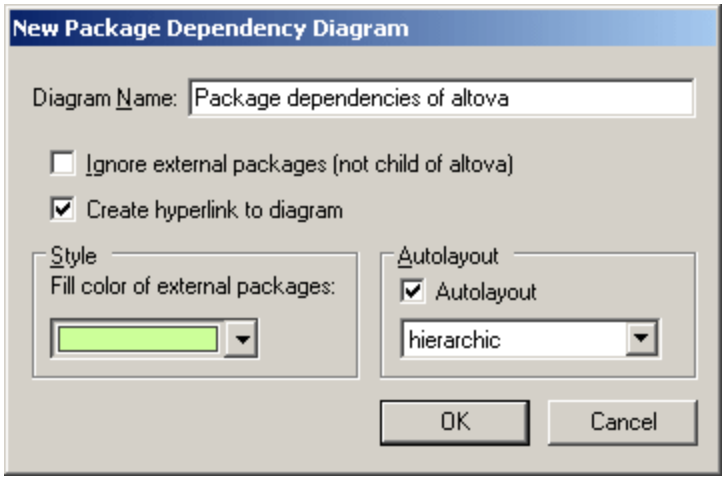
Automatic Package Dependency diagram generation

You can generate a package dependency diagram for any package that already exists in the Model Tree.

Dependency links between packages are created if there are any references between the modeling elements of those packages. E.g. Dependencies between classes, derived classes, or if attributes have types that are defined in a different package.

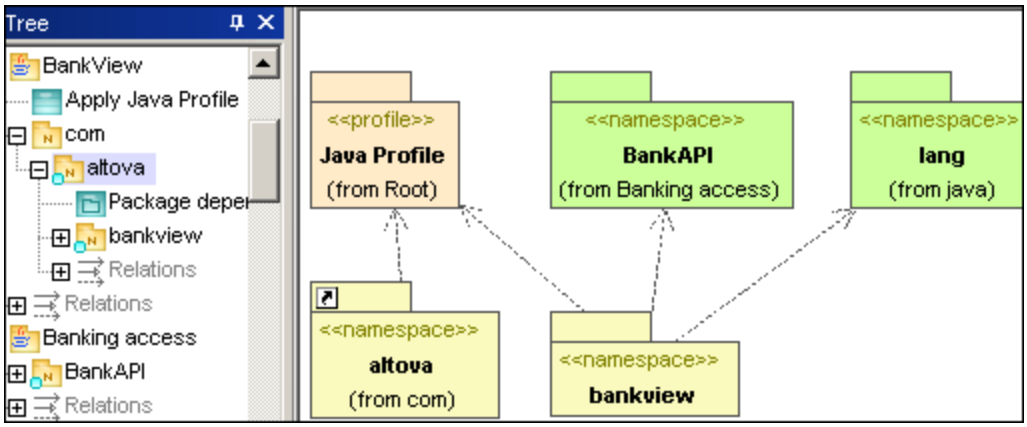
To generate a package dependency diagram:

1. Right click a package in the Model Tree, e.g. altova, and select **Show in new Diagram | Package Dependencies...**. This opens the New Package Dependency Diagram dialog box.



2. Select the specific options you need and click OK to confirm.

© 2019-2025 Altova GmbH



A new diagram is generated and displays the package dependencies of the altova package.

8.2.6.1 Inserting Package Diagram elements

Using the toolbar icons

1. Click the specific icon in the Package Diagram toolbar.

2. Click in the diagram to insert the element. To insert multiple elements of the selected type, hold down the **Ctrl** key and click in the diagram window.

Dragging existing elements into the Package Diagram

Elements occurring in other diagrams, e.g. other packages, can be inserted into a Package diagram.

1. Locate the element you want to insert in the Model Tree tab (you can use the search function text box, or press **Ctrl+F** to search for any element).
2. Drag the element(s) into the diagram.

Package

Inserts the package element into the diagram. Packages are used to group elements and also to provide a namespace for the grouped elements. Being a namespace, a package can import individual elements of other packages, or all elements of other packages. Packages can also be merged with other packages.

Profile

Inserts the Profile element, which is a specific type of package that can be applied to other packages.

The Profiles package is used to extend the UML meta model. The primary extension construct is the Stereotype, which is itself part of the profile. Profiles must always be related to a reference meta model such as UML, they cannot exist on their own.

Dependency

Inserts the Dependency element, which indicates a supplier/client relationship between modeling elements, in this case packages, or profiles.

PackageImport

Inserts an <<import>> relationship which shows that the elements of the included package will be imported into the including package. The namespace of the including package gains access to the included namespace; the namespace of the included package is not affected.

Note: Elements defined as "private" within a package, cannot be merged or imported.

PackageMerge

Inserts a <<merge>> relationship which shows that the elements of the merged (source) package will be imported into the merging (target) package, including any imported contents of the merged (source) package.

If the same element exists in the target package then these elements' definitions will be expanded by those from the target package. Updated or added elements are indicated by a generalization relationship back to the source package.

Note: Elements defined as "private" within a package, cannot be merged or imported.

ProfileApplication

Inserts a Profile Application which shows which profiles have been applied to a package. This is a type of package import that states that a Profile is applied to a Package.

The Profile extends the package it has been applied to. Applying a profile, using the **ProfileApplication** icon, means that all stereotypes that are part of it, are also available to the package.

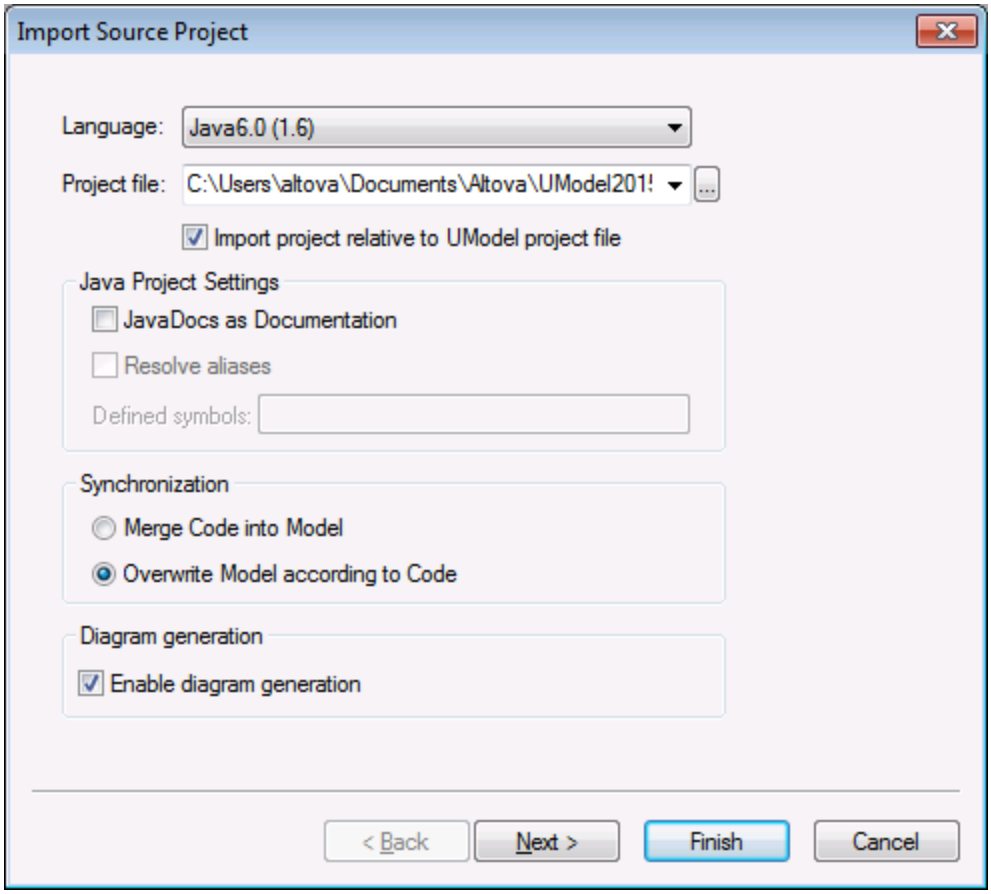
Profile names are shown as dashed arrows from the package to the applied profile, along with the <<apply>> keyword.

8.2.6.2 Generating Package Diagrams

You can instruct UModel to generate package diagrams when importing source code or binaries into the UModel project (see Importing Source Code 192 and Importing Java, C# and VB.NET Binaries 204). When following the import wizard, make sure that:

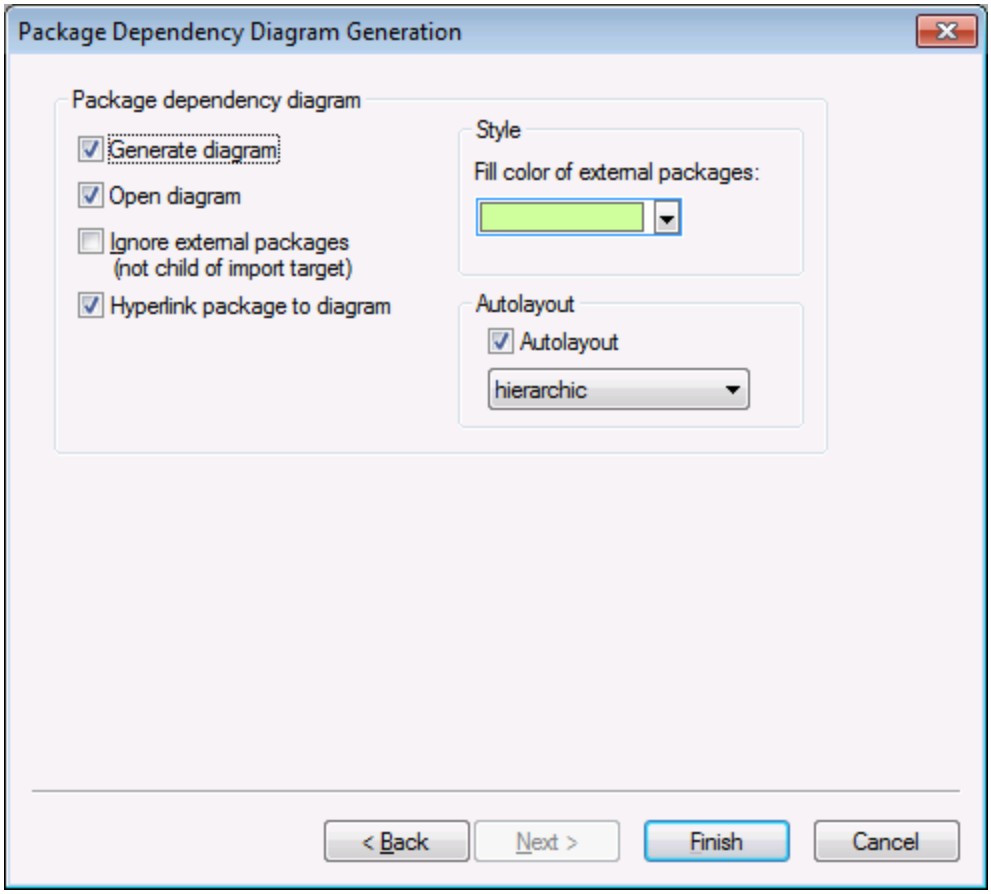
1) The **Enable diagram generation** check box is selected on the "Import Source Project", "Import Binary Types", or "Import Source Directory" dialog box.

 © 2019-2025 Altova GmbH



Import Source Project dialog box

2) The **Generate diagram** option is selected on the "Package Dependency Diagram Generation" dialog box.



Package Dependency Diagram Generation dialog box

Once the import operation is finished, any generated package diagrams are available under "Package Diagrams" in the Diagram Tree.

© 2019-2025 Altova GmbH

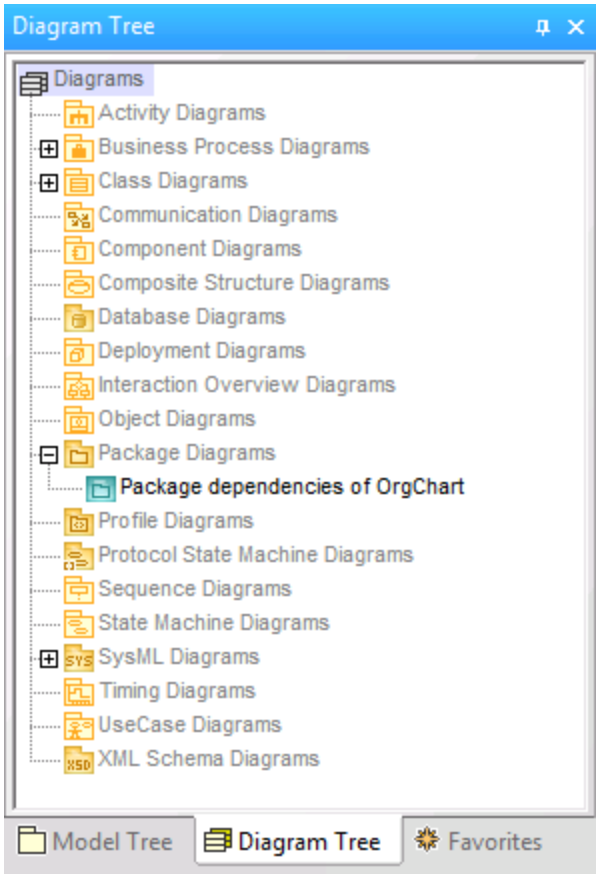


Diagram Tree

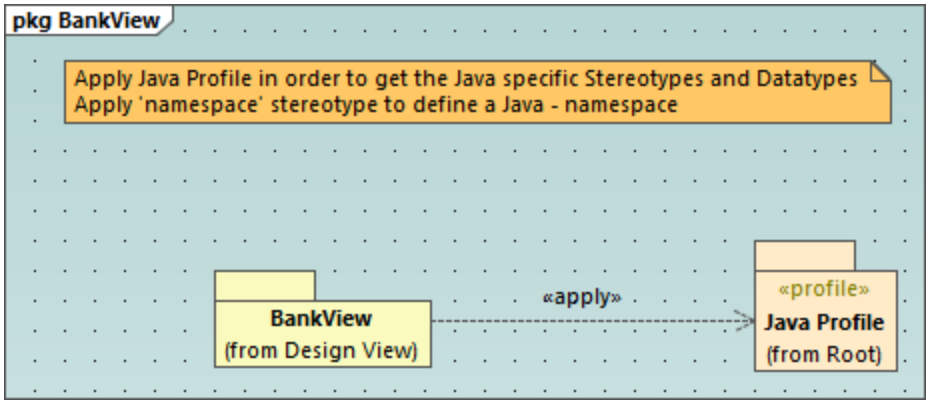
8.2.7 Profile Diagram

Altova website: UML profile diagrams

In UML, profiles are a way to extend UML to a specific platform or domain. Unlike a package, a profile is in the meta-model and consists of "meta" building blocks that extend or constrain something. This is possible with the help of the following extension mechanisms included into a profile: stereotypes, tagged values, and constraints.

In UModel, the profile diagram is where you can conveniently create your own stereotypes, tagged values and constraints bundled as a custom profile. Profiles enable you to extend or adapt UML to your specific domain or customize the appearance of elements in your modeling projects. For example, you may want to define custom styles or add custom icons for UML elements such as classes, interfaces, and so on.

Importantly, the profile diagram is where you can *apply a profile* to a package. For example, the profile diagram below illustrates a **ProfileApplication** relationship between the package **BankView** and the Java profile built into UModel. You can find this diagram in the following sample project: **C:\Users\<username>\Documents\Altova\UModel2025\UModelExamples\BankView_Java.ump**; it is called "Apply Java Profile".



Profile diagram

The applied Java profile means that any class or interface that is part of the **BankView** package (or will be added to this package in future) must look like a Java class or interface and all its members must exhibit behavior specific to that language. For example:

- All Java data types that exist in the profile are available for selection from a drop-down list when you design a class in a class diagram, see also Class Diagrams 31.
- All Java-specific stereotypes defined in the profile, such as «annotations», «final», «static», «strictfp», and so on, are visible as properties in the Properties window when you select an element.

This chapter describes how you can extend UModel projects by means of custom profiles and stereotypes. For information about using the UModel built-in profiles, see Applying UModel Profiles 160 and Stereotypes and Tagged Values 146.

8.2.7.1 Creating and Applying Custom Profiles

The instructions below show you how to create a custom UModel profile and apply it to a package. This is typically required if you need to create and apply stereotypes beyond those included in the default UModel profiles. For information about applying the default UModel profiles, see Applying UModel Profiles 160.

To create a custom profile:

1. Right-click the package where you would like to create the new profile, (for example, "Root"), and select **New element | Profile** from the context menu.
2. Create all the elements that should be part of this profile, such as stereotypes, data types, and so on. You can do this either in the Model Tree window or from a profile diagram. For example, to create a new stereotype in the model, right-click the profile and select **New element | Stereotype** from the context menu. See also Creating Stereotypes 413.
3. Optionally, create a profile diagram (right-click the profile and select **New diagram | Profile diagram** from the context menu). To add all the required elements to the diagram, use the standard UModel menu commands and toolbars, see How to Model... 108.

If you would like to create the profile from a profile diagram, make sure that the diagram is owned by

© 2019-2025 Altova GmbH

(created under) a profile, or by a package inside a profile.

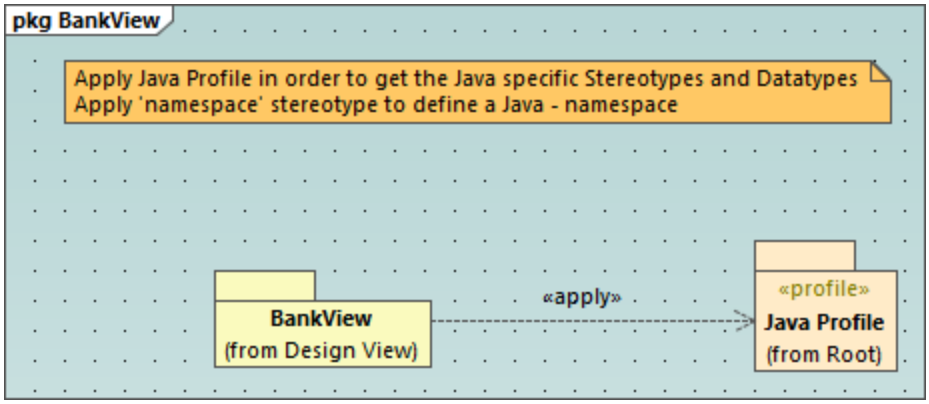
In addition, if you would like to reuse the profile across multiple UModel projects, do the following:

1. Share any packages that you want to make reusable. (Right-click the package or the profile itself, and select **Subproject | Share package** from the context menu.)
2. Save the project to a directory from where you can later include it as a subproject, see Including Subprojects 164.

So far, you have created a profile but have not added (or applied) it to any package. By applying a profile to a package, you make all of the extension mechanisms of that profile (such as stereotypes, data types, and so on) available to elements of the package.

To apply a custom profile to a package:

1. Create a new UModel project, or open an existing one.
2. Do one of the following:
   a. Create your custom profile in the existing project, as shown above.
   b. Include a custom profile from an existing project using the menu command **Project | Include Subproject**. Note that either the entire profile or its packages under must be shared in order to be reusable, see Sharing Packages and Diagrams 166 .
3. Right-click the profile and select **New diagram | Profile diagram** from the context menu.
4. Add some package(s) and the custom profile to the diagram.
5. Draw a **ProfileApplication** relationship from the package to the profile. For example, the profile diagram below illustrates a **ProfileApplication** relationship between the package **BankView** and the Java profile built into UModel. As illustrated below, profile applications are shown as dashed arrows from the package to the applied profile, along with the `<<apply>>` keyword.



8.2.7.2 Creating Stereotypes

When you model projects using any of the UModel built-in profiles (such as C#, Java, VB.NET, XML schema, database, and so on), you shouldn't typically need to create any custom stereotypes. Instead, you can just apply the existing stereotypes to your model's elements, as described in Applying Stereotypes 148.

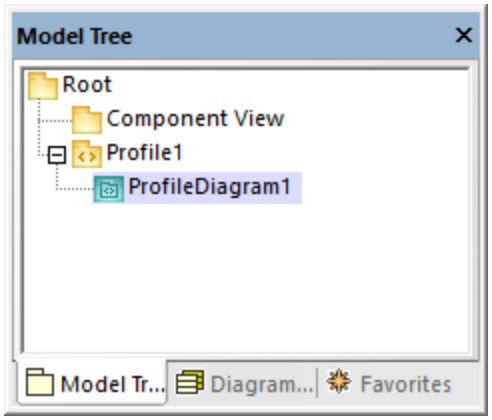
However, if you would like to add custom icons to elements or customize their appearance based on the applied stereotype, this can be achieved by creating custom stereotypes. Note the following prerequisites:

- Stereotypes must be owned by a profile or a package inside a profile. Therefore, in order to create a stereotype, you must create a profile first (or a package inside an existing profile).
- After creating the profile, you must apply it to the package where you need to use the custom stereotypes, as described in Creating and Applying Profiles 412.

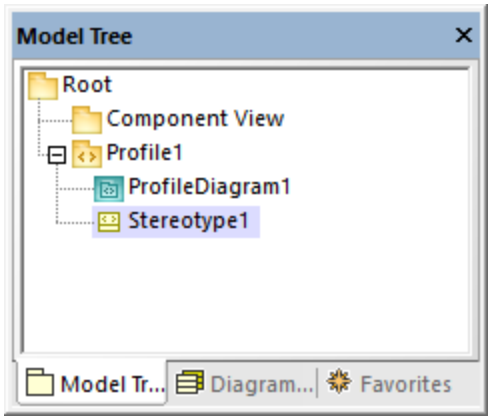
Once you have created a profile, you can start adding stereotypes to it. This can be done either directly in the Model Tree window, or from a profile diagram. If you would like to create stereotypes from a profile diagram, make sure that the diagram is owned by (created under) a profile, or by a package inside a profile, as shown below.

To create a stereotype:

1. If you haven't done so already, create a profile, see Creating and Applying Custom Profiles 412.
2. Optionally, right-click the profile and select **New diagram | Profile diagram** from the context menu. This creates a new profile diagram under the current profile—it will help you visualize in one place all the stereotypes, data types, and other elements that you will subsequently add to the profile.

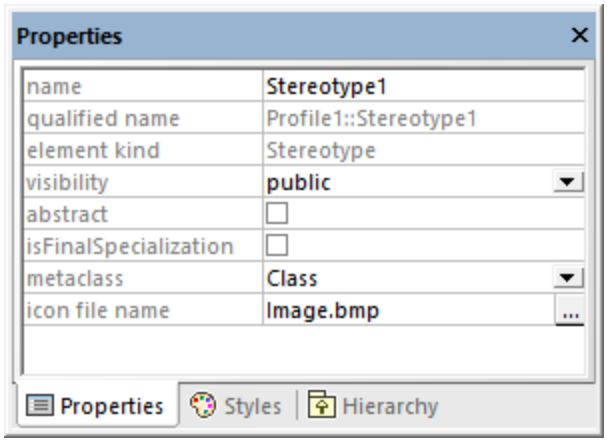


3. Right-click the profile in the Model Tree window, and select **New element | Stereotype** from the context menu.



© 2019-2025 Altova GmbH

4. Optionally, set the stereotype properties in the Properties window. For example, if you set the stereotype's **metaclass** to "Class", the stereotype will apply to classes only. Likewise, you can set a custom icon for the stereotype by clicking the **Ellipsis** button next to **icon file name**.



Notes

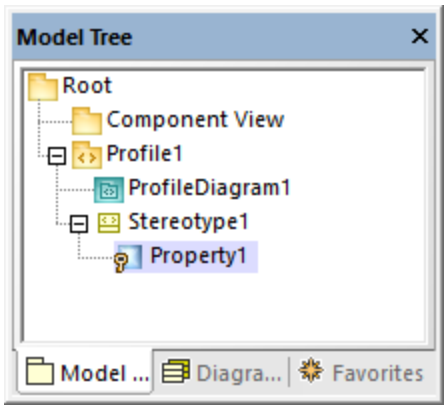
- If the image path is relative, it must be relative to the UModel project's folder.
- To use custom icons with transparent background, set their background color to RGB value 82,82,82.
- To display stereotypes for association relationships, set the **Show MemberEnd stereotypes** property to "true" in the **Styles** window.

Adding stereotype attributes (properties)

The stereotype created above is very simple and does not have any attributes (properties) associated with it. It is, however, possible to add properties to a stereotype. Such properties will become tagged values when this stereotype is applied to some element in future.

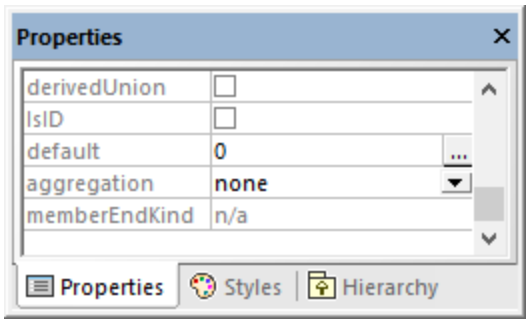
To add attributes (properties) to a stereotype:

1. Click the stereotype in the Model Tree window or on the diagram.
2. Do one of the following
   a. Right-click the stereotype and select **New | Property** from the context menu.
   b. Press **F7**.



You can set the data type of each property from the Properties window, by selecting a value from the **type** list. Any data type previously defined in the same profile as the stereotype is available for selection. If the profile doesn't contain any data types yet, you can define one by right-clicking the profile diagram, and selecting **New | Data type** from the context menu.

To set the default value of a property, enter that value in the **default** field of the Properties window. For example, the stereotype property illustrated below has "0" as default value:



The data type of a stereotype attribute (property) can also be an enumeration, see Example: Creating and Applying Stereotypes 416.

8.2.7.3 Example: Creating and Applying Stereotypes

This example provides a step-by-step demo of the stereotype creation process. It shows you how to achieve the following goals:

- Create a stereotype
- Create stereotype attributes (properties) that become tagged values when applied to an element
- Define a stereotype attribute as an enumeration
- Set a default value for a stereotype attribute
- Apply the stereotype to elements in the model.

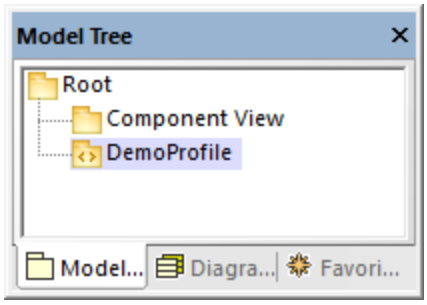
The example is accompanied by a sample project file called **StereotypesDemo.ump**, available at the following path: **C:\Users\<username>\Documents\Altova\UModel2025\UModelExamples\Tutorial**. If you follow the instructions below literally, you will create a similar project.

© 2019-2025 Altova GmbH

Create a new profile

As mentioned above, a stereotype must be owned by a profile; therefore, let's first create a profile.

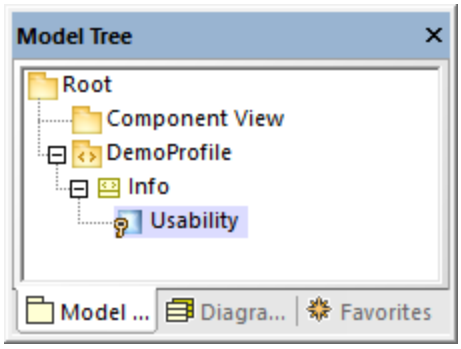
1. Create a new UModel project.
2. Right-click the "Root" package and add a new profile by selecting **New element | Profile** from the context menu.
3. Rename the new profile to "DemoProfile".



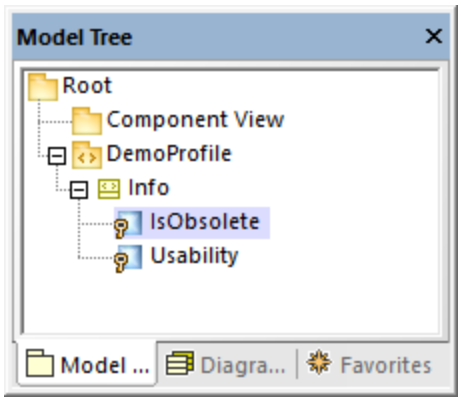
Create a stereotype

For the scope of this tutorial, you will create a stereotype with two attributes: "Usability" and "IsObsolete". The "IsObsolete" attribute will be defined as an enumeration. The enumeration will consist of two values, "Yes" and "No", where "No" is the default value.

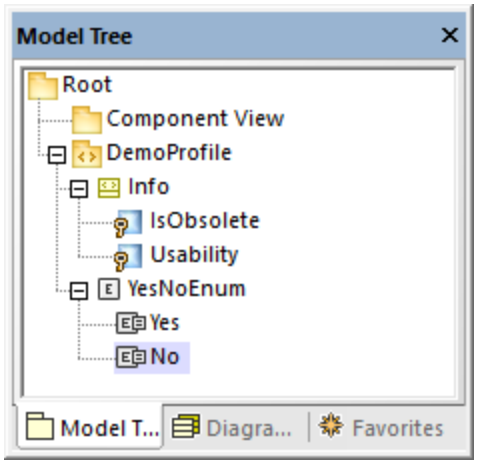
1. Right-click the profile and select **New element | Stereotype** from the context menu. A new stereotype has been added to the profile.
2. Rename the new stereotype to "Info".
3. Right-click the stereotype and select **New element | Property** from the context menu. This adds a new property.
4. Rename the new property to "Usability".



5. Repeat the steps above to create a new property called "IsObsolete".

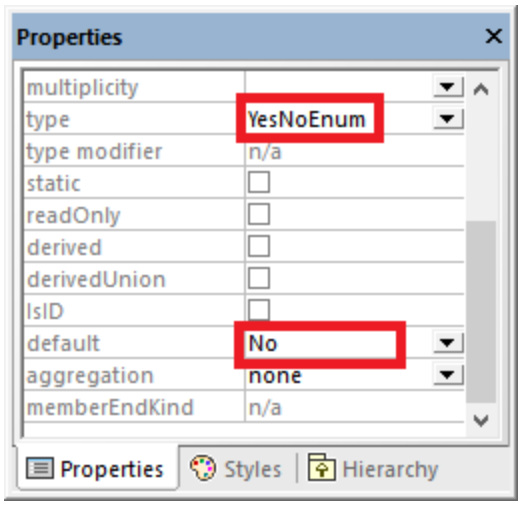


6. Right-click the "DemoProfile" and select **New Element | Enumeration** from the context menu. Rename the enumeration to "YesNoEnum".
7. Right-click the enumeration and select **New Element | EnumerationLiteral** from the context menu. Rename the enumeration literal to "Yes".
8. Repeat the step above and create an enumeration literal called "No".



9. Click the "IsObsolete" property and change its type to `YesNoEnum`. Also, set the **default** property to "No"

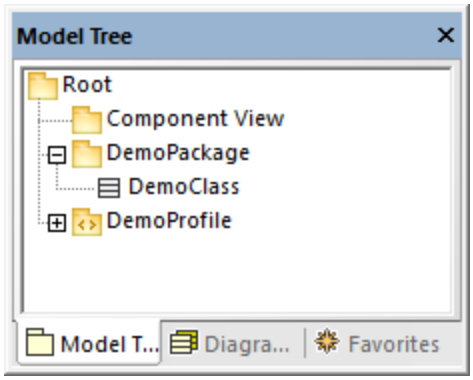
© 2019-2025 Altova GmbH



Create a new package

In order to illustrate how the custom stereotype can be used, let's create a simple package containing only one class.

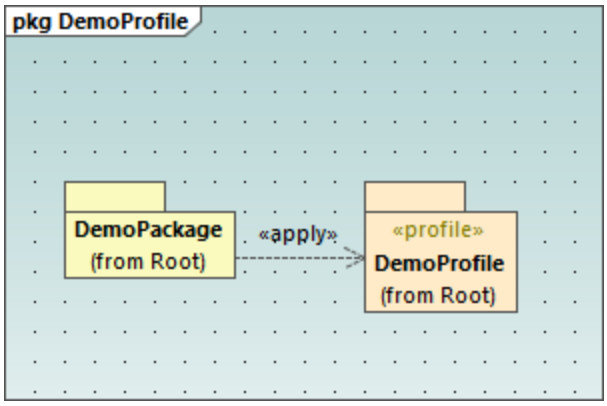
1. Right-click the "Root" package and add a new package by selecting **New element | Package** from the context menu.
2. Rename the new package to "DemoPackage".
3. Add a class to the package (in this example, "DemoClass".



Apply the profile to a package

As you recall from Step 1, the stereotype was created inside a profile. In this step, we apply the profile to a package, so that the stereotype becomes "visible" to the package.

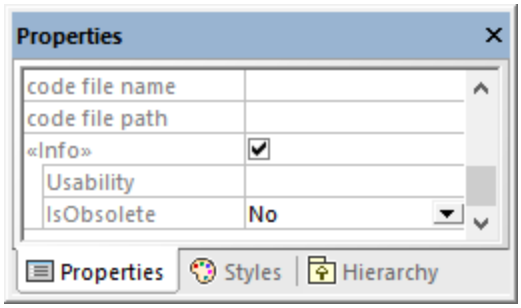
1. Right-click the "DemoProfile" in the Model Tree window and select **New diagram | Profile diagram** from the context menu.
2. Drag both the "DemoPackage" package and the "DemoProfile" profile from the Model Tree window into the diagram.
3. Click the **ProfileApplication** toolbar button, and draw a **ProfileApplication** relationship from the package to the profile.



Apply the stereotype to classes

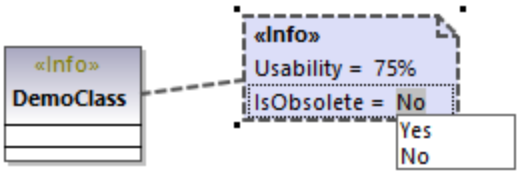
You can now apply the stereotype to a class.

1. Right-click the "DemoPackage" and select **New diagram | Class diagram** from the context menu.
2. Drag the class "DemoClass" onto the diagram.
3. Click the class and select the «Info» stereotype in the Properties window. Notice that the "IsObsolete" property is pre-filled with its default value.



4. Enter a value for the "Usability" property ("75%", in this example).

The class on the diagram now has a "Tagged values" section which displays the stereotype attributes and their values. You can change these values either from the Properties window, or directly from the diagram.



© 2019-2025 Altova GmbH

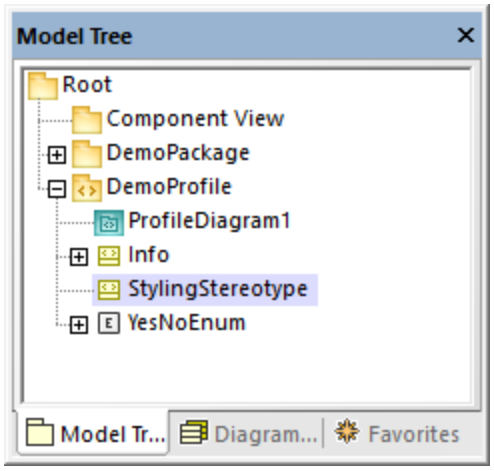
8.2.7.4 Example: Customizing Icons and Styles

This example shows you how to customize the appearance of a class in UModel with the help of stereotypes. After following this example, you will learn how to add custom icons to elements and change the style of all elements that use the same stereotype.

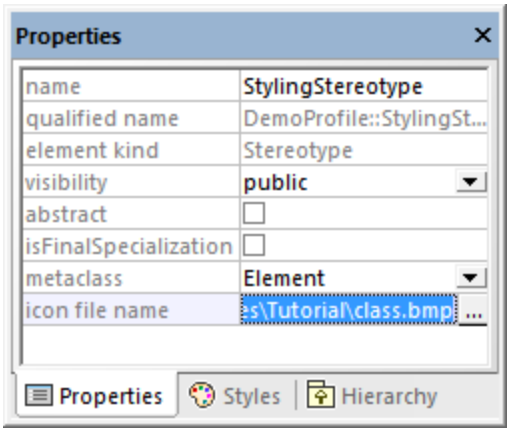
The class that will be customized in this example is in the **StereotypesDemo.ump** project, available at the following path: **C:\Users\<username>\Documents\Altova\UModel2025\UModelExamples\Tutorial**. This is a simple demo project which includes a custom profile under which we will create the stereotype. For an example that shows you how to create profiles and stereotypes from scratch, see Example: Creating and Applying Stereotypes 416.

Let's first create the stereotype to be used for styling:

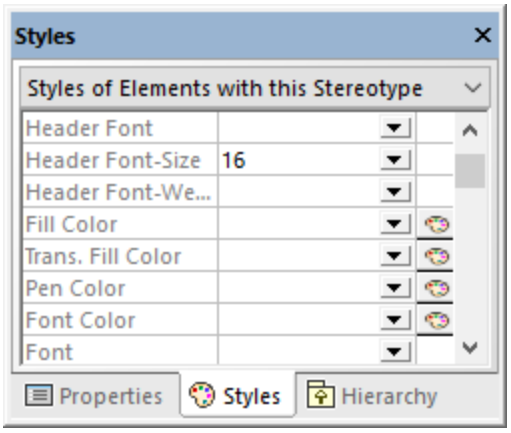
1. Open the **StereotypesDemo.ump** project.
2. Right-click the "DemoProfile" profile in the model tree, and select **New Element | Stereotype** from the context menu.
3. Rename the stereotype to "StylingStereotype".



To add a custom image to the stereotype, click the stereotype, and then click the **Ellipsis** ... button next to **icon file name** property in the Properties window. Select the following sample image: **C:\Users\<username>\Documents\Altova\UModel2025\UModelExamples\Tutorial\class.bmp**.

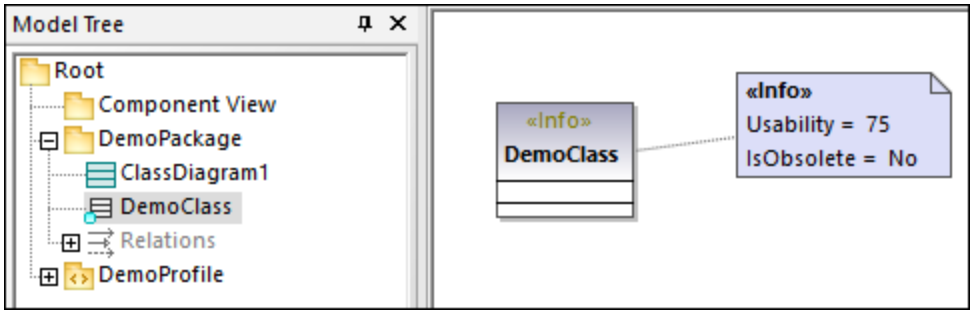


Next, click the **Styles** tab of the Properties window. Select **Styles of Elements with this Stereotype** from the top list, and change the **Header Font Size** property to "16".



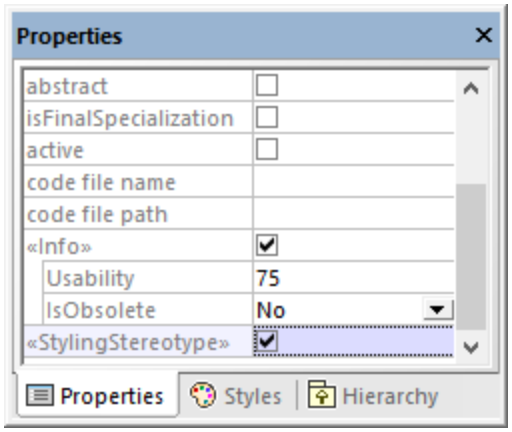
Finally, apply the stereotype to a class.

1. Open the class diagram "ClassDiagram1". You will find this diagram under the "DemoPackage" in the Model Tree view.

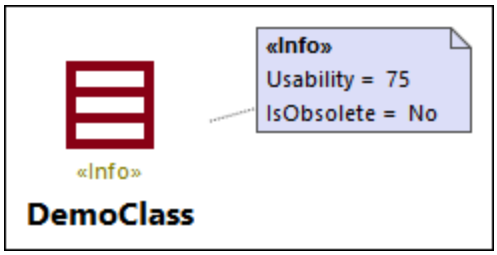


2. Click the "DemoClass" class, and then select the **«StylingStereotype»** check box in the Properties window.

© 2019-2025 Altova GmbH

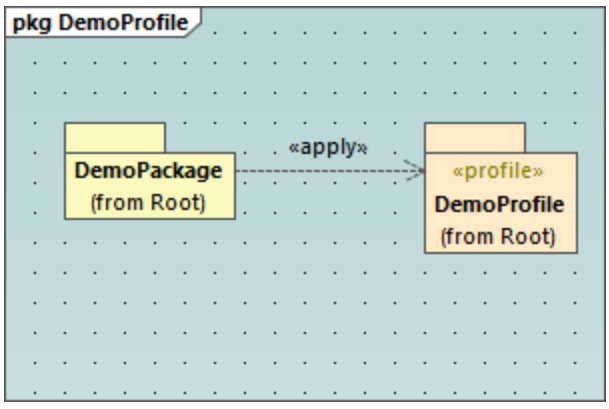


The appearance of the class on the diagram is now changed according to the applied stereotype:



Remarks

The demo project contains a profile diagram, "ProfileDiagram1". In this diagram, notice that the "DemoProfile" is applied to the "DemoPackage" with a **ProfileApplication** relationship. This makes the stereotype available to the package, see also Creating and Applying Custom Profiles 412.



You have now learned how to change the appearance of elements using stereotypes. You can use the same technique in other projects. Just keep in mind that the profile where you create the stereotype must be applied to the target package, as shown above.

8.3 Additional Diagrams

The additional diagram kinds supported by **UModel Professional Edition** are as follows:

- XML Schema diagrams 424
- Business Process Modeling Notation
- SysML diagrams
- Database diagrams 486

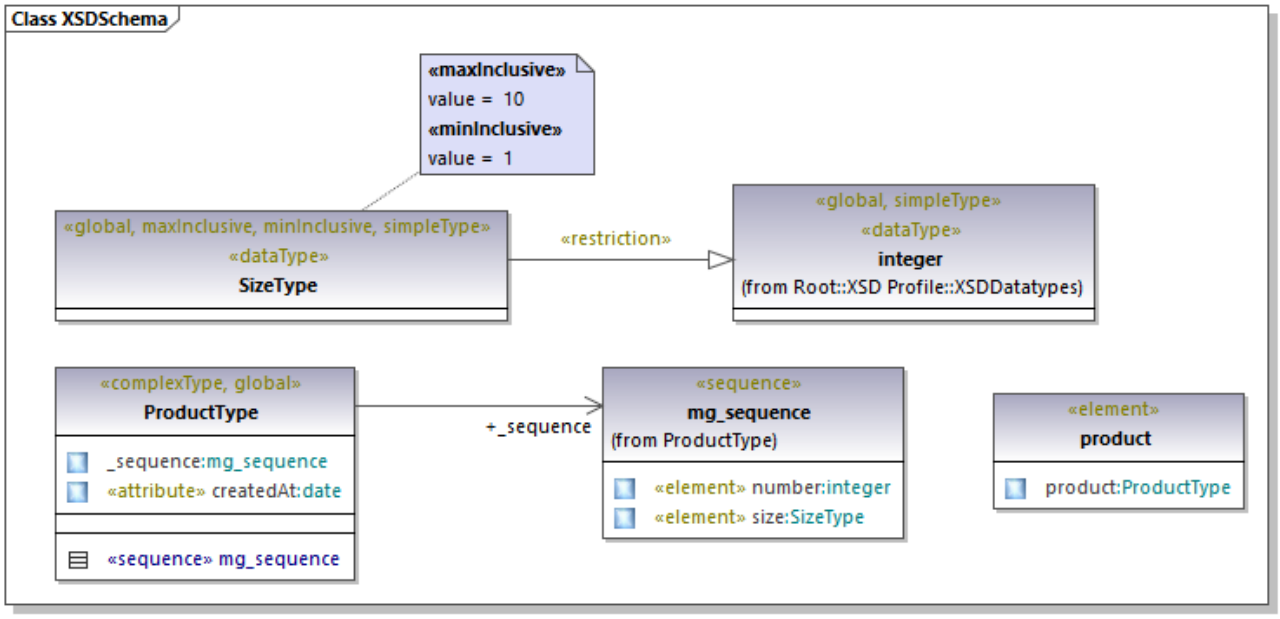
8.3.1 XML Schema Diagrams

Altova website: XML Schemas in UML

UModel supports the import and generation of W3C XML schemas as well as their forward and reverse engineering. In case of XML Schemas, "forward and reverse engineering" means that you can import a schema (or multiple schemas from a directory) into UModel, view or modify the model, and write the changes back to the schema file. When you synchronize data from the model to a schema file, the schema file is always overwritten by the model.

Note: The XML Schema must be valid before it can be imported into UModel. XML Schemas are not validated when you create or import them in UModel, or when you run a project syntax check. Nevertheless, UModel checks whether the XML schema is well-formed when importing it.

XML Schema diagrams display schema components in UML notation. For example, simple types are shown in UModel as data types with the «simpleType» stereotype. Complex types are shown as classes with the «complexType» stereotype. Various schema details are represented as Tagged Values 147, while schema annotations are represented as comments. For a mapping table that illustrates how all the XML schema components map to UModel elements, see XML Schema Mappings 264.



© 2019-2025 Altova GmbH

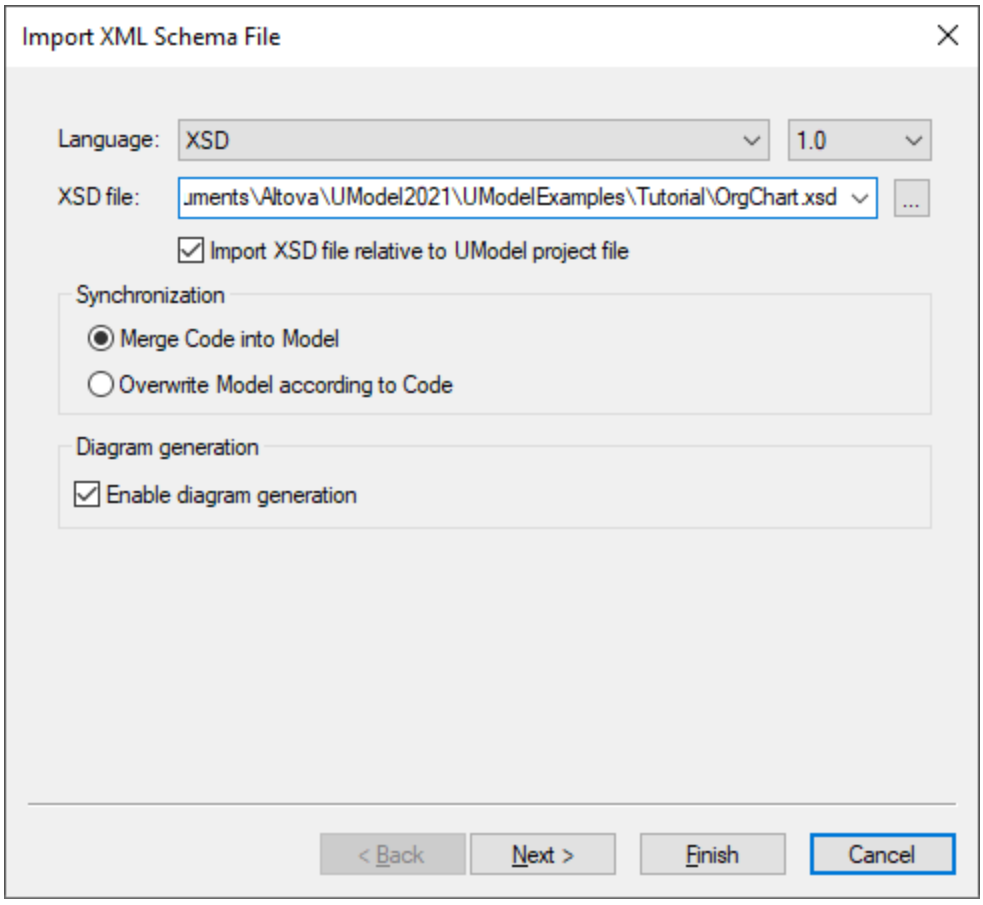
Example XML Schema diagram

8.3.1.1 Importing XML Schemas

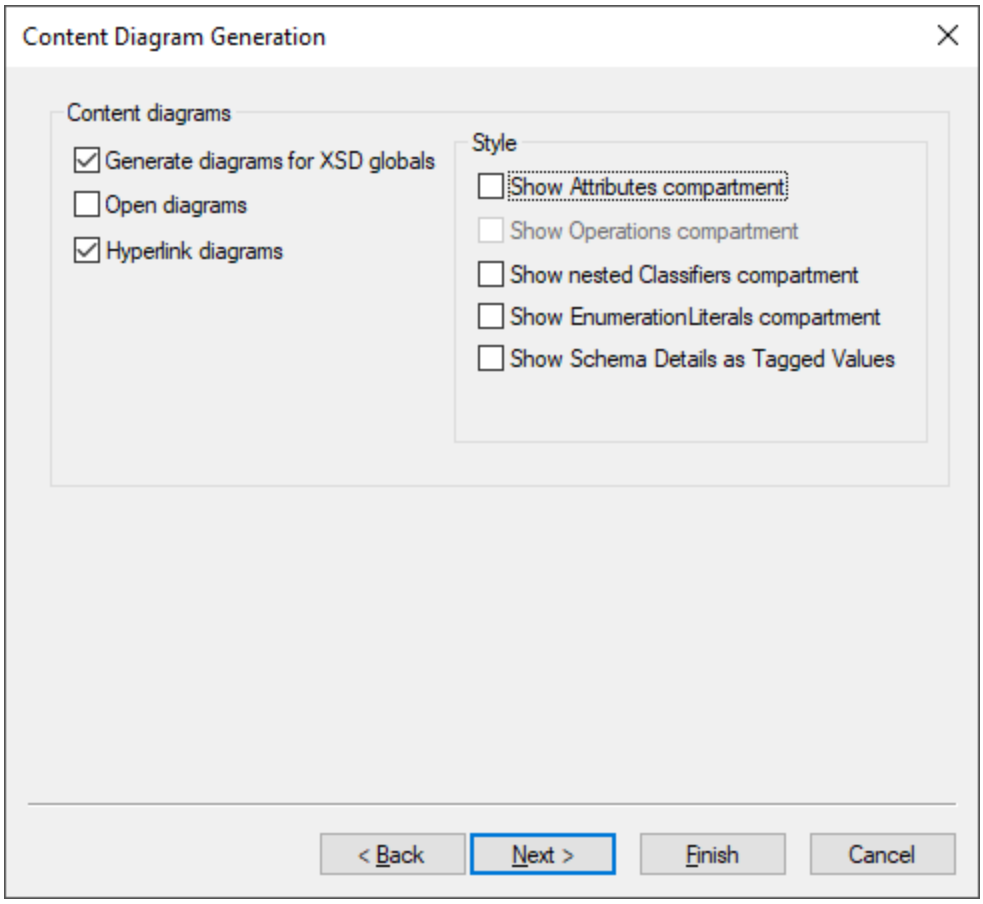
You can import either a single schema file into UModel, or all schemas from a directory. If a schema includes or imports other schemas, these are imported into the model as well.

To import a single XML Schema:

1. Select the menu command **Project | Import XML Schema file**.
2. Click **Browse** and select the source schema to import. For the scope of this example, you can use the following schema: **C:\Users\<username>\Documents\Altova\UModel2025\UModelExamples\Tutorial\OrgChart.xsd**.

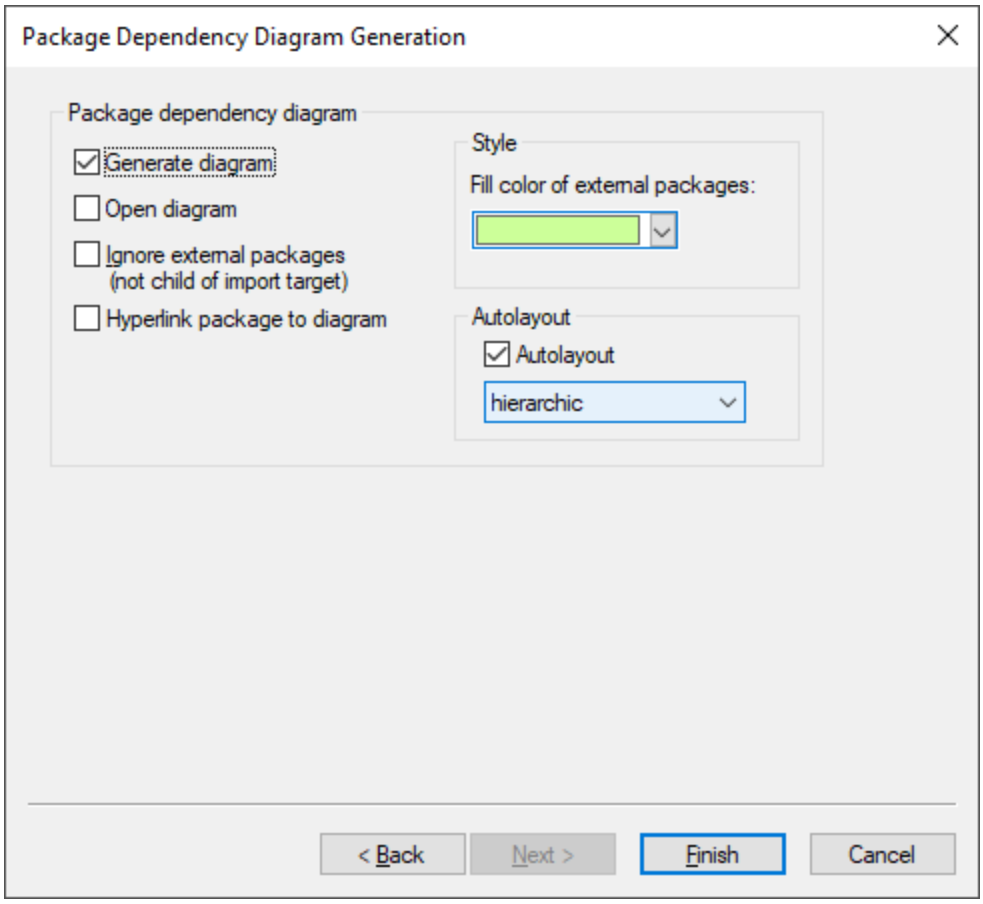


3. To generate diagrams from the schema, make sure that the **Enable diagram generation** check box is selected and click **Next**.



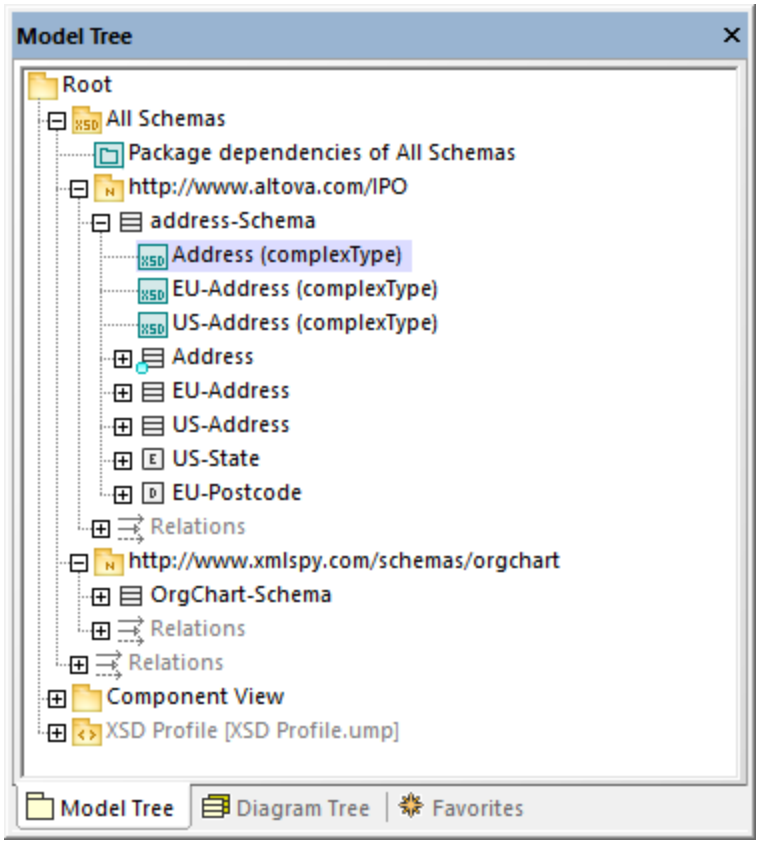
4. To create a separate diagram for each global component in the schema like in this example, select the **Generate diagrams for XSD globals** option. To open all generated diagrams after import, select **Open diagrams**. Options from the "Style" group let you define the compartments that appear by default in diagrams for each schema component. The **Show schema details as tagged values** option displays the schema details as Tagged Values 147.
5. Click **Next**. To generate a Package dependency diagram like the one in this example, select the **Generate Diagram** check box.

© 2019-2025 Altova GmbH



6. Click **Finish**.

Once UModel completes importing the schema, a new package called **All Schemas** is created and set automatically as the "XSD Namespace Root". The **OrgChart.xsd** schema used in this example imports types from another namespace, more specifically, from the **ipo.xsd** schema. Consequently, both schemas appear in the Model Tree window after import, under their respective namespaces:

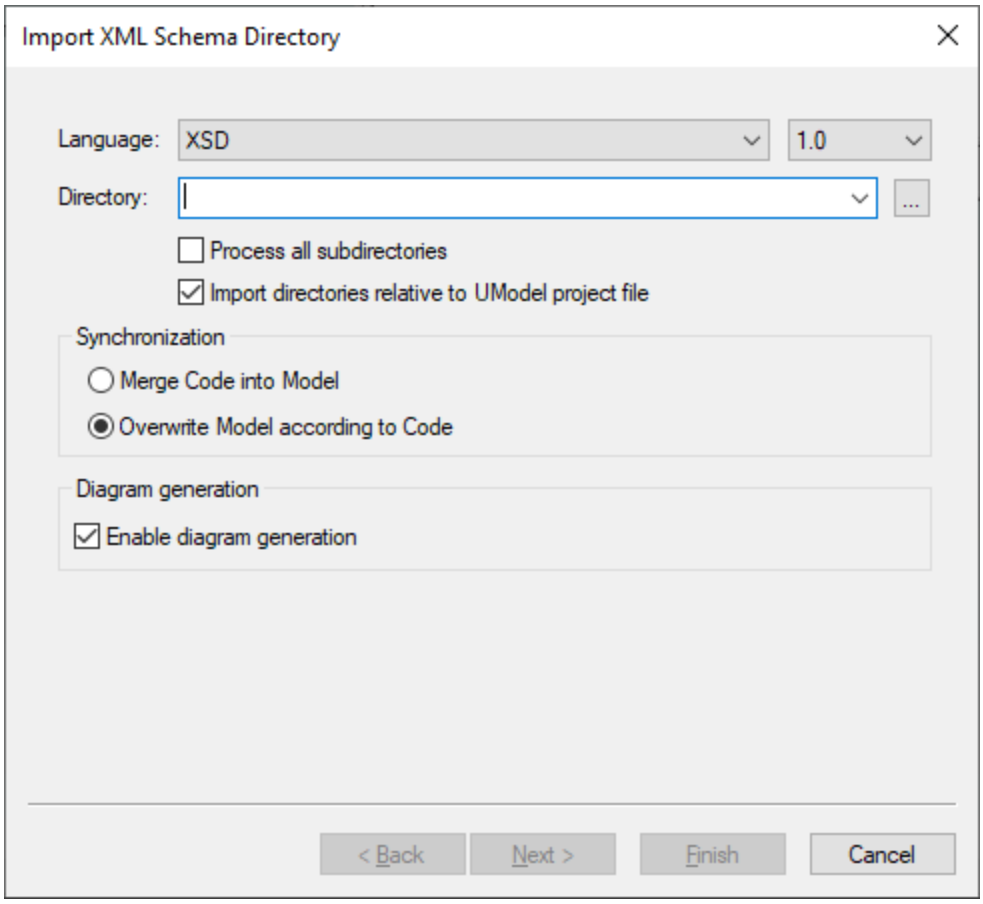


If you have selected the **Generate diagrams for XSD globals** check box, all XSD global components generate an XML Schema diagram, and the diagrams appear under the respective namespace packages, like the "Address (complexType)" diagram in the image above.

To import multiple XML Schemas:

1. Select the menu command **Project | Import XML Schema directory**.

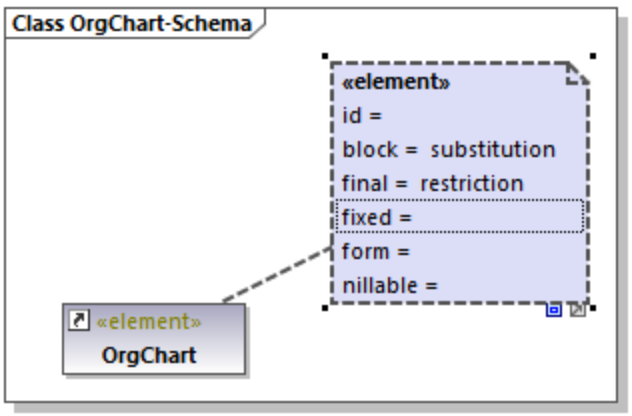
© 2019-2025 Altova GmbH



2. To import schemas from all subdirectories of the selected directory, select the **Process all subdirectories** check box. The rest of the import process is the same as described above for a single XML schema.

Changing the display of tagged values

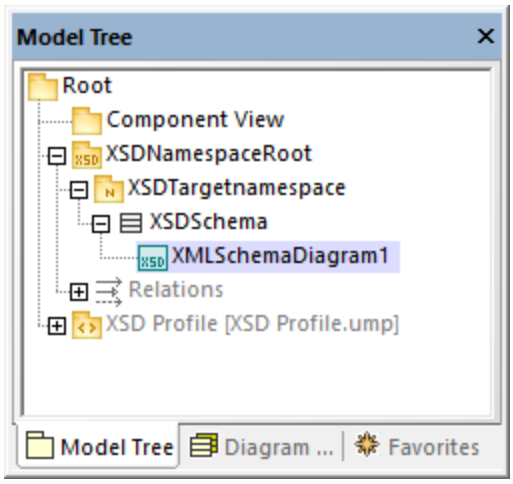
After importing an XML schema, certain schema details may appear as tagged values on the diagram, if you have selected the **Show Schema Details as Tagged Values** option during the import.



You can configure whether such details are to be shown or hidden from the diagram. To do this, right-click the element and select **Tagged Values | <option>** from the context menu. You can configure the display of tagged values not only individually for each element, but also globally at project level. For more information, see Showing or Hiding Tagged Values 150.

8.3.1.2 Modeling XML Schemas

New XML Schema projects in UModel have the structure illustrated below. This structure is created automatically the first time when you add an XML Schema diagram to a new UModel project.



The "Root" and "Component View" packages are common to any UModel Project and cannot be deleted. "Root" is the topmost level under which any other packages are added, and "Component View" is used for code engineering (in this case, importing or generating schema files).

The "XSDNamespaceRoot" package includes all the namespaces used by your schema(s). To turn a package into an XSD Namespace Root, right-click it and select **Code Engineering | Set as XSD Namespace Root** from the context menu. If you import an existing XML schema into the project, this package is called "All schemas" by default.

The "XSDTargetnamespace" package is an XML Schema namespace. Multiple such namespaces may exist under the same XSD Namespace Root. To turn a package into a namespace, first select the package, and then select the «namespace» property (stereotype) in the Properties window.

"XSDSchema" is a schema, or, in UML terms, a class with the «schema» property (stereotype) selected in the Properties window.

XMLSchemaDiagram1 is the actual diagram that describes the schema's model. You can create XML Schema diagrams under an XSD Namespace Root, under an XML Schema Namespace, or under an XML Schema. In the example project illustrated above, the diagram is created under the XML schema.

The **XSD Profile** enables all the types and structures required to work with XML Schema in the project. If your project does not have this profile, you will be prompted to include it whenever you create a new XML Schema diagram. You can also add the XSD profile to a project explicitly, see Applying UModel Profiles 160.

© 2019-2025 Altova GmbH

Creating XML Schema diagrams

To create a new XML schema diagram:

1. Do one of the following:
   a. Right-click a package in the Model Tree Window[83] and select **XML Schema Diagram** from the context menu.
   b. Right-click "Diagrams" or "XML Schema Diagrams" in the Diagram Tree Window[87] and select **New Diagram | XML Schema diagram** from the context menu. A dialog box opens asking you to select the owner of the diagram. Select a package where the diagram should be stored, and click **OK**.
2. If the current UModel project does not include the XSD profile, a dialog box opens asking you to include it. Click **OK** to include the XSD profile into the current project, see also Applying UModel Profiles[160].

Adding new XML Schema elements

To add XML schema elements to a diagram:

- Click a specific toolbar button, and then click inside the XML Schema diagram.

To insert multiple elements of the same type, hold down the **Ctrl** key and click multiple times in the diagram.

As stated above, XML Schema diagrams can be created at various levels in the project's structure. If the diagram is at a level which does not allow placing a particular element, certain toolbar buttons are not meaningful and they show a tooltip with information instead of adding the element.

The table below lists all the toolbar buttons and their purpose.

	XSD Target Namespace	Adds an XSD target namespace. Clicking this button is meaningful if the diagram was created directly under an XSD Namespace Root.
	XSD Schema	Adds an XML Schema Definition (XSD). Clicking this button is meaningful if the diagram was created under an XSD target namespace.
	Element (global)	Adds a global element to the diagram. When you add an element, a property with the same name as the element is automatically generated in the attributes compartment. Set the property type to set the element's type.
	Group	Adds a named model group to the diagram.
	Complex Type	Adds a global complex type to the diagram. In UML terms, this is a class that has the «global» and «complexType»

		stereotypes applied.
	Complex Type with Simple Content	Adds a global complex type with simple content. In UML terms, this is a data type that has the «global», «complexType», and «simpleContent» stereotypes applied.
	Simple Type	Adds a global simple type.
	List	Adds a list type.
	Union	Adds a union type.
	Enumeration	Adds an enumeration.
	Attribute	Adds an attribute.
	Attribute group	Adds an attribute group.
	Notation	Adds a notation type.
	Import	Adds an import relationship.
	Include	Adds an include relationship.
	Redefine	Adds a redefine relationship.
	Restriction	Adds a restriction relationship.
	Extension	Adds an extension relationship.
	Substitution	Adds a substitution relationship.
	Comment	Adds a comment. Comments are converted to annotations when you generate the schema file from the model. You can specify the annotation type by selecting the required stereotype from the Properties window.
	Note	Adds an explanatory note.
	Note link	Links a note to some other element on the diagram.

For step-by-step schema modeling instructions, see Example: Create and Generate an XML Schema 432.

8.3.1.3 Example: Create and Generate an XML Schema

This example shows you how to model a new XML Schema with UModel, step by step. After modeling the schema visually using UML, you will generate the schema file. More specifically, you will learn how to create and generate the **product.xsd** schema listed below.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema targetNamespace="http://www.altova.com/umodel"
xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns:prod="http://www.altova.com/umodel">
   <xs:simpleType name="SizeType">
```


© 2019-2025 Altova GmbH

```
      <xs:restriction base="xs:integer">
         <xs:maxInclusive value="10"/>
         <xs:minInclusive value="1"/>
      </xs:restriction>
   </xs:simpleType>
   <xs:complexType name="ProductType">
      <xs:sequence>
         <xs:element name="number" type="xs:integer">
         </xs:element>
         <xs:element name="size" type="prod:SizeType">
         </xs:element>
      </xs:sequence>
      <xs:attribute name="createdAt" type="xs:date">
      </xs:attribute>
   </xs:complexType>
   <xs:element name="product" type="prod:ProductType">
   </xs:element>
</xs:schema>
```

product.xsd

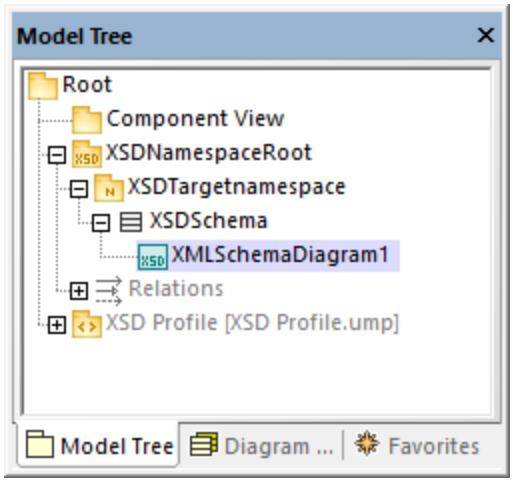
As shown above, the **product.xsd** schema has two namespace declarations:

1. The default XML Schema namespace `http://www.w3.org/2001/XMLSchema` mapped to the "xs" prefix.
2. The secondary namespace `http://www.altova.com/umodel` mapped to the "prod" prefix, which is also the target namespace.

Also, the XML schema has a global `product` element, a complex type `ProductType` and a simple type `SizeType`.

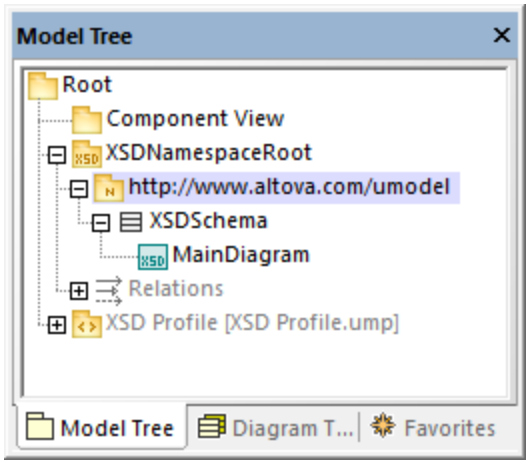
Declaring namespaces and file encoding

To proceed, create a new UModel project. Right-click the **Root** package, and select **New Diagram | XML Schema Diagram** from the context menu. When prompted to include the UModel XSD Profile, click **OK**.



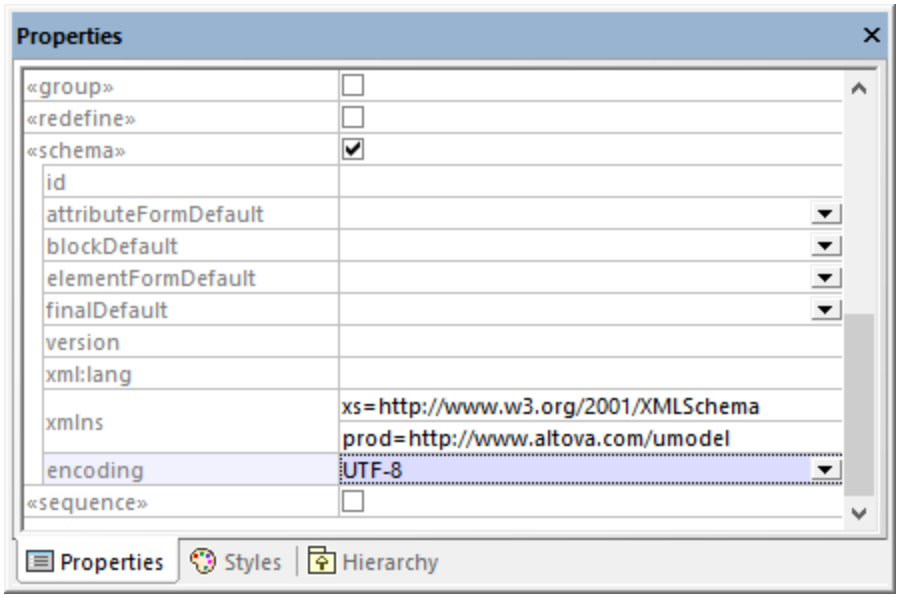
In the Model Tree Window[83], rename "XMLSchemaDiagram1" to "MainDiagram". This is the diagram where most schema components will be created, except for namespace declarations.

Next, rename "XSDTargetNamespace" to "http://www.altova.com/umodel" (recall that this is the required target namespace). This declares the target namespace of the new schema.



The two "xmlns" namespaces and the UTF-8 encoding can be set as follows:

1. Select the **XSDSchema** schema in the Model Tree.
2. In the Properties window, right-click the **xmlns** property and select **Add Tagged Value | xlmns**.
3. Edit the **xmlns** and **encoding** properties as shown below.

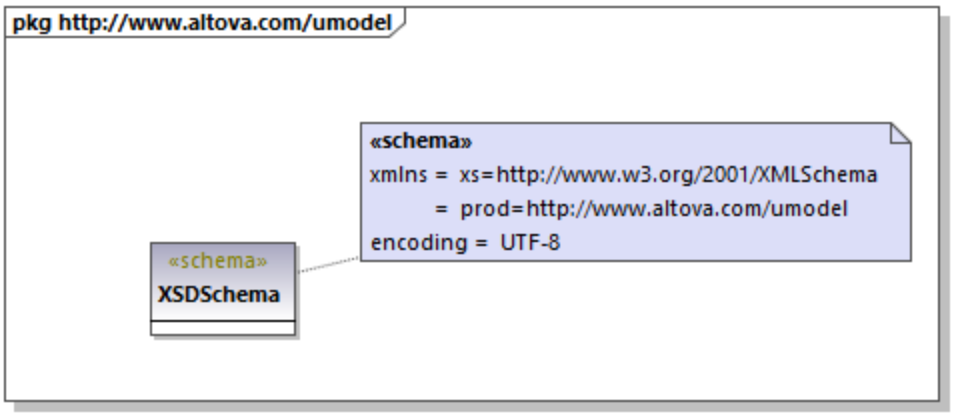


Optionally, you can quickly generate a new XML Schema diagram at namespace level that presents the same information visually, as follows:

1. In the Model Tree, right-click the namespace "http://www.altova.com/umodel" and select **New Diagram | XML Schema diagram** from the context menu.

© 2019-2025 Altova GmbH

2. When a message box with the following text appears: *"Do you want to add the 'XML Schema Diagram' to a new 'XSD Schema'?"*, click **No**.
3. Drag the XML Schema from the Model Tree into the diagram.

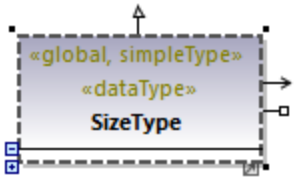


As shown above, the namespace and encoding are stored as Tagged Values [147] and can be edited from the diagram window as well.

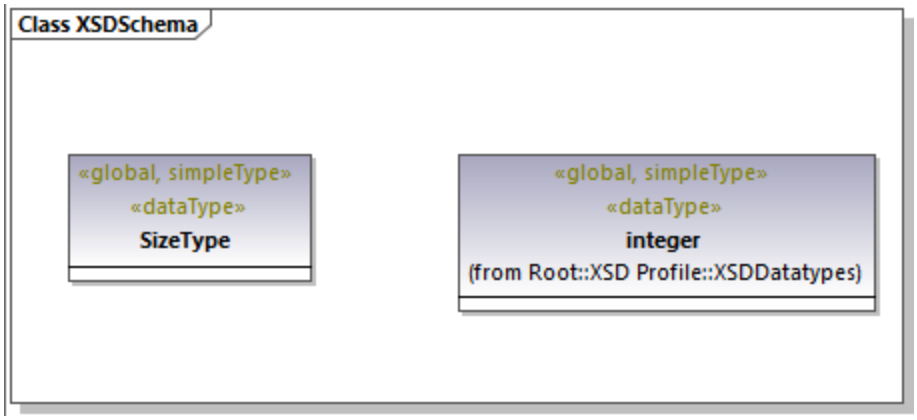
Add a simple type

The following steps create the `SizeType` simple type to the XML schema. This is a type that restricts the base `xs:integer` type; therefore, we will add the base type to the diagram as well, and create a restriction relationship.

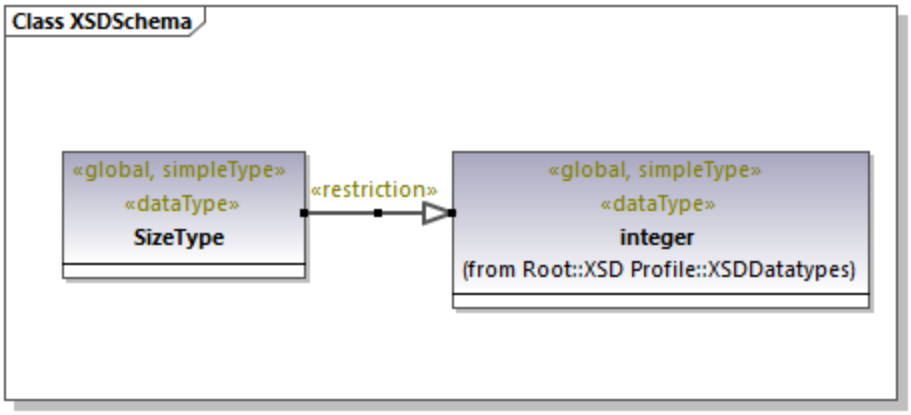
1. Double-click the **MainDiagram** in the Model Tree to open it.
2. Click the **XSD Simple Type** toolbar button, and then click inside the diagram.
3. Rename the newly added simple type to `SizeType`.



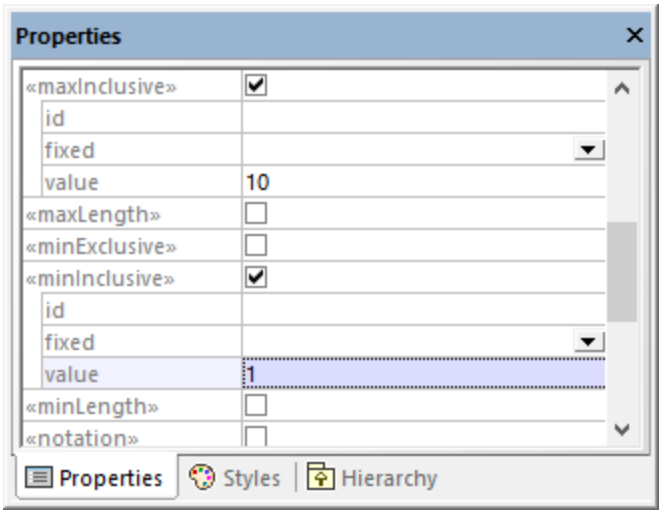
4. Click inside the Model Tree and press **Ctrl+F**. The Find dialog box appears. Start typing "integer" and locate the `integer` type from the "XSDDataTypes" package of the "XSD Profile".
5. Drag the `integer` type into the diagram.



6. Click the **Restriction** toolbar button and drag the cursor from `SizeType` to `integer`. This creates the restriction relationship; see also Creating Relationships 136.



7. To define the `minInclusive` and `maxInclusive` values, select the simple type and edit the properties with the same name in the Properties window.

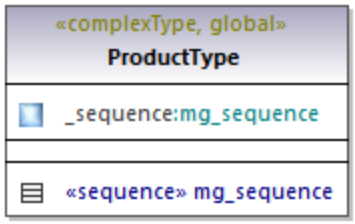


© 2019-2025 Altova GmbH

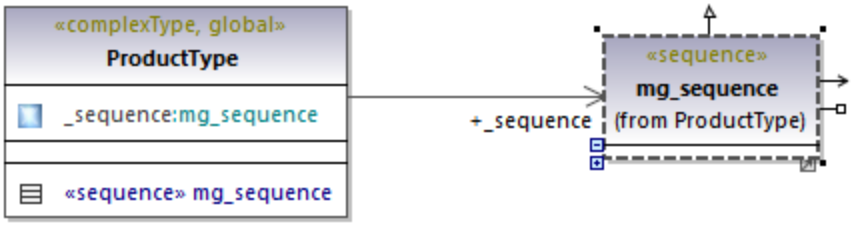
Add a complex type

The following steps add the `ProductType` complex type to the XML schema. All these steps take place in the **MainDiagram** as well.

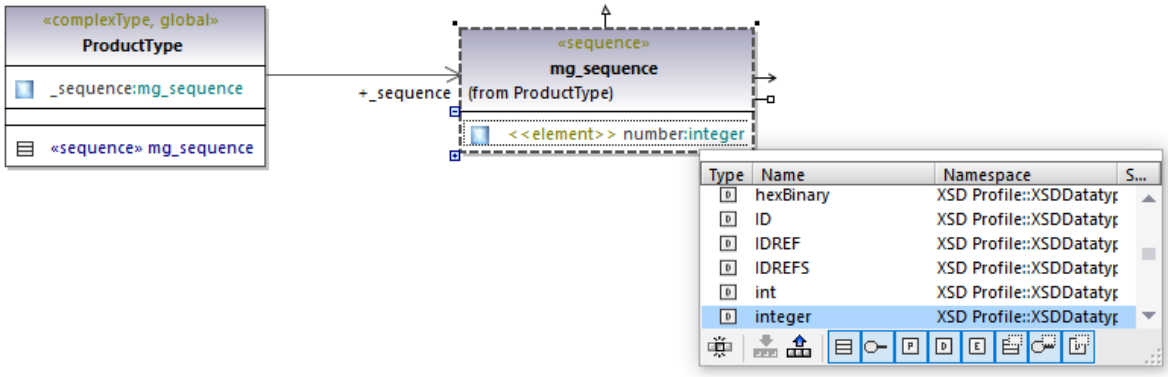
1. Click the **XSD Complex Type** toolbar button, and then click inside the diagram.
2. Rename the complex type to `ProductType`.
3. Right-click the complex type and select **New | XSD Sequence** from the context menu.



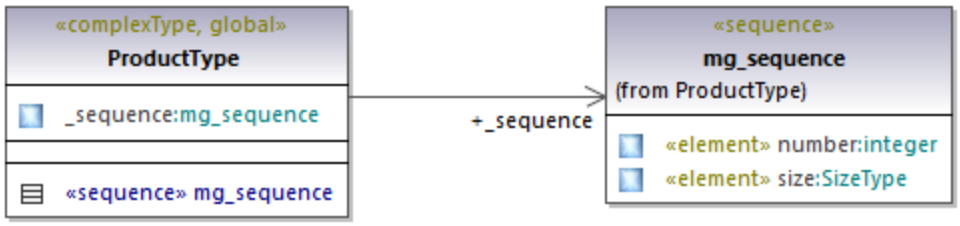
4. Drag the «sequence» class away from the complex type and into the diagram.



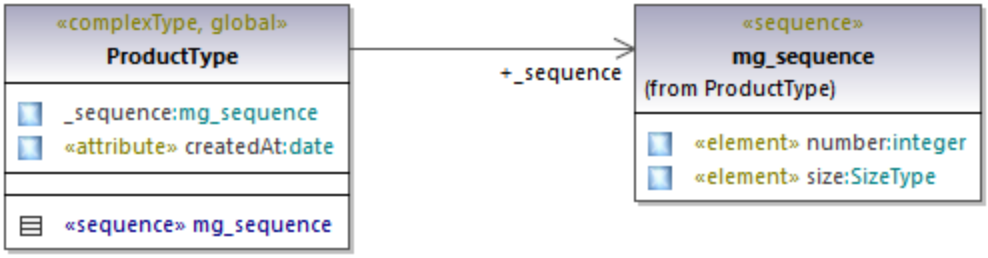
5. Right-click the sequence and select **New | XSD Element (local)**.
6. Change the element's name to **number** and set the type to `integer`. The `integer` type is a base XML Schema type from the XSD Profile. For instructions about setting an element's type, see Type Autocompletion in Classes 134.



7. Using the same steps as above, create the element **size** of type `SizeType`. Note that `SizeType` is the simple type created previously.



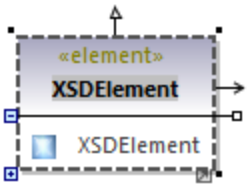
8. Right-click the complex type on the diagram and select New | XSD Attribute (local) from the context window.
9. Change the attribute's name to **createdAt** and the type to `date`.



Add an element

Now that all the required types of the schema have been defined, you can add a product element of type `ProductType`, as follows:

1. Click the **XSD Element (global)** toolbar button, and then click inside the diagram. Notice that a class with the `«element»` stereotype and a single property is added.

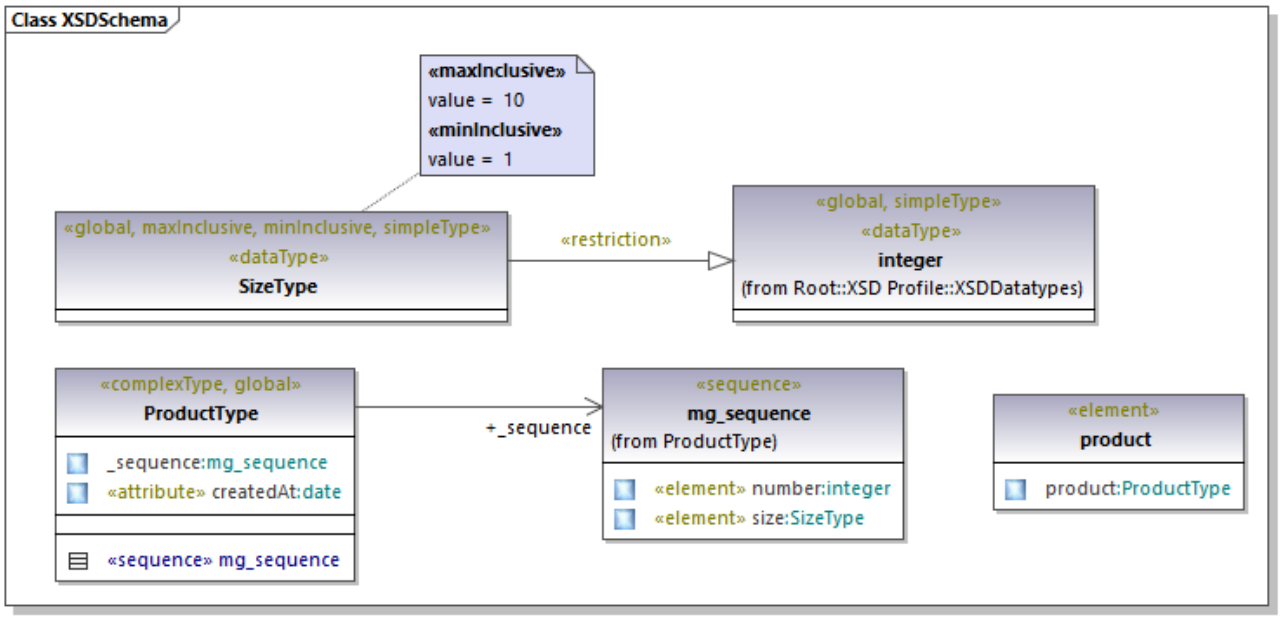


2. Rename the property to **product** and change its type to `ProductType`.

Completed design

The steps above conclude the design part of the schema. By now, your full schema design should look as follows:

© 2019-2025 Altova GmbH

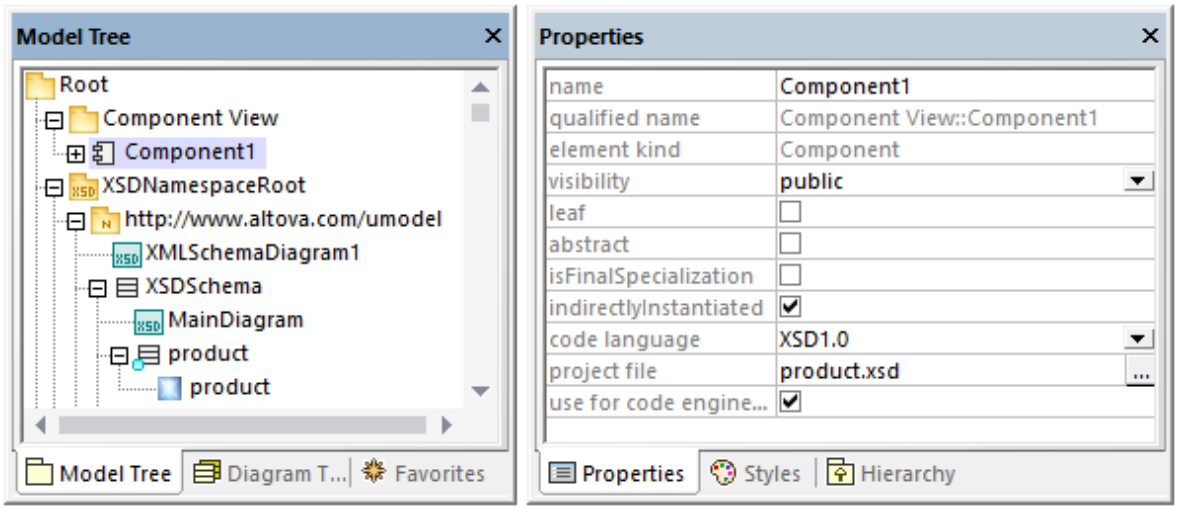


Enable code engineering

To make it possible to generate a schema file from the model, let's now add a code engineering component that provides the schema generation details. The code engineering component is similar to other UModel project kinds, see also Adding a Code Engineering Component 171.

Right-click the "Component View" package in the Model Tree and add a new element of type **Component**. Make sure to change the component's properties as shown below:

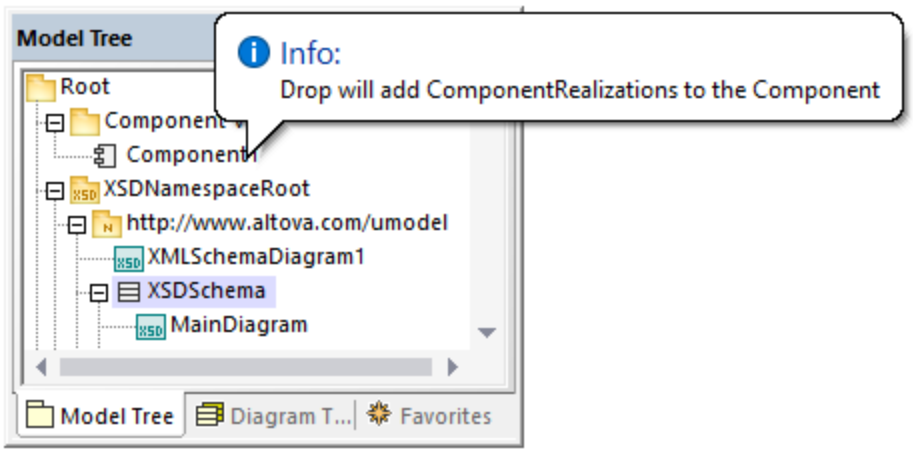
1. The **use for code engineering** property must be enabled.
2. The **code language** property of the code engineering component must be set to "XSD 1.0".
3. The **project file** property of the code engineering component must point to the schema file that is to be generated (in this example, **product.xsd**).



Note: If a **project file** property is missing, enter **product.xsd** in the **directory** property and press **Enter**. A message box should now appear asking you to refer to a project file instead. Click **Yes** to confirm.

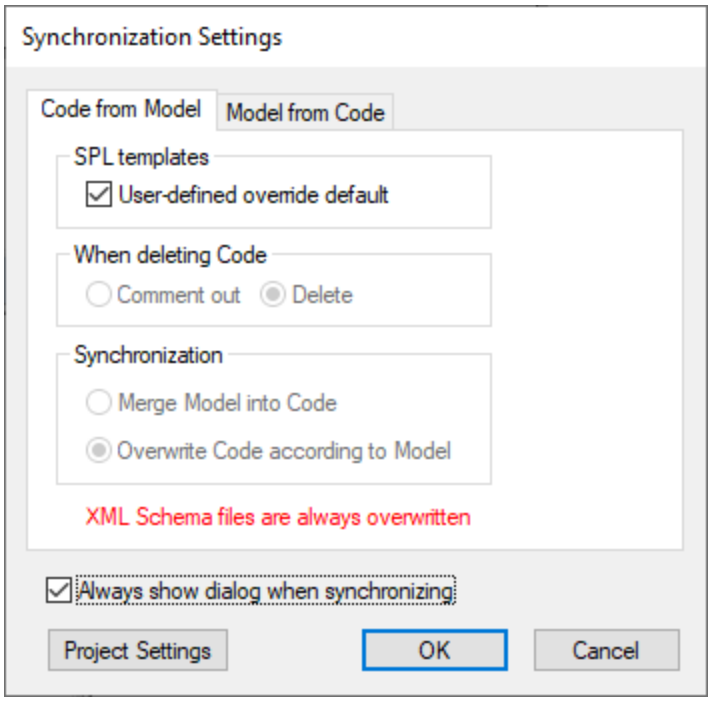
Finally, the XML Schema must be realized by the code engineering component, as described in Adding a Code Engineering Component 171. For the scope of this example, the quickest way to create the **ComponentRealization** relationship is as follows:

- In the Model Tree, drag the **XSDSchema** schema over the code engineering component (**Component1**) and drop it when a tooltip appears such as the one below:



You can now generate the schema file. To do this, either press **F12** or select the **Project | Overwrite Program Code from UModel project** menu command. Note that merging is not supported in case of XML Schemas; therefore, the dialog box shows a message in red to state this fact.

© 2019-2025 Altova GmbH



The new XML schema will be generated in the same folder as your UModel project.

8.3.2 Business Process Modeling Notation 1.0 / 2.0

Altova website: Business Process modeling in UModel

BPMN is a standardized flow-chart notation which shows business processes as a workflow and is easily understandable by all involved in the business process. UModel supports BPMN versions 1.0 and 2.0. Both BPMN 1.0 and BPMN 2.0 diagrams can coexist in the same UModel project. Conversion from BPMN 1.0 to BPMN 2.0 can be done at any time.

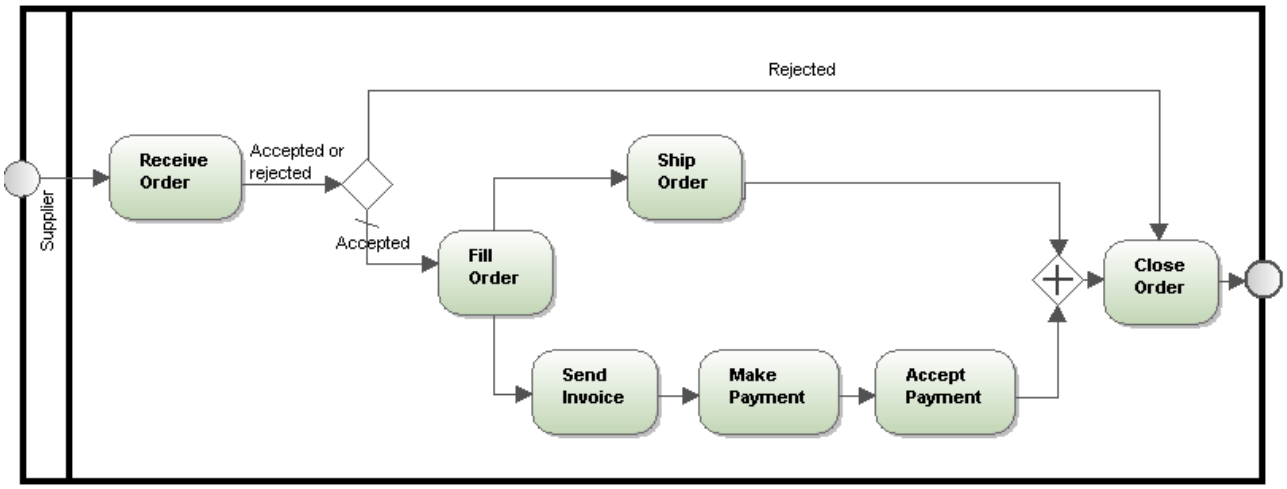
There are four basic element BPMN categories:

Flow objects 442	Events, Activities (Tasks or Sub-Processes), Gateways
Connecting objects 451	Sequence flow, Message Flow, Association
Swimlanes 453	Pool, Lane
Artifacts 454	Data Objects, Group, Text Annotation

Inserting BPMN diagrams and BPMN objects works in exactly the same way as inserting modeling elements in UModel.

Objects can be inserted using the icon bar; associations to other objects can be directly created by clicking on the object "handles" and dragging the connector to the target object. Properties can be viewed and set using the Properties Window[89].

Note that you can create multiple layers per BPMN diagram, see Adding Layers to Diagrams[132].



To convert BPMN 1.0 diagrams to BPMN 2.0 diagrams:

- Right-click in a BPMN 1.0 diagram and select the option **Convert to BPMN 2.0 diagram**. If more than one BPMN 1.0 diagram exists in the same package, you will be prompted to convert all of those in that package.

  A second prompt appears, asking if you want to include the BPMN 2 Profile to the project. Clicking OK converts the diagrams.

8.3.2.1 Flow objects

Flow objects are the graphical elements that define the behaviour of a business process. There are three Flow Objects: Events, Activities and Gateways.

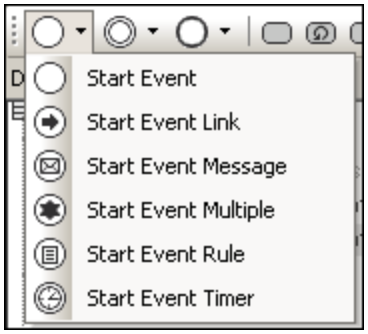
Events

An event is something that occurs during a business process and is represented by a circle. Events affect the flow of the process and generally have a cause (trigger) and a result. There are three different types of events: start, intermediate or end, where each group has its own drop-down combo box.

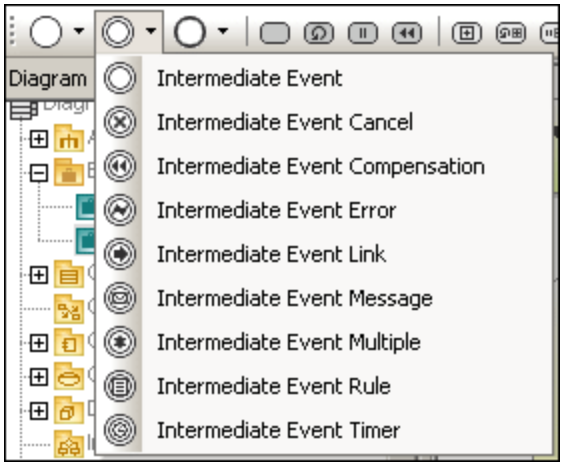
To insert an Event:

1. Click the combo box to open the drop-down list of the type of event you want insert.
2. Select the specific Event and click in the diagram tab to insert it.

© 2019-2025 Altova GmbH

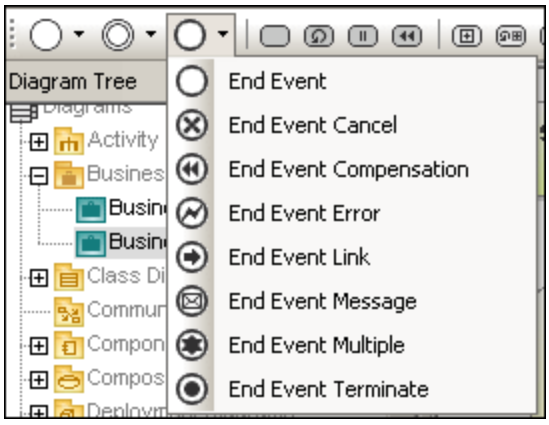


Start Event



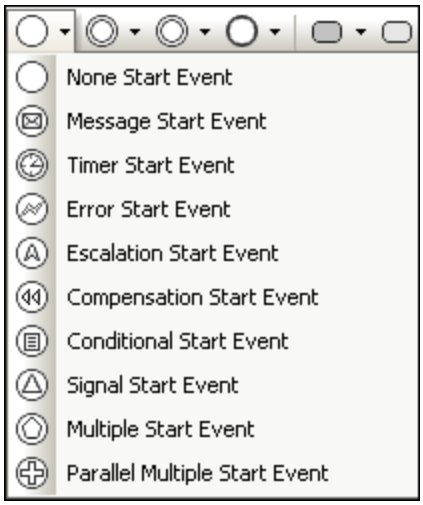
Intermediate Event

Intermediate events can be attached to the boundary of a Task or Sub-Process, and show that the activity is to be interrupted when the event is triggered.

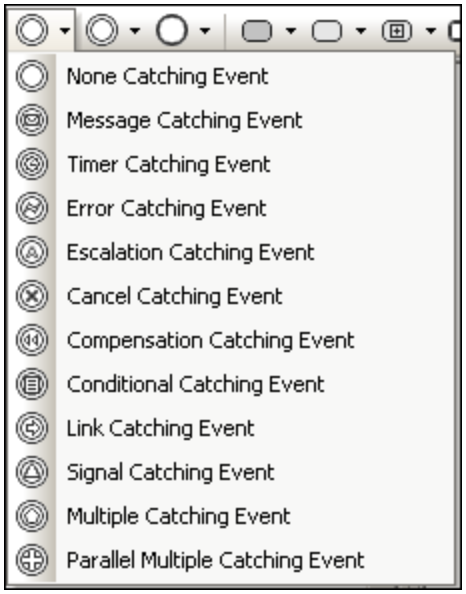


End Event

BMPN 2.0 Events



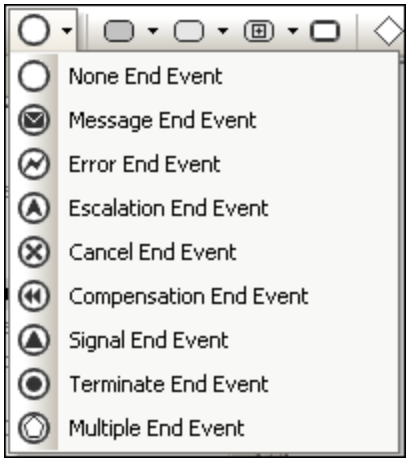
Start Events



Catching Events

© 2019-2025 Altova GmbH

Throwing Events



End Events

Activity

Activities are actions that are performed during a business process, and are represented by rounded rectangles. Process models can contain the following types of activity: Process, Sub-Process and Task. Activities can occur singly or multiple times within a loop.

To insert an Activity:

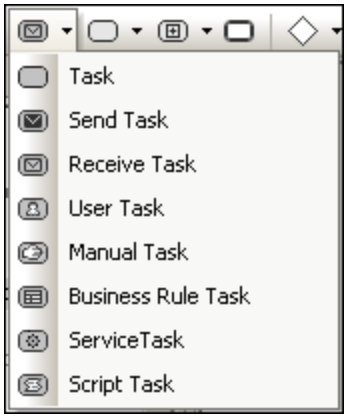
1. Click the specific Task or Sub-Process icon of the icon bar.
2. Click in the diagram tab.

Activity - Task

Tasks are activities that are included in a process. Tasks cannot be broken down into lower level subtasks, they are atomic.

Loop Task	
Multi Instance Task	
Compensation Task	

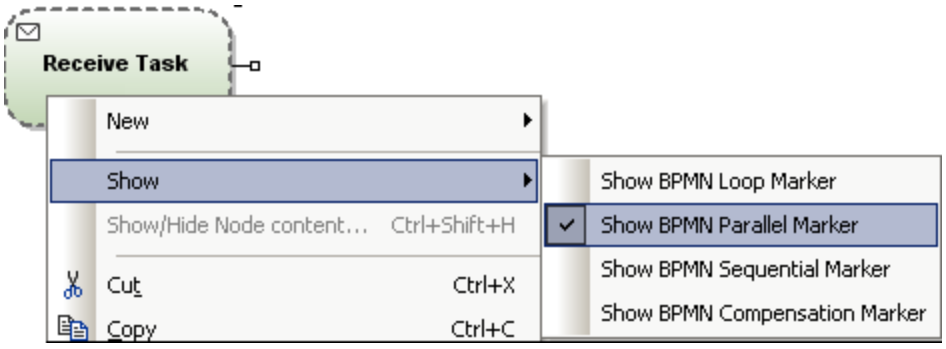
BPMN 2.0 Tasks



To define a Loop, Parallel, Sequential or Compensation marker:

- Right click the inserted task and select the specific marker, e.g. **Show | Show BPMN Parallel Marker**.

© 2019-2025 Altova GmbH



Note: You can also define the Marker in the Properties tab under the "MultiInstanceLoopCharacteristics entry.

Activity - Sub-Process

A Sub-Process is a compound activity that is included in a process, and allows hierarchical business process model development. A Sub-Process can be broken down into finer detail through various sub-activities.

A collapsed Sub-Process 450 is displayed as a top-level element, where the details of the sub-process are not visible. A "plus" icon in the element shows that an additional layer of complexity exists.

An expanded Sub-Process 448 displays the details of the Sub-Process within its boundaries. Note that a sequence flow cannot cross the boundary of a Sub-Process.

Gateway

Gateways are used to determine how Sequence Flows branch and merge within a process. Gateways are always shown as a diamond (*see table below*).

Inclusive Gateway (OR)	
Parallel Gateway (AND)	
Data Based Exclusive Gateway (XOR)	
Event Based Exclusive Gateway (XOR)	
Complex Gateway (Decision/Merge)	

BPMN 2.0 Gateways

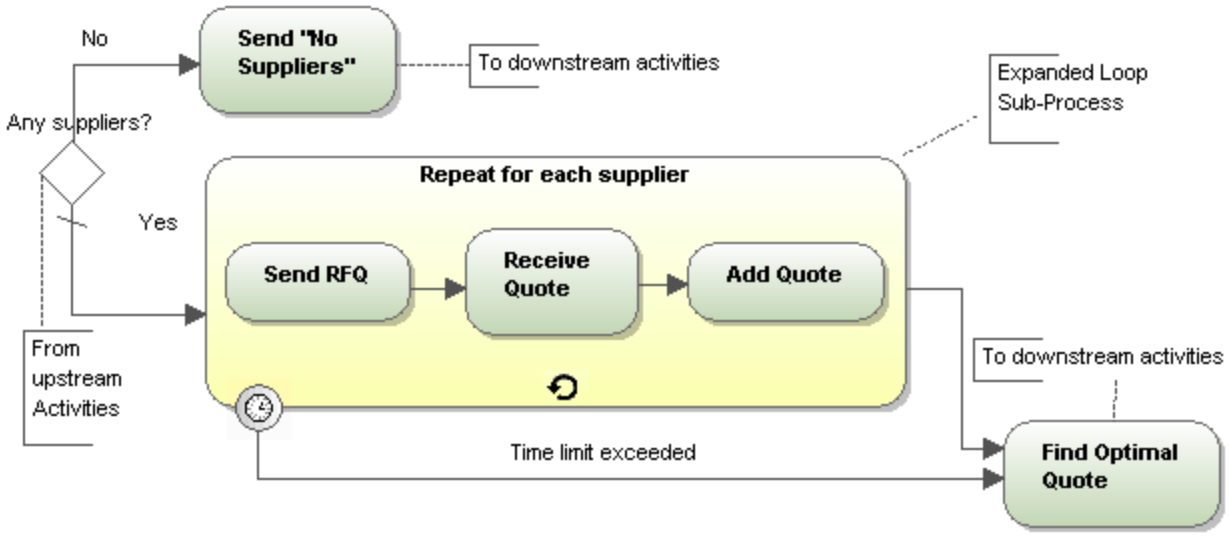
The screenshot below shows supported BPMN 2.0 Gateways. UModel allows you to show an Exclusive Gateway with or without an X. To see the icon with an X, set the value `showXIcon` of an Exclusive Gateway to `true`.

8.3.2.1.1 Expanded Sub Processes

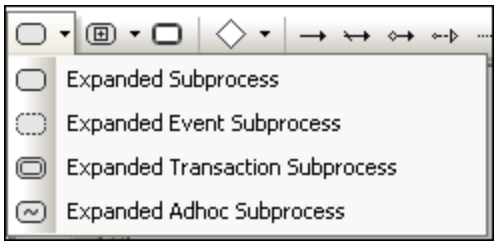
Expanded versions of sub processes show the process detail within the element boundaries.

Expanded Sub-Process	
Expanded Loop Sub-Process	
Expanded Multi Instance Sub-Process	
Expanded Ad Hoc Sub-Process	
Expanded Compensation Sub-Process	

© 2019-2025 Altova GmbH

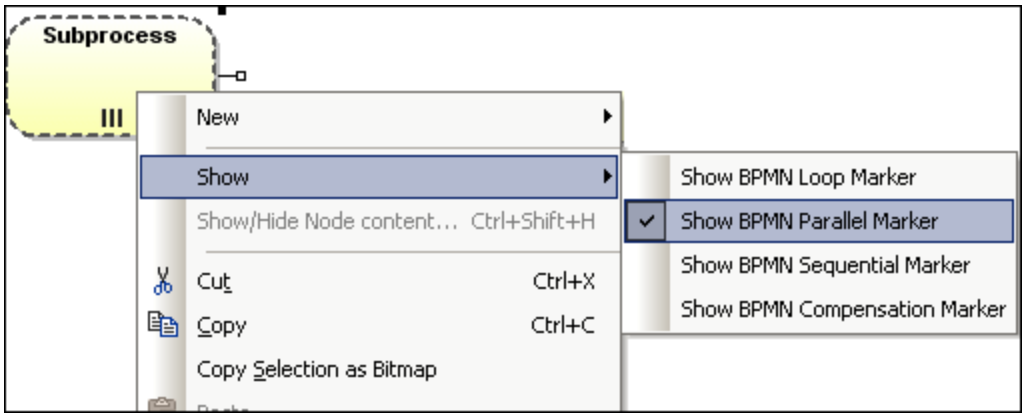


BPMN 2.0 Expanded Sub Processes



To define a Loop, Parallel, Sequential or Compensation marker:

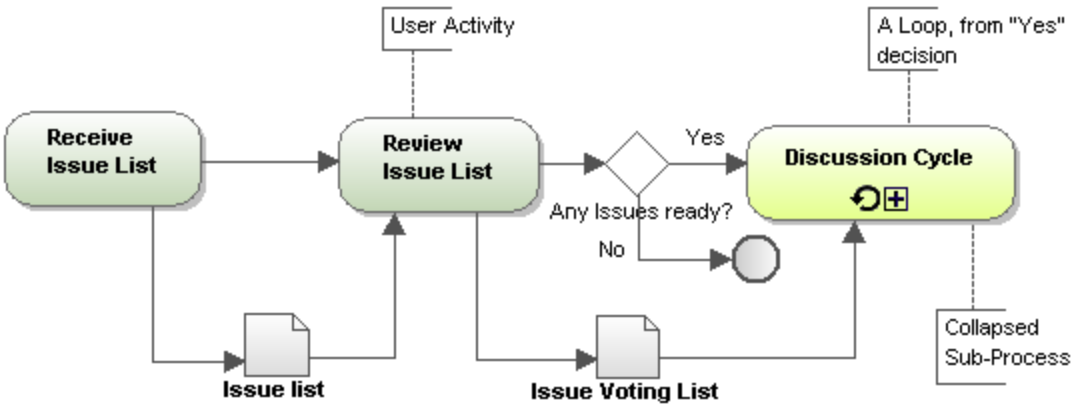
- Right-click the inserted task and select the specific marker, e.g. **Show | Show BPMN Parallel Marker**.



8.3.2.1.2 Collapsed Sub Processes

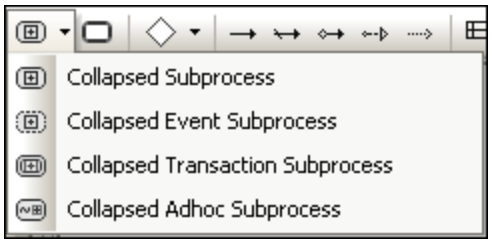
Collapsed versions of sub-processes hide the process detail. The specific type of Sub-Process is shown by the icon within the Sub-Process element.

Collapsed Sub-Process	
Collapsed Loop Sub-Process	
Collapsed Multi Instance Sub-Process	
Collapsed Ad Hoc Sub-Process	
Collapsed Compensation Sub-Process	



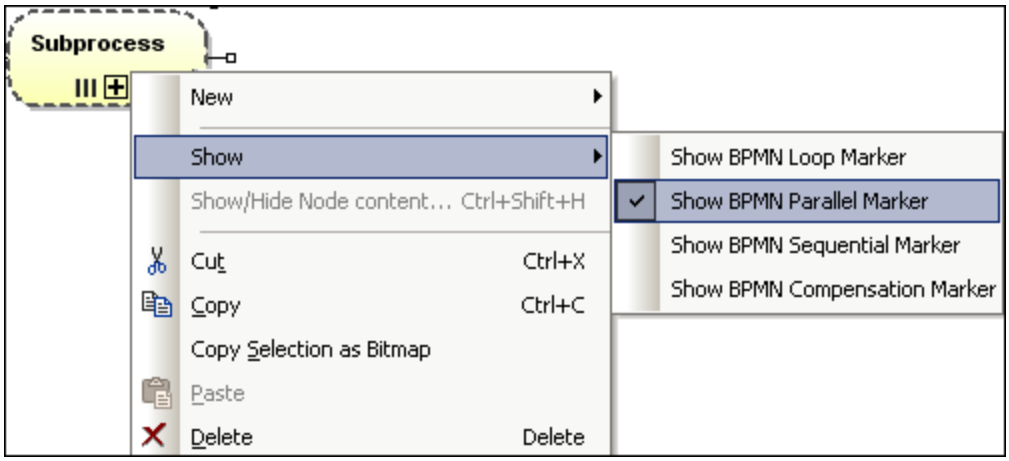
© 2019-2025 Altova GmbH

BPMN 2.0 Collapsed Sub Processes



To define a Loop, Parallel, Sequential or Compensation marker:

- Right-click the inserted task and select the specific marker, e.g. **Show | Show BPMN Parallel Marker**.

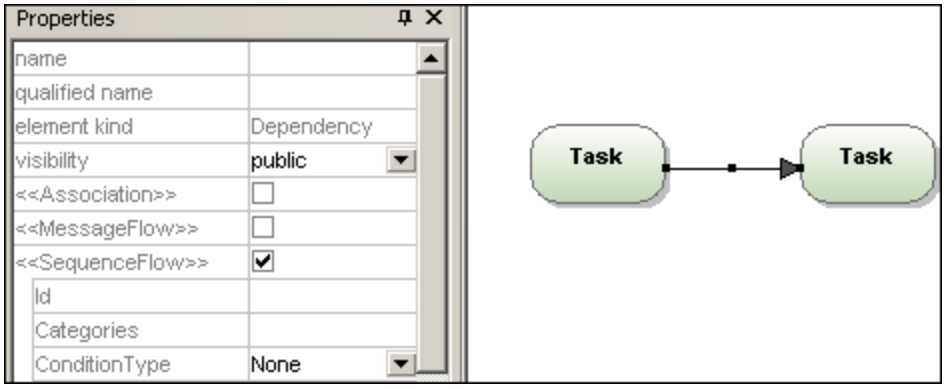


8.3.2.2 Connecting objects

There are two ways of connecting objects: a Flow (using a sequence or message), and an Association.

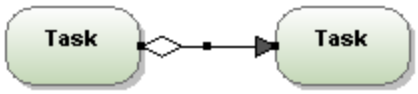
Sequence Flow

A Sequence Flow shows the order that activities are performed within a Process.



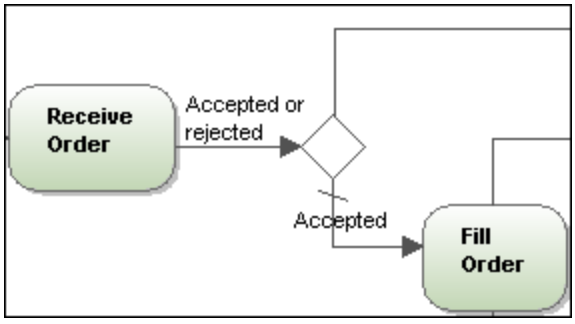
Conditional Flow

This type of Sequence Flow can have a conditional expression which is evaluated to determine if the flow will be used or not. If the conditional flow originates from an activity, then a mini diamond is displayed at the origin of the arrow.



Default Flow

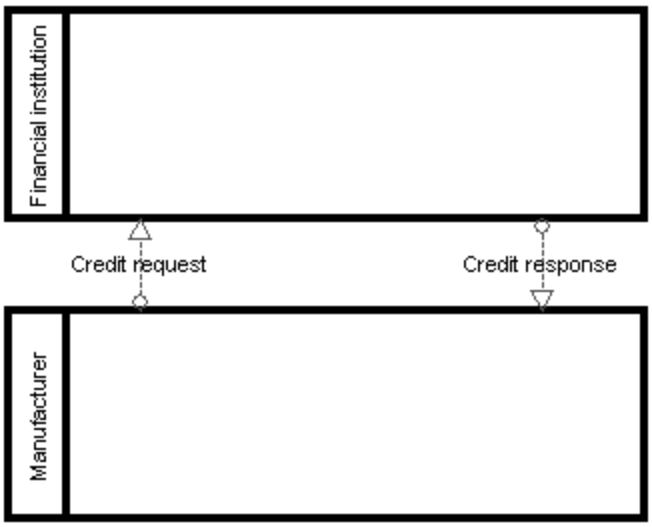
This type of flow is used if all other conditional flows are "false" in Data-Base Exclusive, or Inclusive decisions. A **diagonal slash** at the beginning of the arrow line is used as a visual indication, e.g. "Accepted" default flow.



Message Flow

A Message Flow shows the flow of messages between two participants (entities or roles), that can send and receive them. Participants are shown as separate Pools in the diagram.

© 2019-2025 Altova GmbH

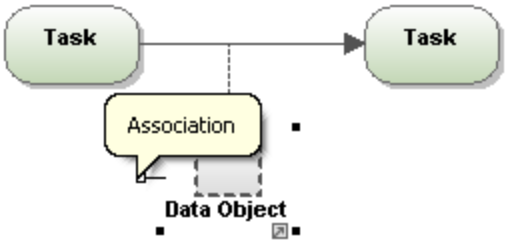


Association

Associations are used to associate Text and non-Flow Object data with Flow Objects, and show how data are input and output from Activities. The diagram below shows a Text annotation which provides the additional information "User Activity" for the Task "Review Issue List".

To create an Association between a Data Object and a Flow control:

1. Click the Association handle of the Data Object (on the left of the object).
2. Drag the connector onto the Flow Control arrow which is highlighted when you can drop it.



Alternatively, click the Association icon and drag from the Data Object to the Flow Control.

8.3.2.3 Pools / Swimlanes

Pool

Pools are used to partition and organize activities. A business process may show the interaction between various processes or participants. Each participant is represented by a rectangular box called a Pool. A participant could be a business role or entity.

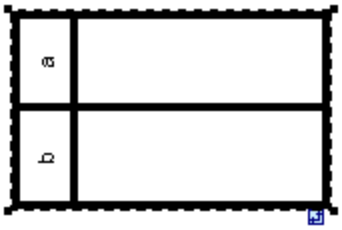
- BPMN objects placed within a pool become part of it when the pool boundary is highlighted.
- Objects within a pool can be individually selected using **Ctrl+Click**, or by dragging the marquee inside the pool.
- Click the pool boundary, or title, and drag to reposition it.

Lane

Pools can be further subdivided into Lanes, which categorize activities within a pool. Note that both horizontal and vertical lanes can be defined.

To add a new lane to a pool:

- Right-click the header of an existing pool object and select **New | Lane**. This adds a new lane to the pool. Each lane can be named separately, by double clicking in the name field.



Note: Right clicking in one of the lanes allows you to add any of the elements allowed to be placed in a pool using the New option.

8.3.2.4 Artifacts

Artifacts allow you to show additional information about a Process i.e. how data, documents and other objects are used and updated during the business process. Artifacts are not directly related to sequence, or message flow, of the process.

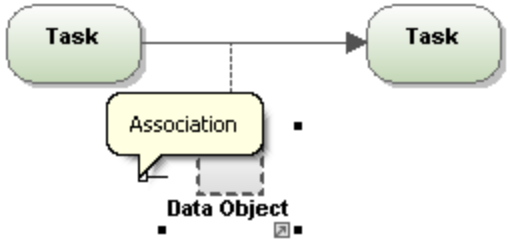
Data Object

Data Objects are documents or other types of data, that show how data are used during a business process. Data objects can be used to define the input and output of data to/from activities.

© 2019-2025 Altova GmbH

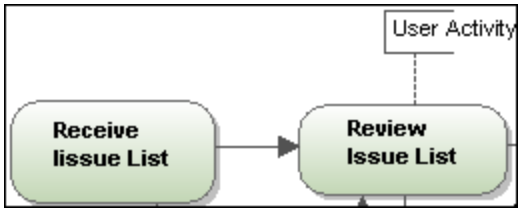
To create an Association between a Data Object and a Flow control:

1. Click the Association handle of the Data Object (on the left of the object).
2. Drag the connector onto the Flow Control object which is highlighted when you can drop it.



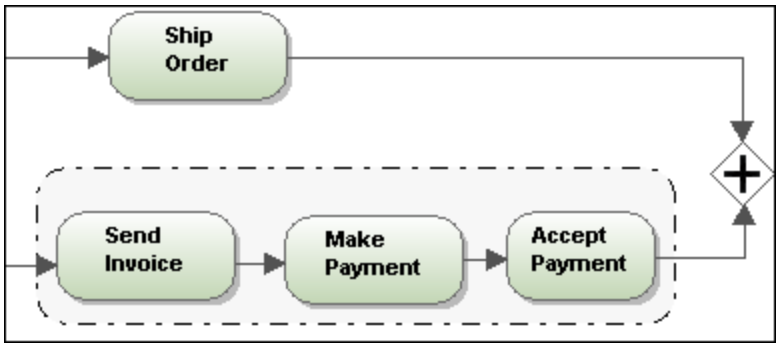
Text Annotation

Text Annotations allow you to annotate various sections of a business process and are connected to the specific object using an association.



Group

Groups are often used to highlight certain sections of a diagram, even across different pools. Groups cannot connect to a sequence or message flow. Group objects are generally placed behind task or process objects in the diagram.

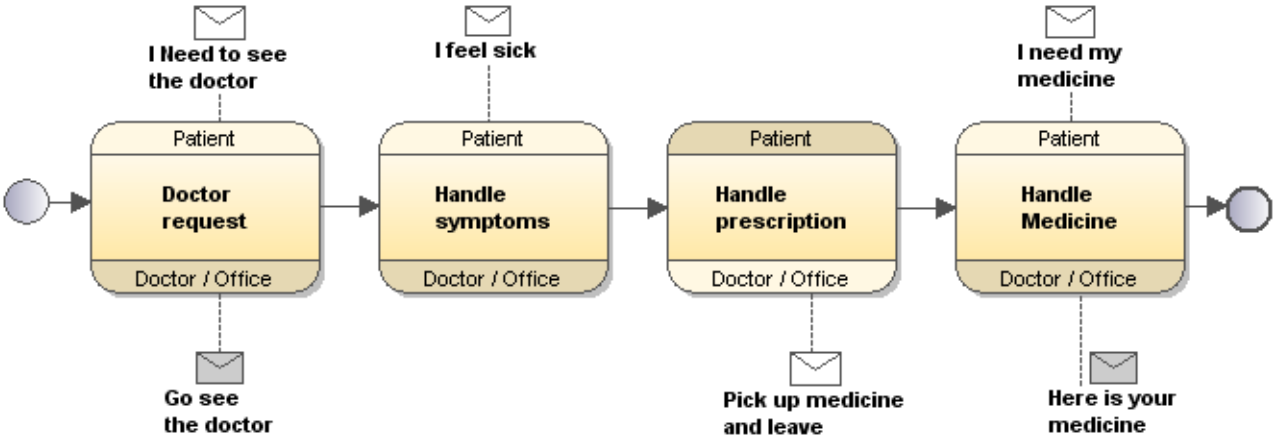


8.3.2.5 Choreography diagram

Choreography Diagrams specify the way business participants **coordinate** their interactions. They can also be seen as a business contract between participants, where the focus lies on the exchange of information (Messages) between the participants.

Business contracts are often in the form of a purchase order sent to a supplier, the confirmation by the supplier to process the order, then the fulfilling of the order. Choreographies also have Activities ordered by Sequence Flows.

Activities comprise of one or more interactions between the various participants. Interactions are often called Message Exchange Patterns (MEP). A MEP is the "Activity" of a choreography, and can also be called a **Choreography Task**.



8.3.2.5.1 Choreography Tasks

There are four types of choreography tasks that can be inserted into the diagram:

Choreography Task

Sub Choreography Task (collapsed)

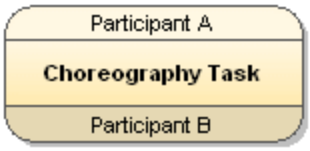
Sub Choreography Task (expanded)

Call Choreography Task

To insert a choreography task:

1. Click the Task icon of the Task that you want to insert, e.g. Choreography Task, then click in the Choreography Diagram.

© 2019-2025 Altova GmbH

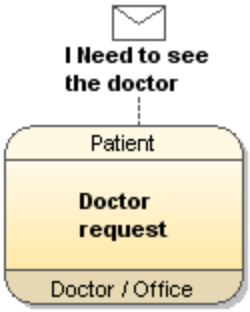


2. The screenshot shows the default view when the Task is inserted; the "Choreography Task" text is automatically highlighted.
3. Enter text to rename the Choreography Task.
4. Click in the top **band** to enter the name of Participant A, and in the bottom band to name Participant B. The Participant bands are shown as shaded/unshaded. The **initiator** of the Activity is the **unshaded** Participant, which is Participant A when the Task is first inserted.

To add/associate messages to a Choreography Task:

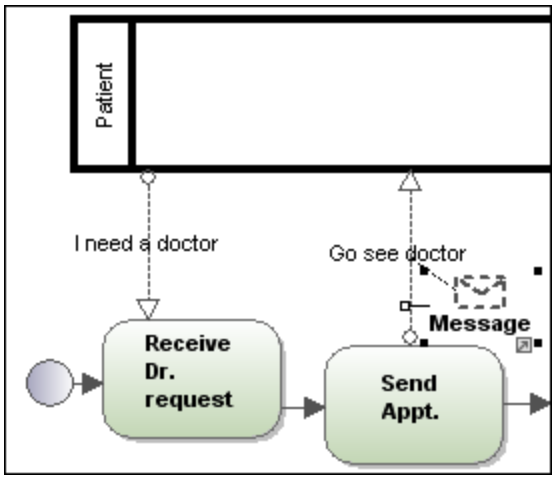
1. Click the message icon in the icon bar, then click in the diagram to insert it.

2. Enter the name of the message, e.g. "I need to see the doctor".
3. Click the **Association** handle (on the left) and drag it to the Choreography Task you want to associate it to.



To add a message to a line e.g. association:

1. Click the **line** that you want to add the message to.
2. Click the Message icon in the icon bar.
3. Click the same line again to attach it.

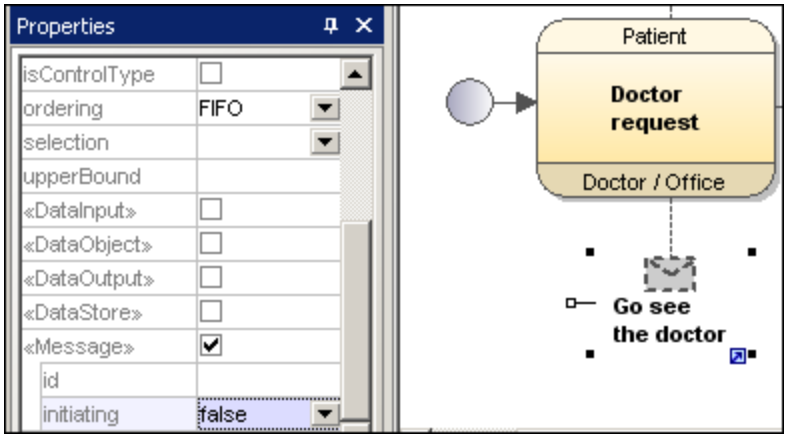


The message is placed on top of the line and automatically attached to it.

To change the initiating Message / Participant:

- When inserting a **Message,** it will automatically be defined as the initiating message, i.e. it is unshaded.

1. Click the Message and select **false** from the "initiating" combo box, in the **Properties** tab.

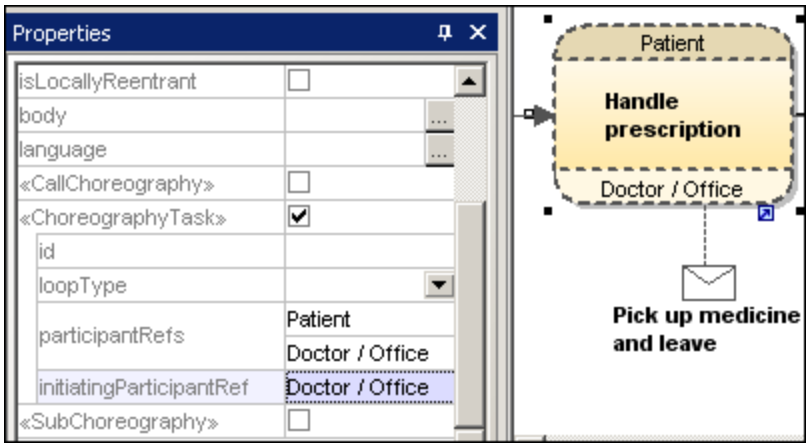


The message element is now shaded.

- When inserting a **Choreography Task**, Participant A is automatically defined as the Initiating Participant.

1. Click the Choreography Task that contains the Participant you want to be the initiator.
2. Enter the name of the Participant you want to define as the initiator in the "InitiatingParticipantRef" combo box, e.g. "Doctor / Office".

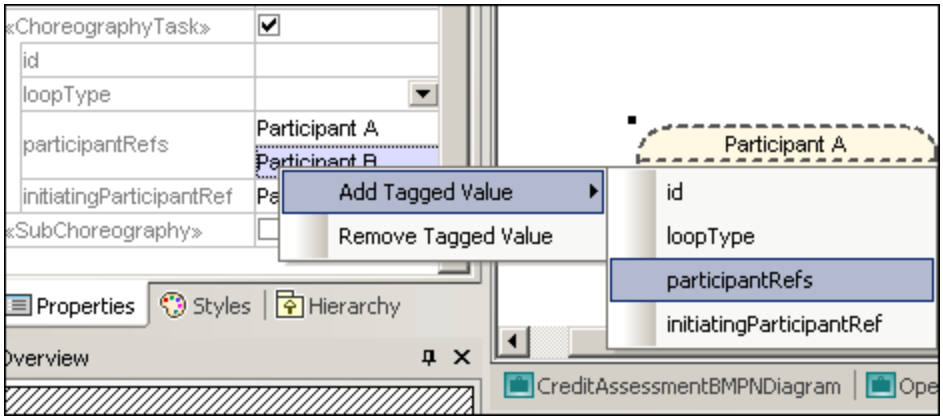
© 2019-2025 Altova GmbH



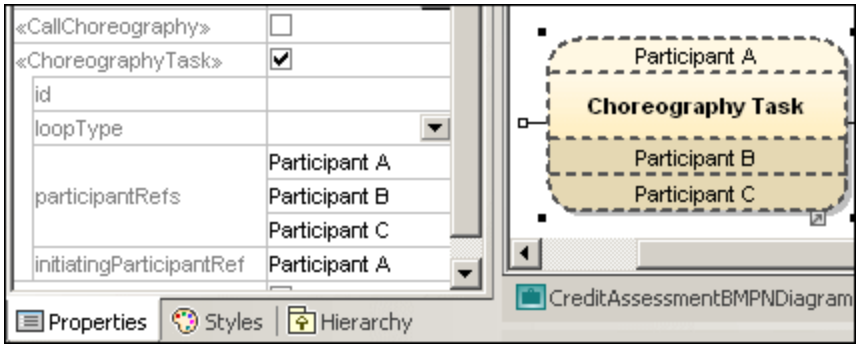
The Doctor / Office band is now unshaded, showing that it is the Initiating Participant. The Patient band is now shaded.

To add new Participants to a Choreography Task:

1. Click the Task you want to add the Participant to in the diagram window.

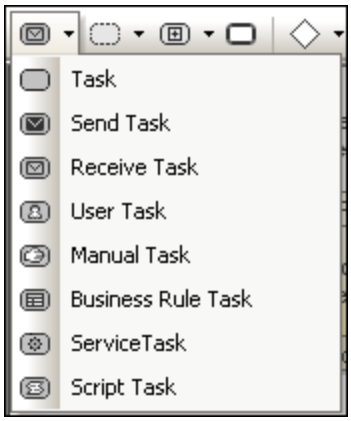


2. Right-click the **participantsRefs** field in the Properties tab and select **Add Tagged Value | participantRefs**.
3. Enter the name of the new Participant e.g. Participant C.

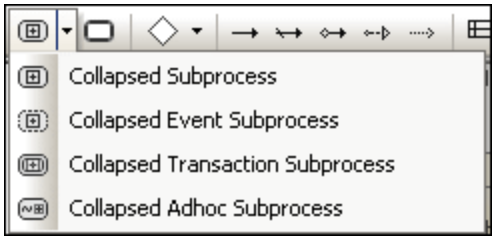


8.3.2.5.2 Tasks and Subprocesses

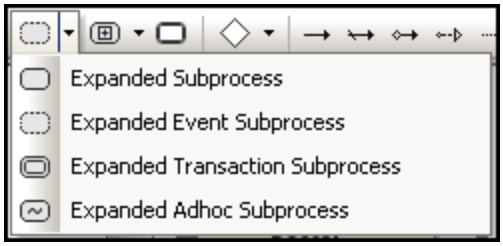
Click the Task drop-down icon to insert the specific Task.



Click the Collapsed Subprocess drop-down icon to insert the specific Collapsed Subprocess.



Use the Expanded Subprocess drop-down icon to insert the specific Expanded Subprocess.



8.3.2.5.3 Data Objects

Data is represented by five modeling elements and are inserted by clicking one of the following icons:

Data Object

Represents information flowing through the process, such as emails, business documents, and so on. Data objects provide information about what activities require to be performed and/or what they produce.

© 2019-2025 Altova GmbH

Data Output

Represents the result of the process.

Data Input

Is an external input for the entire process. Can be read by an activity.

Data Collection

Represents a collection of information, for example, a list of order items.

Data Store

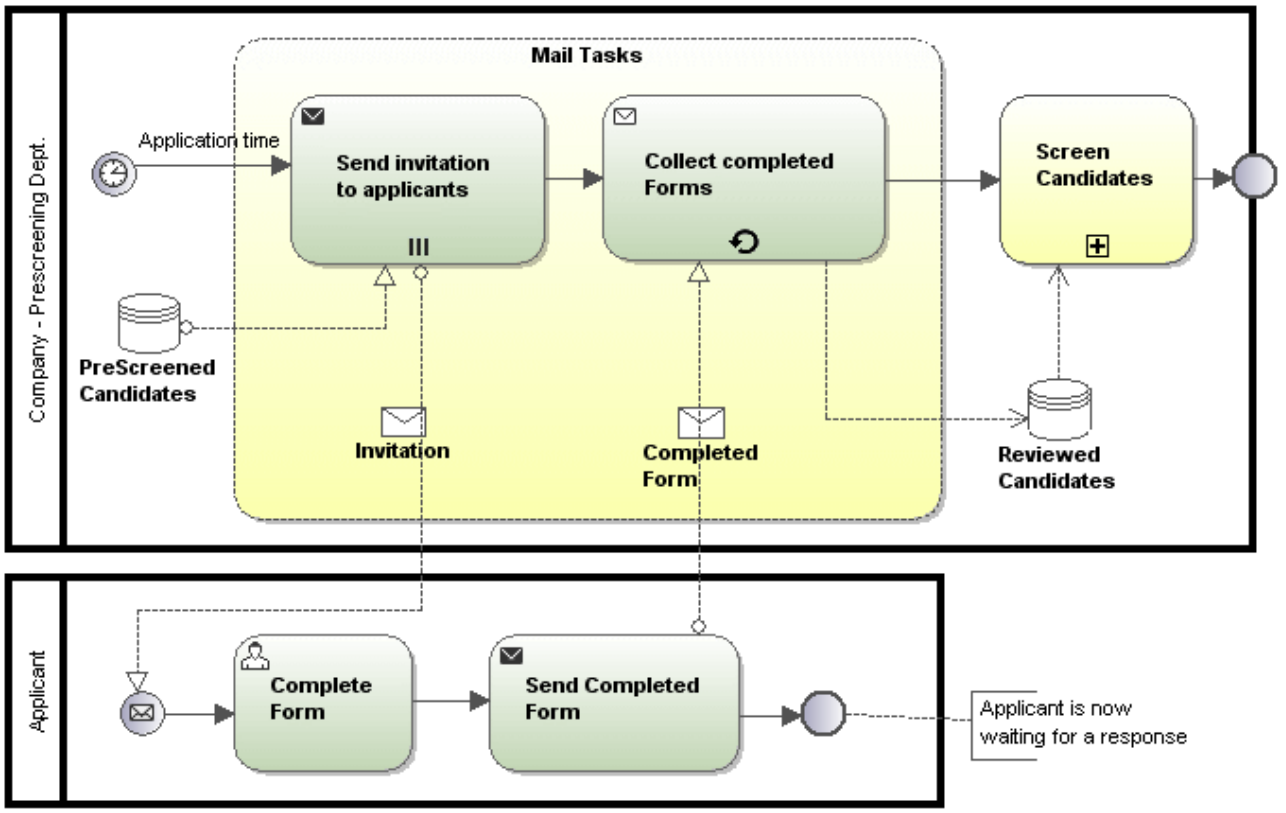
A place where the process can read or write data, for example, a database.

8.3.2.6 Collaboration diagram

Collaboration Diagrams specify the **interactions** between two or more processes.

A Collaboration generally consists of two or more pools which represent the participants in the collaboration. Message exchanges between participants are shown by Message Flows that connect the two pools, or the objects within the pools. Pools may also be empty, in this case they are black boxes.

All combinations of Pools, Processes, and a Choreography are allowed in a Collaboration diagram.



8.3.2.6.1 Conversations

A Conversation is a simplified version of a Collaboration, and has access to the same modeling elements. A Conversation defines a set of logically related message exchanges, where the message exchanges are related to each another reflecting a distinct business scenario, e.g. a request followed by a response.

A Conversation has two other graphical elements not available in any other BPMN diagrams:

- Conversation node elements (Conversation, Sub-Conversation and Call-Conversation)
- Conversation links

Conversation

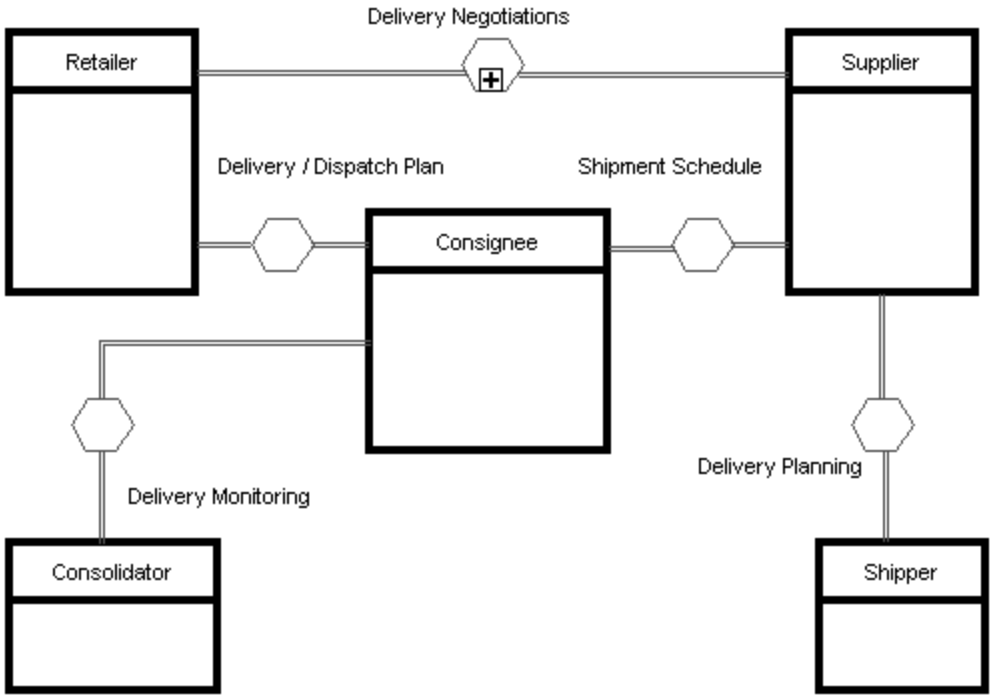
Sub-Conversation

Call-Conversation

Participant / Pool

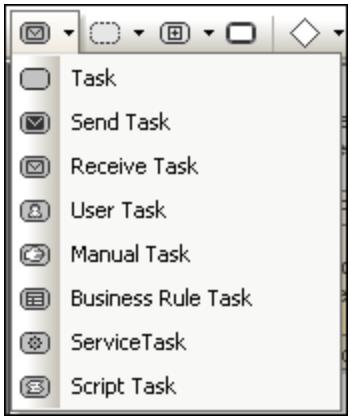
Conversation Link

© 2019-2025 Altova GmbH

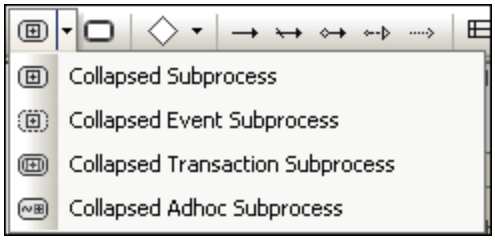


8.3.2.6.2 Tasks and Subprocesses

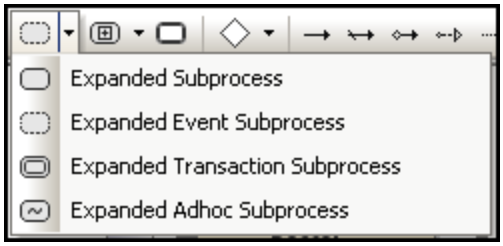
Click the Task drop-down icon to insert the specific Task.



Click the Collapsed Subprocess drop-down icon to insert the specific Collapsed Subprocess.



Use the Expanded Subprocess drop-down icon to insert the specific Expanded Subprocess.



8.3.2.6.3 Data Objects

Data is represented by five modeling elements and are inserted by clicking one of the following icons:

Data Object

Represents information flowing through the process, such as emails, business documents, and so on. Data objects provide information about what activities require to be performed and/or what they produce.

Data Output

Represents the result of the process.

Data Input

Is an external input for the entire process. Can be read by an activity.

Data Collection

Represents a collection of information, for example, a list of order items.

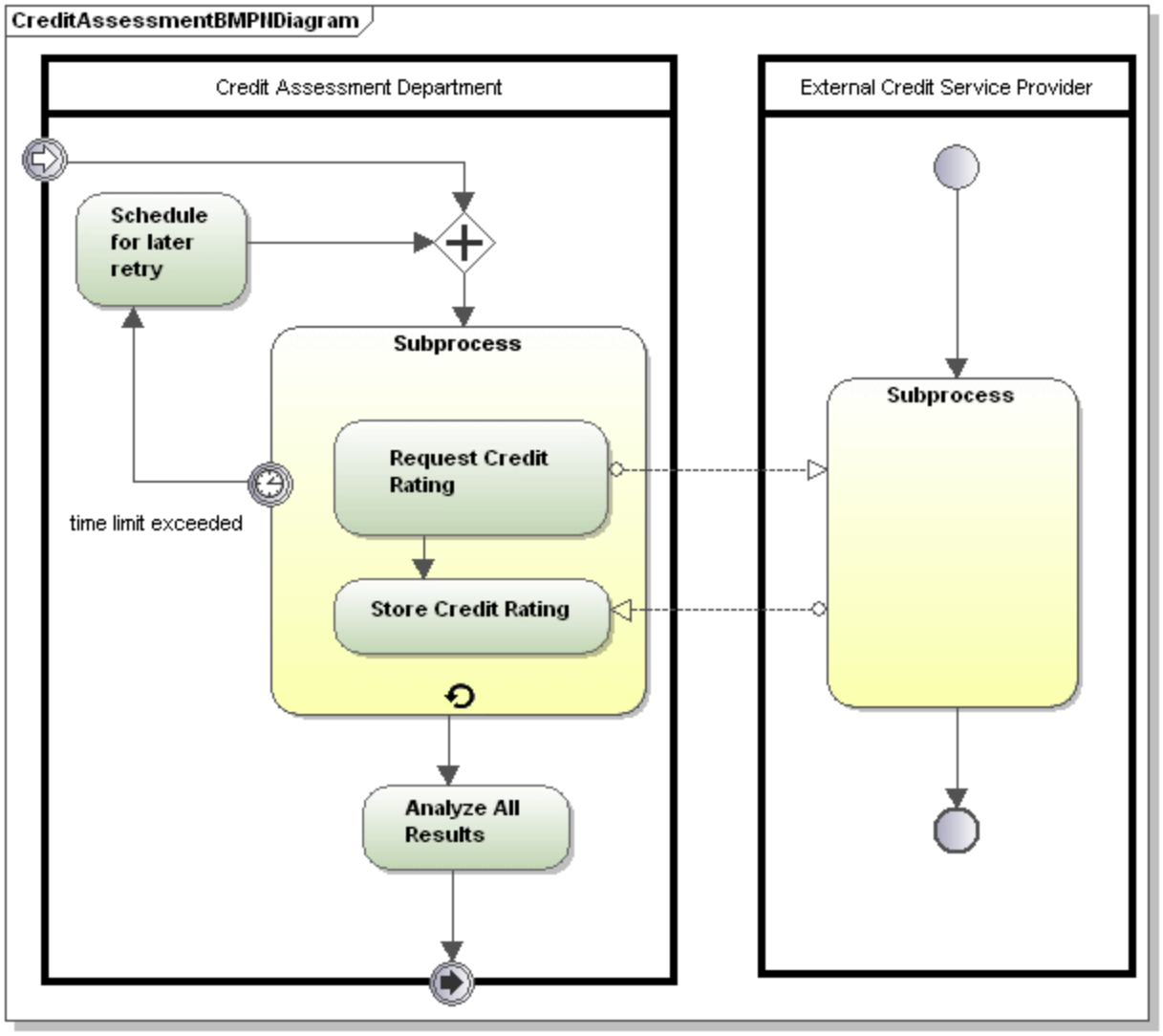
Data Store

A place where the process can read or write data, for example, a database.

© 2019-2025 Altova GmbH

8.3.2.7 Standard Business Process diagram BPMN 2.0

Business Process diagrams cover a wide range of information and employ several different types of modeling, to create Business Processes.



There are three types of Business Processes:

Private non-executable (internal) Business Processes:

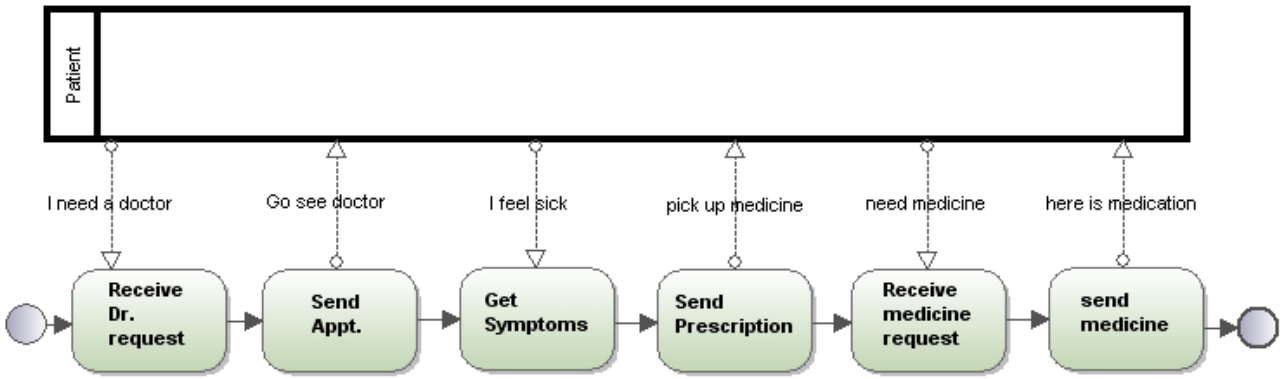
- Non executable Processes are those where there is not enough detail for the process to execute; generally during the development cycle.

Private executable (internal) Business Processes:

- Executable Processes are processes that are executable, due to the fact that they have been completely modeled according to the BPMN 2.0 semantics.

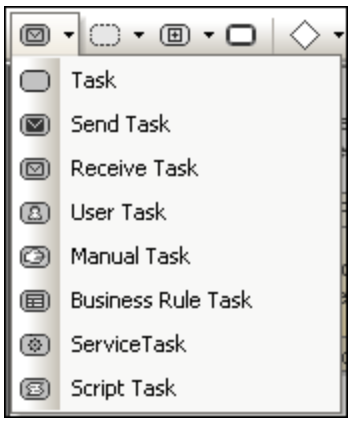
Public Processes:

- Define the interaction between a ***private*** process and a separate process, or participant. E.g. Doctor, Patient interactions.



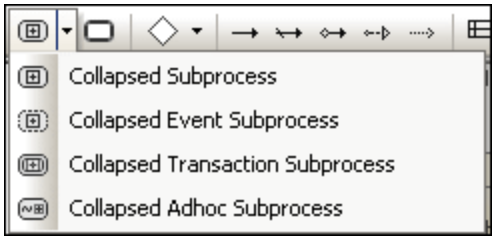
8.3.2.7.1 Tasks and Subprocesses

Click the Task drop-down icon to insert the specific Task.

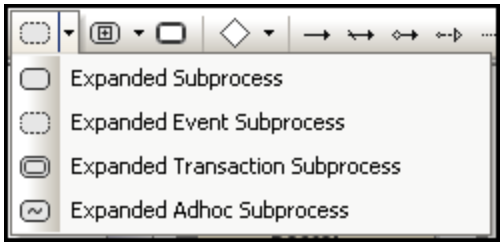


Click the Collapsed Subprocess drop-down icon to insert the specific Collapsed Subprocess.

© 2019-2025 Altova GmbH



Use the Expanded Subprocess drop-down icon to insert the specific Expanded Subprocess.



8.3.2.7.2 Data Objects

Data is represented by five modeling elements and are inserted by clicking one of the following icons:

Data Object

Represents information flowing through the process, such as emails, business documents, and so on. Data objects provide information about what activities require to be performed and/or what they produce.

Data Output

Represents the result of the process.

Data Input

Is an external input for the entire process. Can be read by an activity.

Data Collection

Represents a collection of information, for example, a list of order items.

Data Store

A place where the process can read or write data, for example, a database.

8.3.3 SysML Diagrams

Altova website: Modeling SysML diagrams in UModel

SysML is a graphical modeling language that supports the analysis, specification, design, verification and validation of systems such as hardware, software, data, procedures and others. In UModel, you can create SysML diagrams from scratch, or you can import or export existing SysML models via XMI, see XMI - XML Metadata Interchange 568.

The table below lists the diagrams available in SysML.

Kind	Diagram	Notes	Abbreviation
Structure	Block Definition diagram 469	Modified from UML	bdd
	Internal Block diagram 472	Modified from UML	ibd
	Package diagram 478	Reused from UML	pkg
	Parametric diagram 477	Specific to SysML	par
Requirement	Requirement diagram 480	Specific to SysML	req
Behavior	Activity diagram 481	Modified from UML	act
	Sequence diagram 482	Reused from UML	sd
	State Machine diagram 483	Reused from UML	stm
	Use Case diagram 484	Reused from UML	uc

As illustrated above, SysML diagrams can be broadly classified into structure, requirement, and behavior diagrams. Furthermore, some of the SysML diagrams are reused from the UML, some are modified from the UML, and some are specific to SysML only. The abbreviation indicated for each diagram appears by default in the top-left corner of the Diagram Window 98, unless you choose to hide the diagram's heading.

Aside from the specifics of each diagram, designing SysML projects with UModel is not different from designing standard UModel projects, see Creating, Opening, and Saving Projects 154. An example UModel project that includes various SysML diagrams is available at the following path: **C:\Users\<username>\Documents\Altova\UModel2025\UModelExamples\Bank_SysML.ump**.

Creating SysML diagrams

To create SysML diagrams, your UModel project must include the *SysML profile*, which is a built-in UModel profile. You will be prompted to include this profile when you add the first SysML diagram to your project, as shown below. You can also add the SysML profile explicitly into your project, see Applying UModel Profiles 160.

© 2019-2025 Altova GmbH

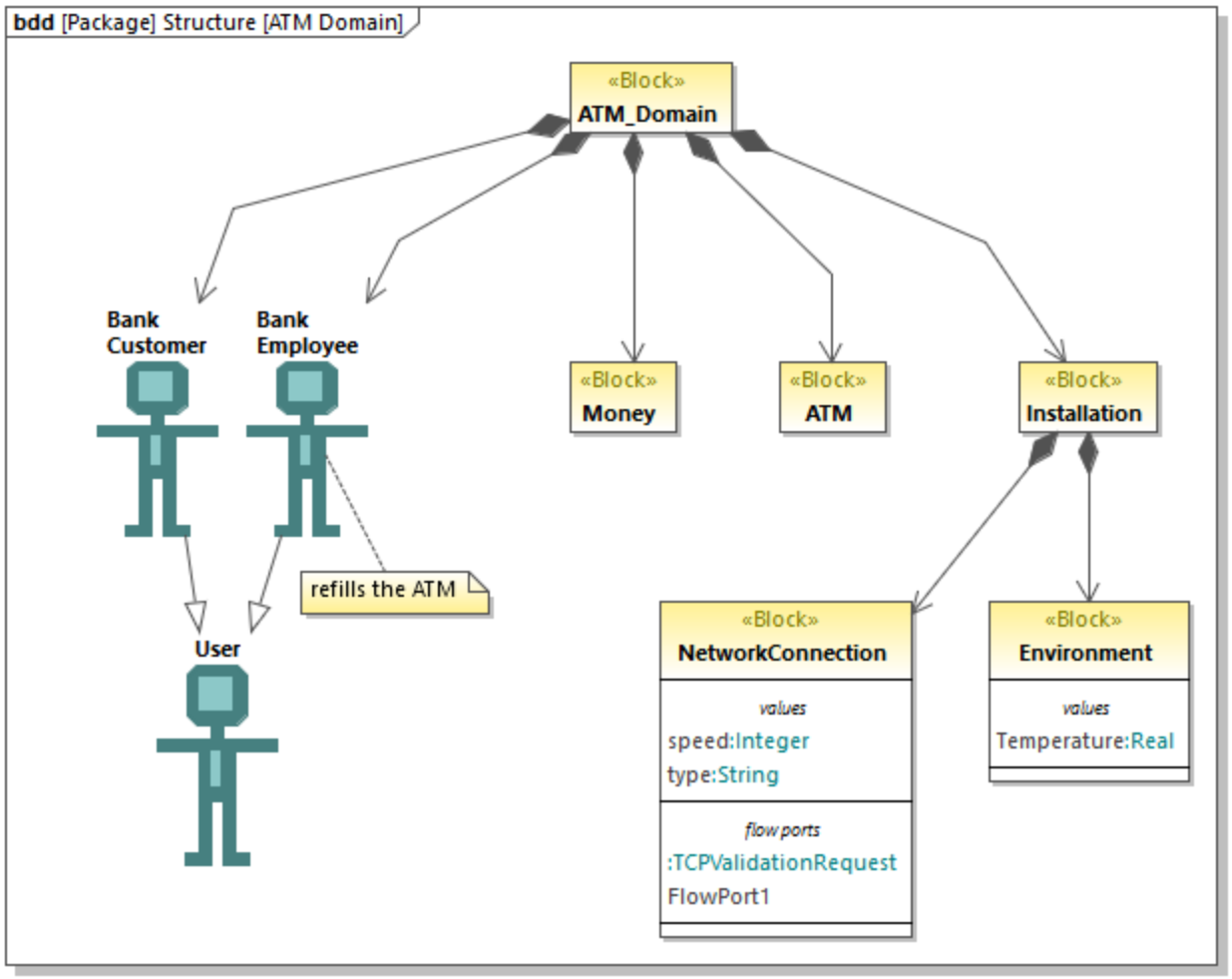
To create a SysML diagram:

1. Do one of the following:
   a. Right-click a package in the Model Tree Window[83] and select **New Diagram | SysML diagrams | <diagram kind>** from the context menu, where "diagram kind" is one of SysML diagram types.
   b. Right-click "Diagrams" or "SysML Diagrams" in the Diagram Tree Window[87] and select **New Diagram | <diagram kind>** from the context menu, where "diagram kind" is one of SysML diagram types. A dialog box opens asking you to select the owner of the diagram. Select a package where the diagram should be stored, and click **OK**.
2. If the current UModel project does not include the SysML profile, a dialog box opens asking you to include it. Click **OK** to include the SysML profile into the current project, see also Applying UModel Profiles[160].

Note: If you selected the "root" package in step 1, SysML diagrams are created in their own "SysML" package.

8.3.3.1 Block Definition Diagram

Block Definition Diagrams are based on the UML Class Diagrams[387], with restrictions and extensions as defined by SysML. The Block Definition Diagram presents structural elements called "blocks", and their relationships, such as associations, generalizations, and dependencies.



Block Definition diagram

Blocks are fundamental units for describing structure in SysML; they are similar to classes in UML class diagrams. Blocks may include components such as parts, operations, properties and ports. A property can be specialized; for example, it can be a **PartProperty**, a **ReferenceProperty**, or a **ValueProperty**.

To create a block:

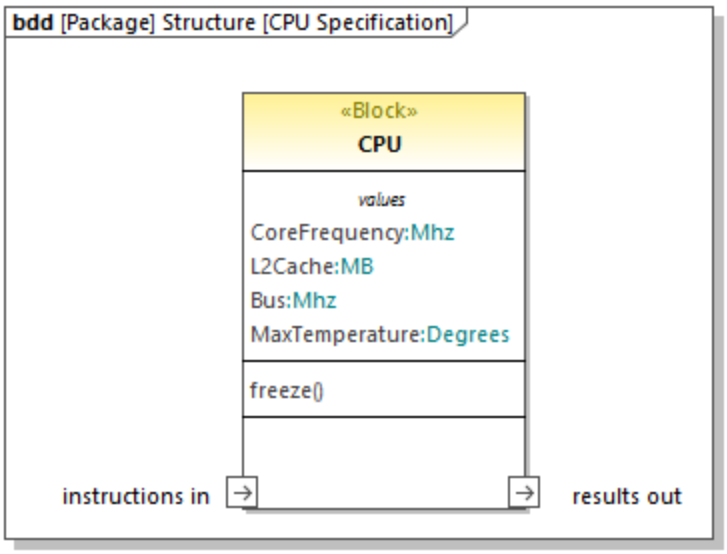
1. Create a new Block Definition diagram, see Creating SysML diagrams 468.
2. Do one of the following:
   - Right-click an empty area in the diagram, and select **New | Block** from the context menu.
   - Click the **Block** toolbar button and then click inside the diagram.

To add a property to a block:

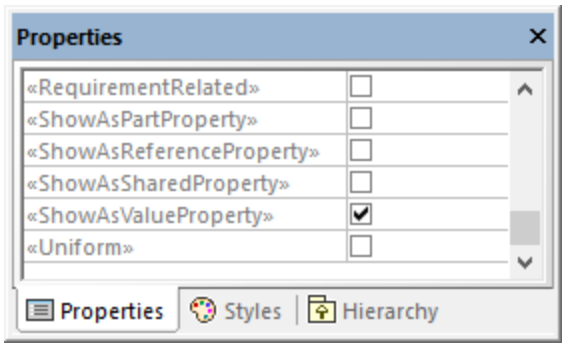
- Right-click an existing block and select **New | Property** (or **PartProperty**, **ReferenceProperty**, **ValueProperty**, as applicable) from the context menu.

© 2019-2025 Altova GmbH

A new compartment is added to the block, for example, "parts" for a **PartProperty**, or "values" for a **ValueProperty**.

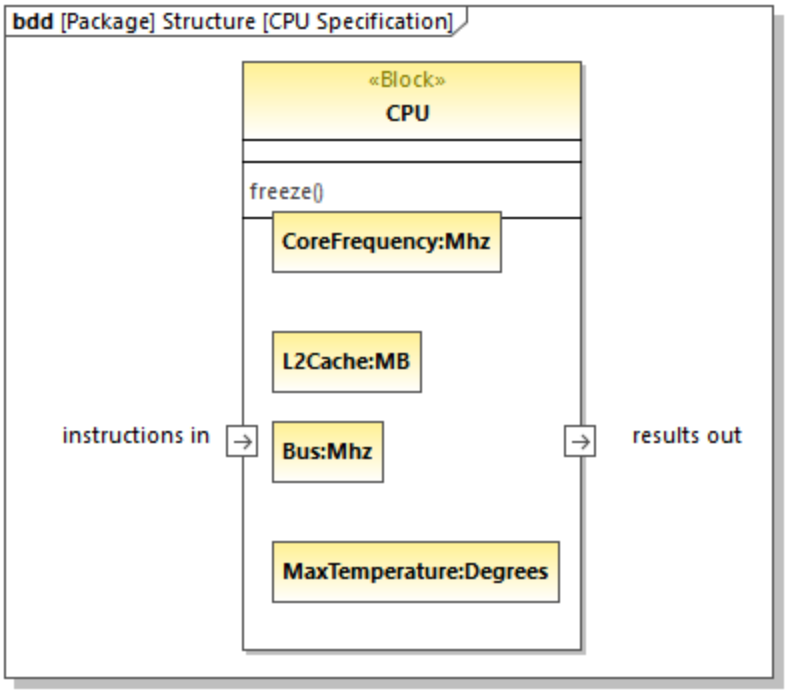


You can change at any time the specialization of an existing property (for example, you can convert a **PartProperty** to a **ValueProperty**). To do this, first select the property on the diagram or in the Model Tree, and then select the check box with an appropriate stereotype in the Properties window, for example:



To show block properties as nodes:

- Right-click a block and select **Show | Show properties as nodes on node**.



To undo the action above, right-click a property (for example, `Bus:Mhz` in the image above), and select **Delete from diagram only** from the context menu.

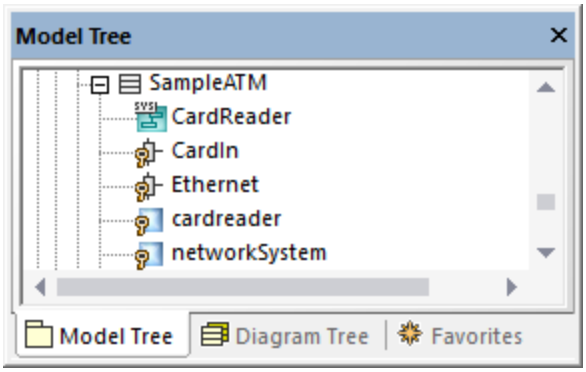
8.3.3.2 Internal Block Diagram

Internal Block Diagrams are based on the UML Composite Structure Diagrams 401, with restrictions and extensions as defined by SysML. The Internal Block Diagram describes the internal structure of a block and connections between its constituent parts, using ports, connectors, and flows. The typical way to create a new Internal Block Diagram is as follows:

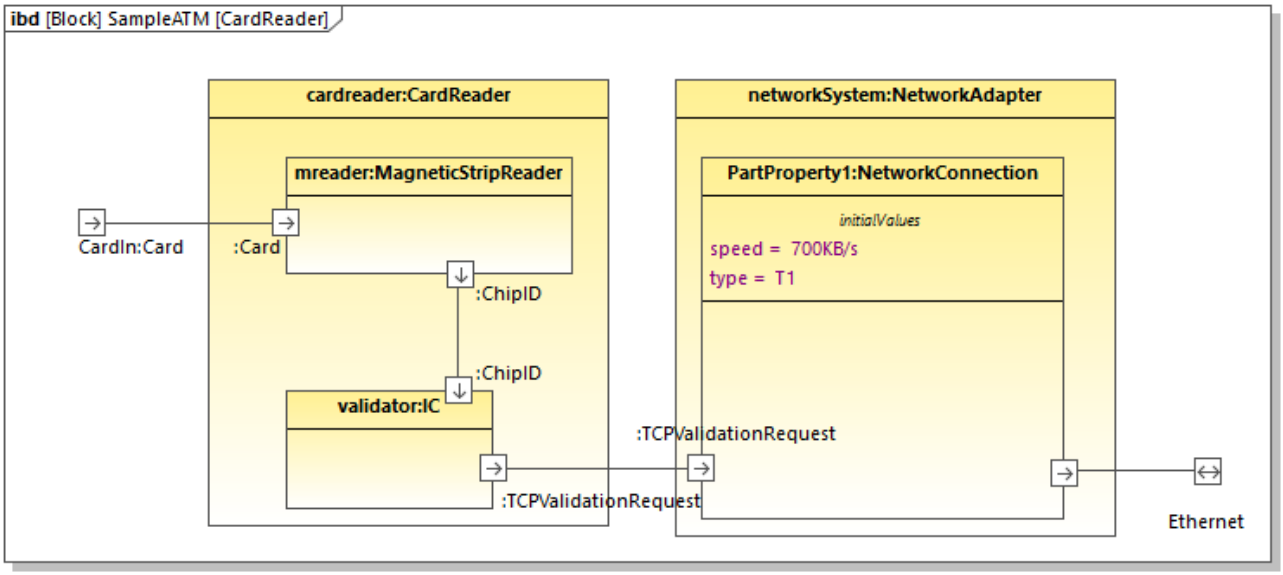
- Right-click an existing block in the Model Tree and select **New Diagram | SysML Internal Block Diagram** from the context menu.

If you create a new Internal Block Diagram without right-clicking an existing block first, a new block is created as well in the Model Tree, and the new diagram is nested under the block because it is assumed to describe it. For example, in the model illustrated below, the "CardReader" diagram describes the "SampleATM" block.

© 2019-2025 Altova GmbH



The "CardReader" diagram is available in the following demo project: **C:\Users\<username>\Documents\Altova\UModel2025\UModelExamples\Bank_SysML.ump**.



CardReader diagram

The **cardreader** and **networksystem** properties illustrated above have the type `CardReader` and `NetworkAdapter`, respectively. These types exist in the same model and are blocks, which determines the appearance of properties in the diagram. Note that you can set or change the type of a property from the **type** drop-down list in the Properties Window[89].

Initial values

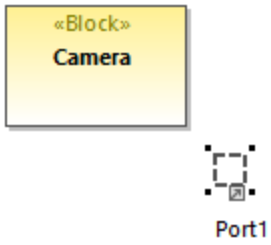
When a property has a type that is a "block" like in this example, it can be created with initial values. For example, the property **PartProperty1** in the **CardReader** diagram has the initial value of `speed = 700KB/s`.

To add initial values to a property:

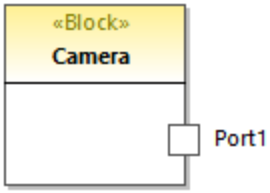
1. Right-click the property and select **New | Initial Values**.
2. Double-click the placeholder and enter the values (for example, `speed = 700KB/s`).

Standard ports

To add a standard port, click the **Port** toolbar button and then click on the diagram. The port is now added to the diagram.



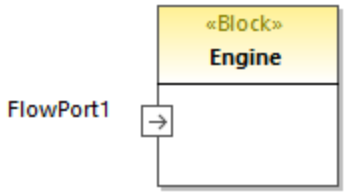
To attach the port to a block, drag it over the border of the block ("Camera", in this example) and drop it when the border becomes highlighted. The port is now attached to the border of the block.



To change the port's name and type, first select the port on the diagram, and then change the **name** and **type** properties of the Properties Window 89.

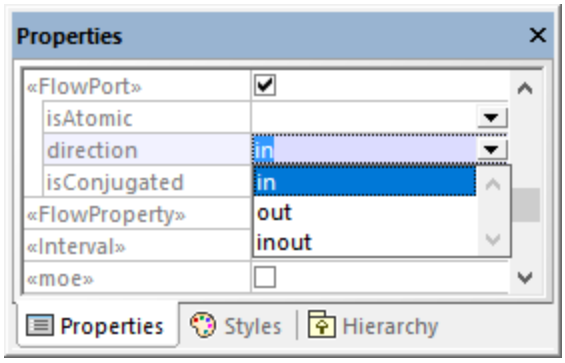
Flow ports

To create a flow port, click the **FlowPort** toolbar button, and then click the border of a block. The flow port is now attached to the border of the block. You can also create and attach flow ports in two separate steps, as shown above for standard ports.



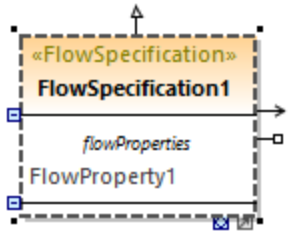
You can change the port's name and type by editing the respective properties in the Properties window. Note that flow ports have additional properties in the Properties window that let you specify the direction, for example (in, out, inout).

© 2019-2025 Altova GmbH



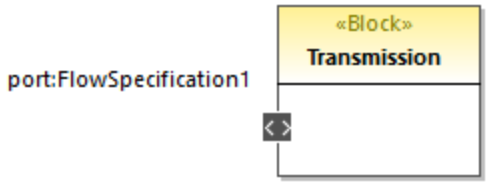
To create an atomic conjugated flow port:

1. Create a FlowSpecification (interface) in a Block Definition Diagram (BDD).



2. Click the flow port in the Internal Block Diagram (IBD).
3. In the Properties window, set the **type** property to the FlowSpecification created earlier.
4. In the Properties window, set the **isConjugated** property to `true`.

An atomic conjugated port is shown with a dark background.



Joining ports

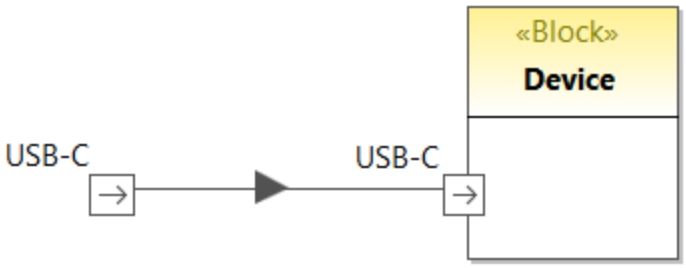
You can join two ports as follows:

1. Click the **Connector** toolbar button.
2. Drag and drop from the first to the second port.
3. Drop the connector on the port, when the port object is highlighted in the diagram.

Item flows

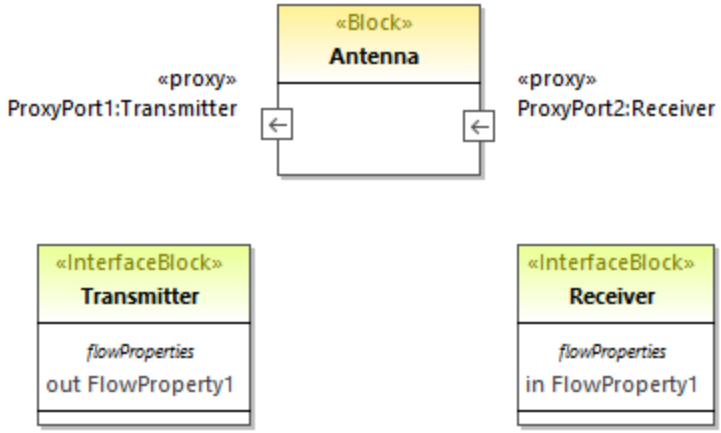
Item flows can be created between block associations, or on other connectors between parts of SysML diagrams.

To create item flows, right-click an existing connector and select **New | Item flow (left to right, or right to left)** from the context menu. An arrowhead is added to the connector, displaying the direction of the item flow.



Proxy ports and direction

In newer versions of SysML, proxy ports can show direction, similar to flow ports of older SysML versions. For example, the diagram illustrated below consists of a block ("Antenna") with two proxy ports that show direction.



Here is an example of how to add direction to proxy ports:

1. Add a block and two interface blocks to the diagram. In the example above, the block is "Antenna" and the two interface blocks are "Transmitter" and "Receiver".
2. Select the "Transmitter" and press **F7** to add a new flow property to it.
3. Add a proxy port to the block and change its type to "Transmitter" from the Properties window.
4. Select the flow property on the diagram and, from the Properties window, change the direction property to **out**. Notice that the direction of the proxy port changes to reflect this fact.
5. Select the "Receiver" and press **F7** to add a new flow property to it.
6. Add a second proxy port to the block and change its type to "Receiver" from the Properties window.

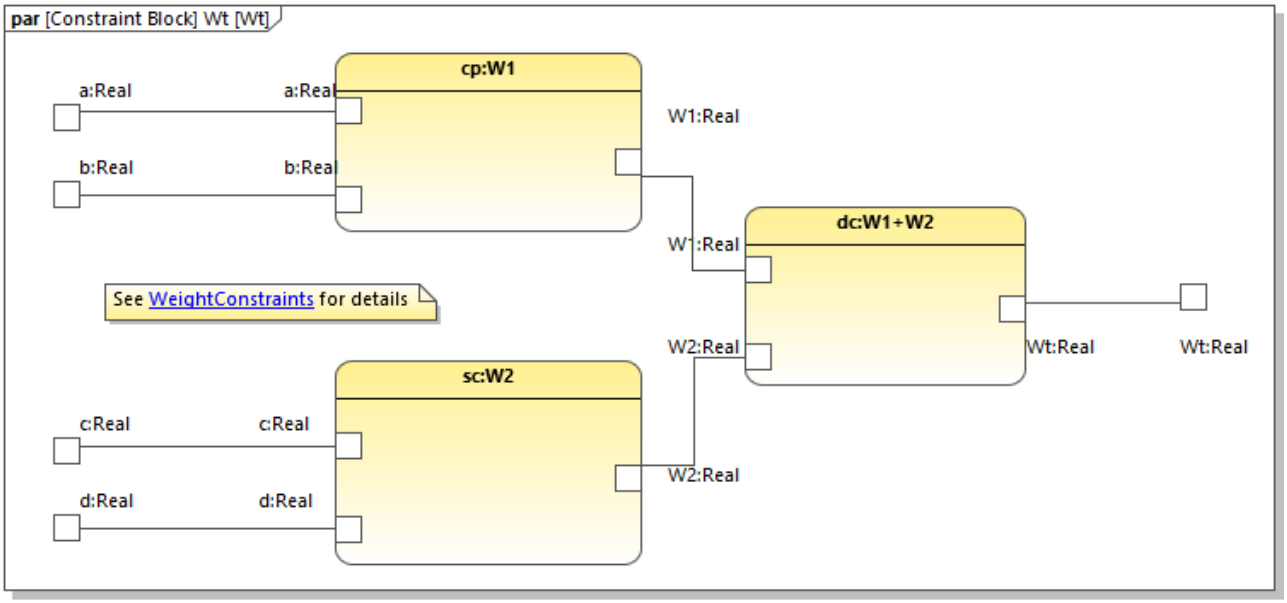
© 2019-2025 Altova GmbH

7. Select the flow property on the diagram and, from the Properties window, change the direction property to **in**. Again, the direction of the proxy port changes to reflect this fact.

8.3.3.3 Parametric Diagram

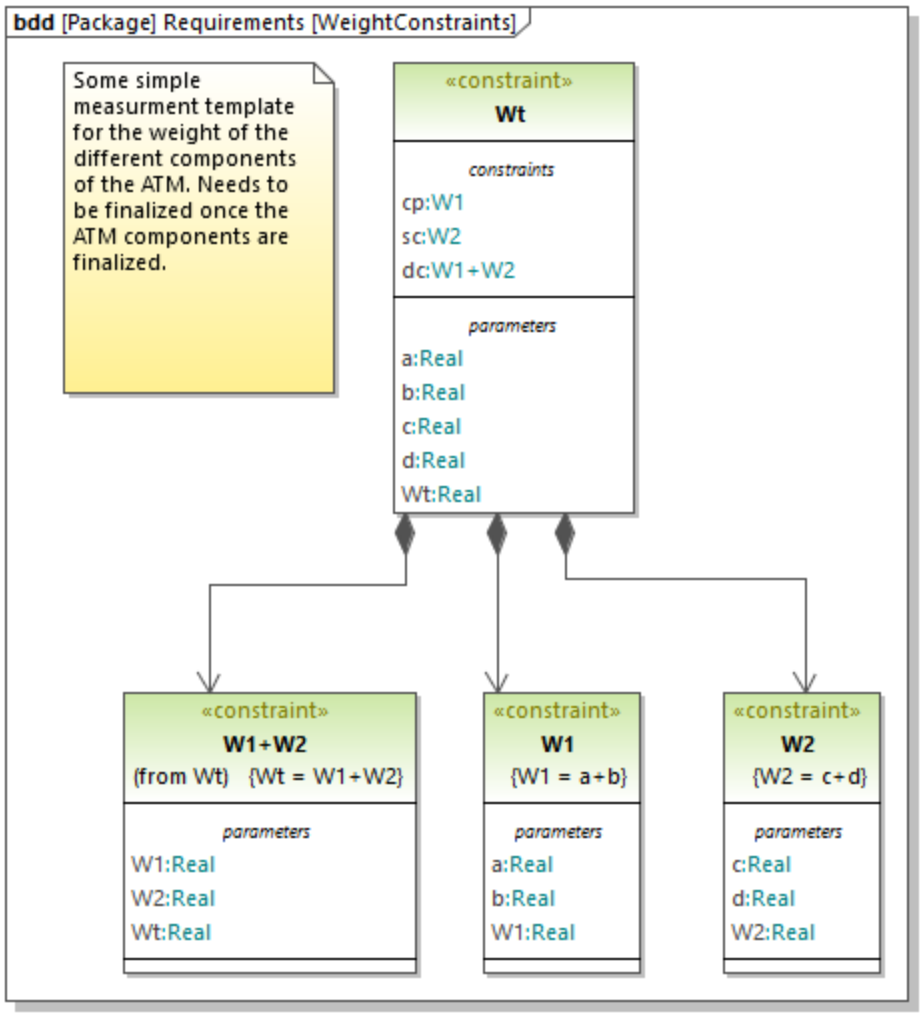
The Parametric diagram is a diagram type specific to SysML that integrates engineering analysis with design modeling. A parametric diagram is similar to an Internal Block Diagram 472, with the exception that only those type of connectors may be shown which are connected to constraint parameters on at least one of their ends.

The Parametric diagram makes use of constraint blocks defined in a Block Definition Diagram to constrain the properties of other blocks in the Parametric diagram. Constraint blocks are shown with rounded corners rather than being square as an ordinary part.



Parametric diagram

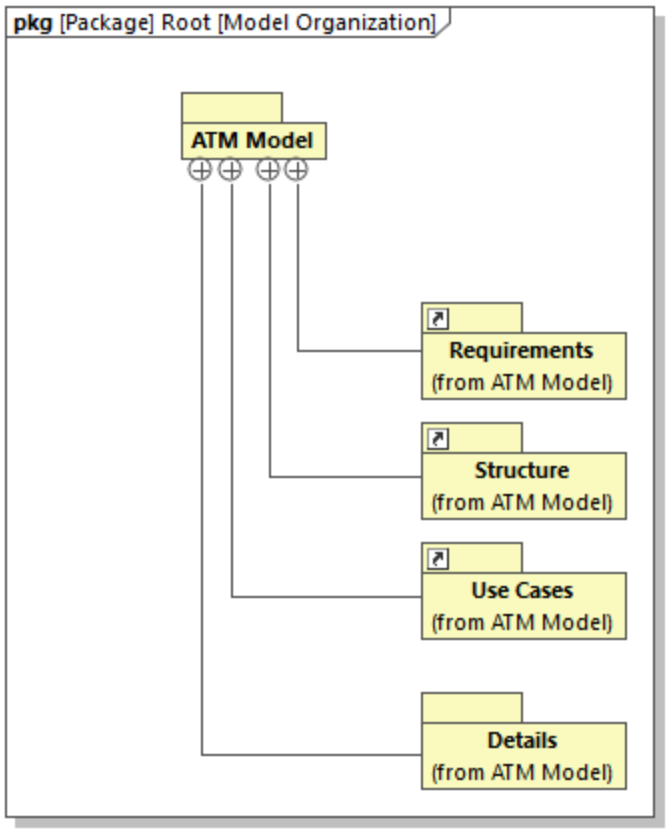
The «constraint» stereotype on a block states that the block is a constraint block. In a Block Definition diagram, parameters of the constraint are shown in a "parameters" compartment.



8.3.3.4 Package Diagram

The Package diagram is used to organize model elements into packageable elements. In such diagrams, you can also define dependencies between packages and model elements within the package. For example, the diagram below illustrates the high-level organization of the model defined in the **Bank_SysML.ump** demo project from the **C:\Users\<username>\Documents\Altova\UModel2025\UModelExamples** directory. The links available for **Requirements**, **Structure**, and **Use Cases** point to the respective packages in the same model, see also Hyperlinking Elements 118.

 © 2019-2025 Altova GmbH

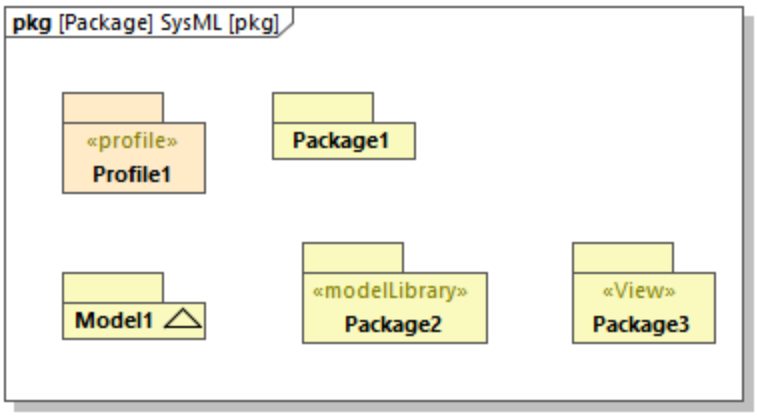


Package diagram

The Package diagram illustrated above is just one of the ways to organize of a model; you can, of course, organize a model by other aspects, for example, by system hierarchy or by diagram type.

In a package diagram, you can add various elements to the diagram in the standard way, by clicking the respective toolbar buttons (such as **Package**, **Profile**, or **View**) and then clicking inside the diagram. Note, however, that some package specializations may not have commands available as toolbar buttons, in which case you can add them as follows:

1. Click the **Package** toolbar button and then click inside the diagram to add the new package.
2. In the Properties window, select the check box with the desired stereotype (for example, «ModelLibrary»).

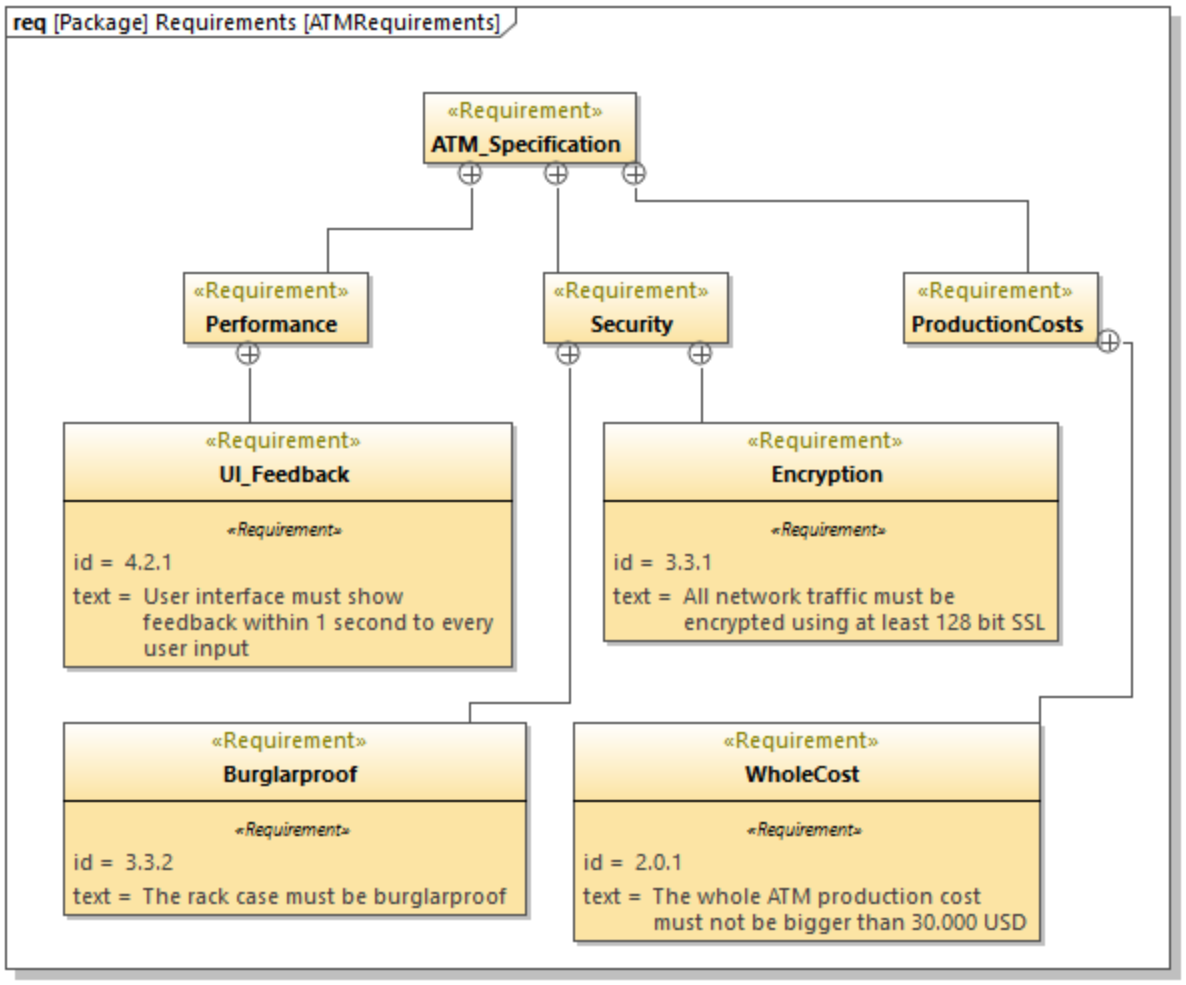


In the package diagram above, **Package2** has the «ModelLibrary» stereotype and **Package3** has the «View» stereotype. See also Applying Stereotypes 148.

8.3.3.5 Requirement Diagram

The Requirement diagram is a diagram type designed specifically for SysML. It integrates the behavior and structure models of SysML with engineering analysis models, such as performance or reliability models. It models text-based requirements and the relationship between requirements and other model elements that satisfy or verify them.

© 2019-2025 Altova GmbH



Requirement diagram

With Requirement diagrams, you may often need to create multiple lines of text, in order to maintain the size of requirement blocks within reasonable limits.

To create multiple lines of text:

1. Double-click the text.
2. While holding the **Ctrl** key pressed, press **Enter**.

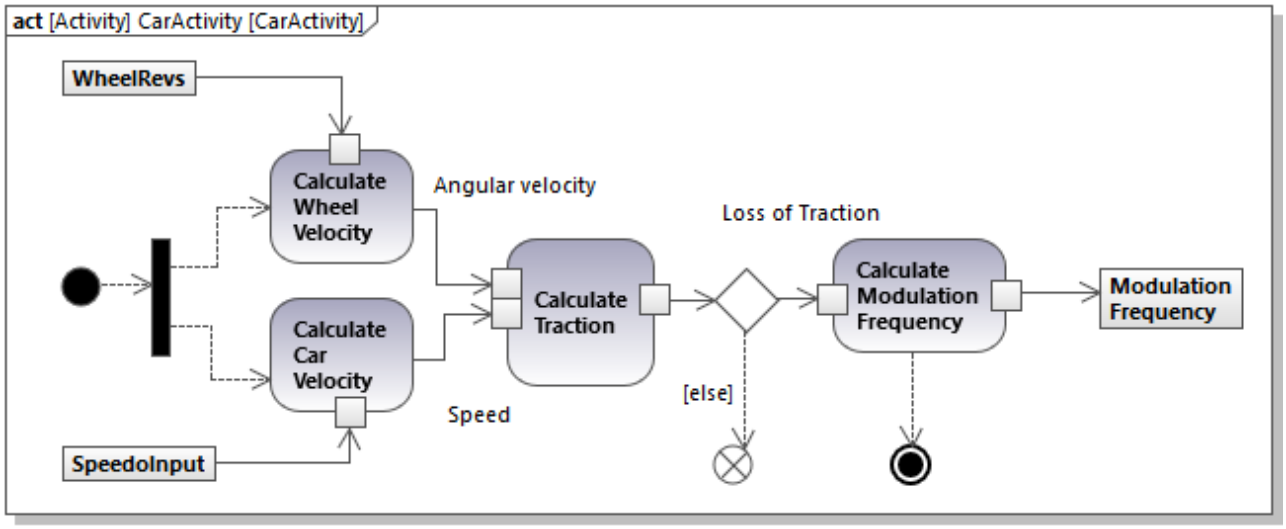
8.3.3.6 Activity Diagram

SysML activity diagrams express information about a system's dynamic behavior, such as the flow of objects during system operation. Such diagrams express the order in which actions are performed, and which of the structures performs a particular action. The flows themselves can be control flows or object flows. You can add either kind of the flow through the respective toolbar buttons:

→ **Control flow**

⤑ **Object flow**

The example Activity diagram illustrated below uses both flows. Control flows appear as dashed lines, while object flows appear as uninterrupted lines.



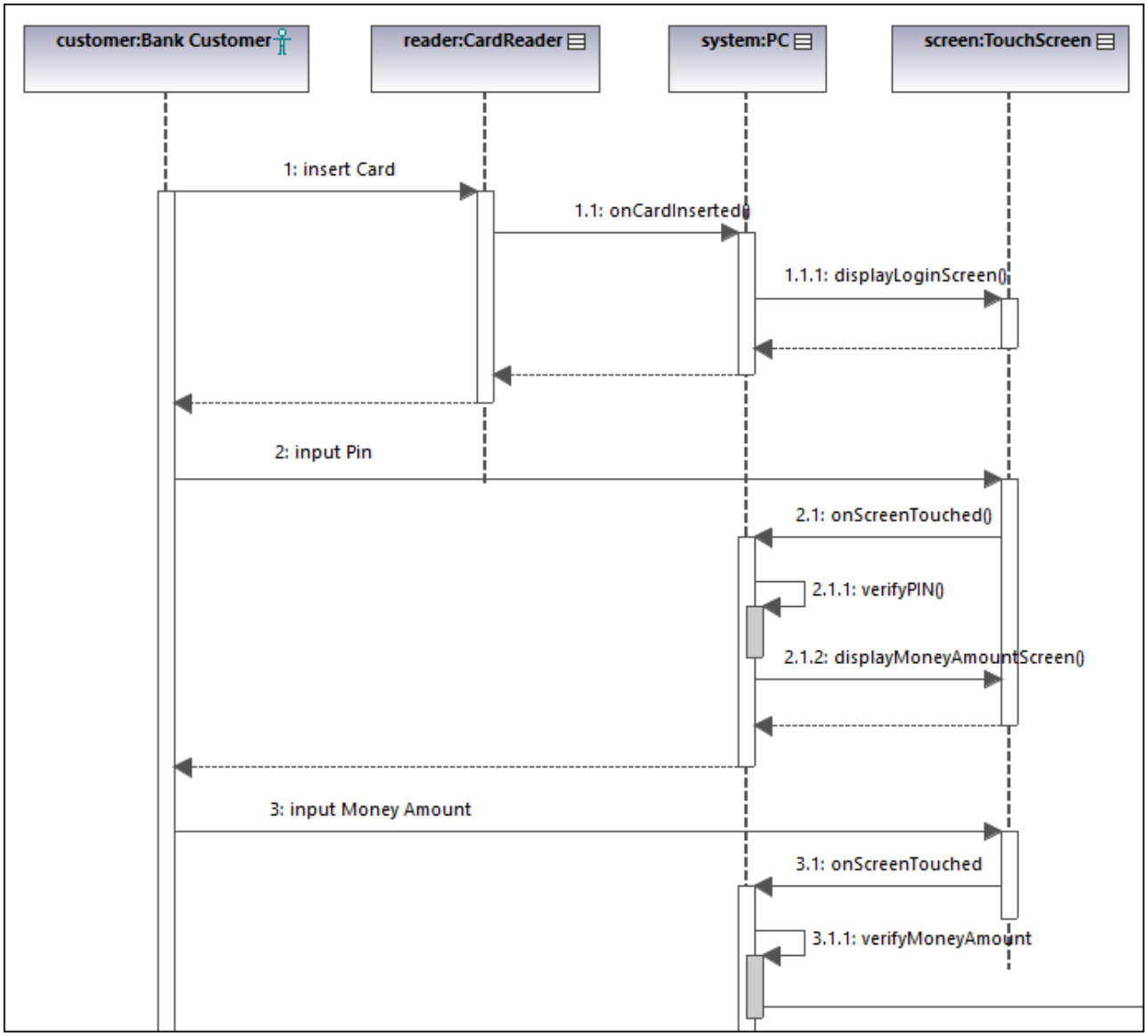
SysML Activity diagram

The SysML Activity diagram is modified from UML, with SysML extensions. For general information about designing UML Activity diagrams with UModel, see Activity Diagram 297.

8.3.3.7 Sequence Diagram

The SysML Sequence diagram also describes a system's dynamic behaviour, like the Activity diagram, but it is more precise. It informs not only about the *order* of actions and which structures perform the actions, but also provides information about the structures which *invoke* a particular action. For this reason, Sequence diagrams tend to become complex unless they focus on a very specific scenario. The image below shows a fragment of a SysML Sequence diagram from the **Bank_SysML.ump** example project.

© 2019-2025 Altova GmbH

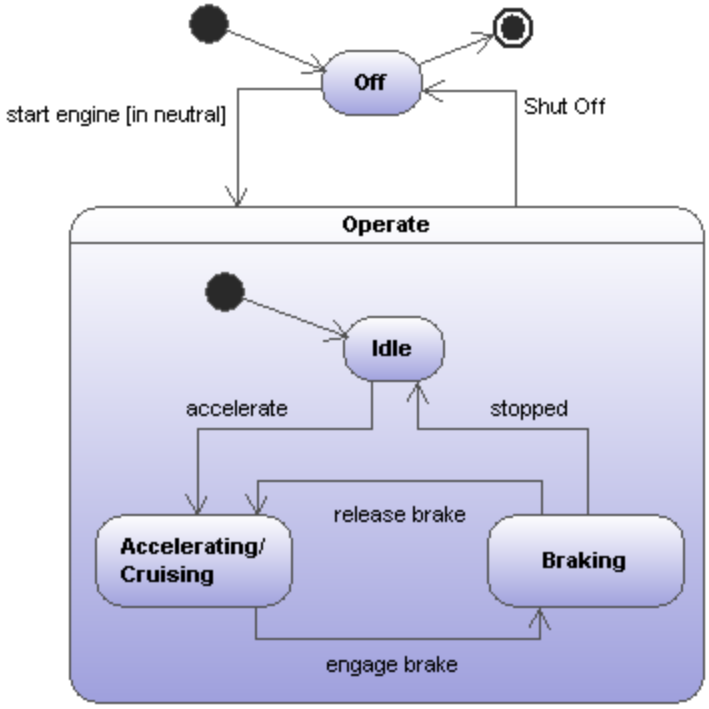


SysML Sequence diagram

The SysML Sequence diagram follows the UML specification. Designing this diagram in UModel requires no specific knowledge compared to the standard UML Sequence diagrams. For general information about the latter, see Sequence Diagram 351.

8.3.3.8 State Machine Diagram

SysML State Machine diagrams express transitions among the states in a running system. SysML State Machine diagrams express system behaviour, just like the Sequence and Activity diagrams of SysML.



SysML State Machine diagram

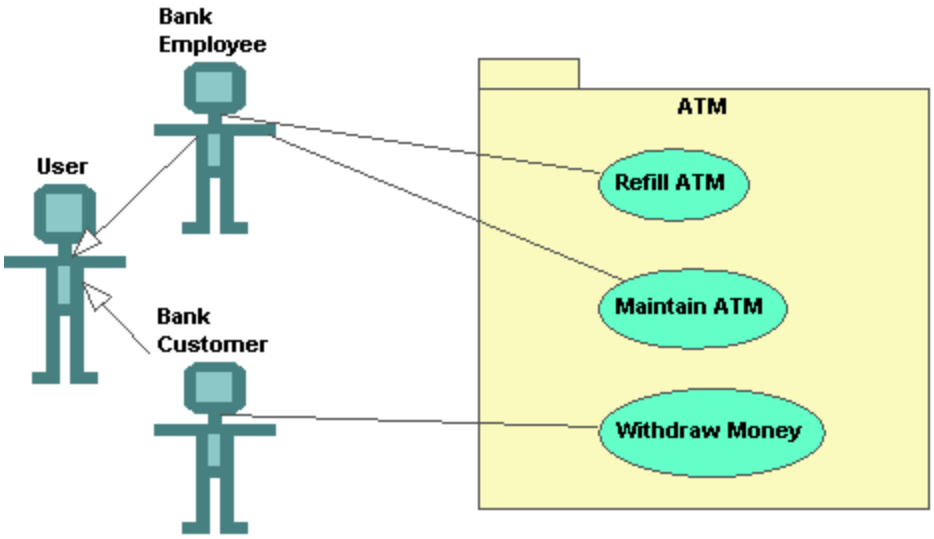
The SysML State Machine diagram follows the UML specification. Designing this diagram in UModel requires no specific knowledge compared to the standard UML State Machine diagrams. For general information about the latter, see State Machine Diagram 314.

8.3.3.9 Use Case Diagram

The SysML Use Case diagram displays elements and relations that describe services provided by a system. It also depicts various stakeholders (such as users or system operators) that consume services. In the **Bank_SysML.ump** example project from the **C:\Users\<username>\Documents\Altova\UModel2025\UModelExamples** directory, the following are examples of services:

- A bank customer interacts with an ATM to withdraw cash
- A bank employee performs ATM maintenance
- A bank employee refills the ATM

© 2019-2025 Altova GmbH

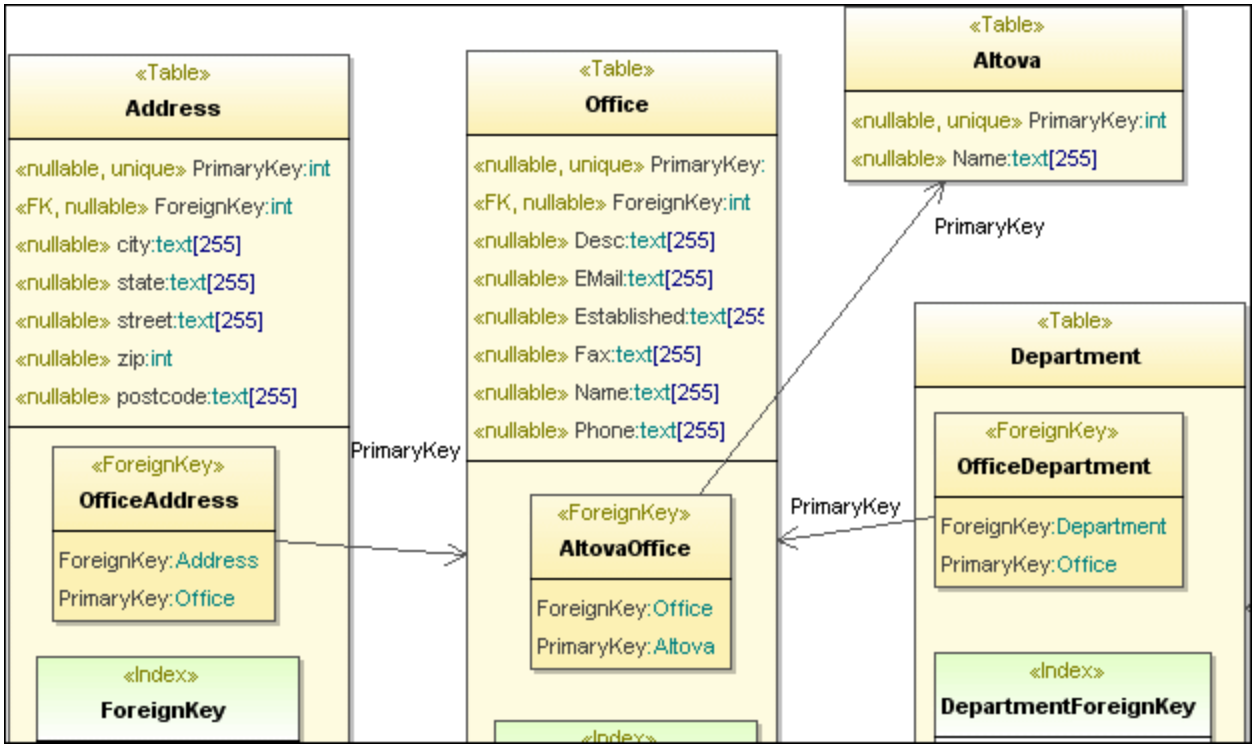


SysML Use Case diagram

The SysML Use Case diagram follows the UML specification, and designing it is not different compared to the standard Use Case diagram of the UML. For more information, see the Use Cases [22] section in the tutorial.

9 UModel and Databases

You can import SQL databases into UModel in order to view their structure or modify it using UML (for a list of supported databases, see Database Support 16). UModel can conveniently display the database structure in UML Database diagrams similar to the one illustrated below.



The following database elements can be imported to the database model:

- Tables
- Check Constraints
- Primary / Foreign / Unique keys
- Indices
- Views
- Triggers
- Stored procedures
- Functions

Note: Views, Triggers, Stored procedures and Functions can only be imported, though not added, in UModel.

After importing the database structure in UModel, you can modify it and apply the changes to the actual database, using the **Merge Program Code from UModel project** command. This creates a database change script file which can be executed, or saved for later execution. Alternatively, if changes took place in the database since the last synchronization, you can merge them into the model (or overwrite the model with the changes).

For information about how database elements map to UModel elements, see Database Mappings 273.

© 2019-2025 Altova GmbH

9.1 Modeling Databases in UModel

You can model databases in UModel in one of the following ways:

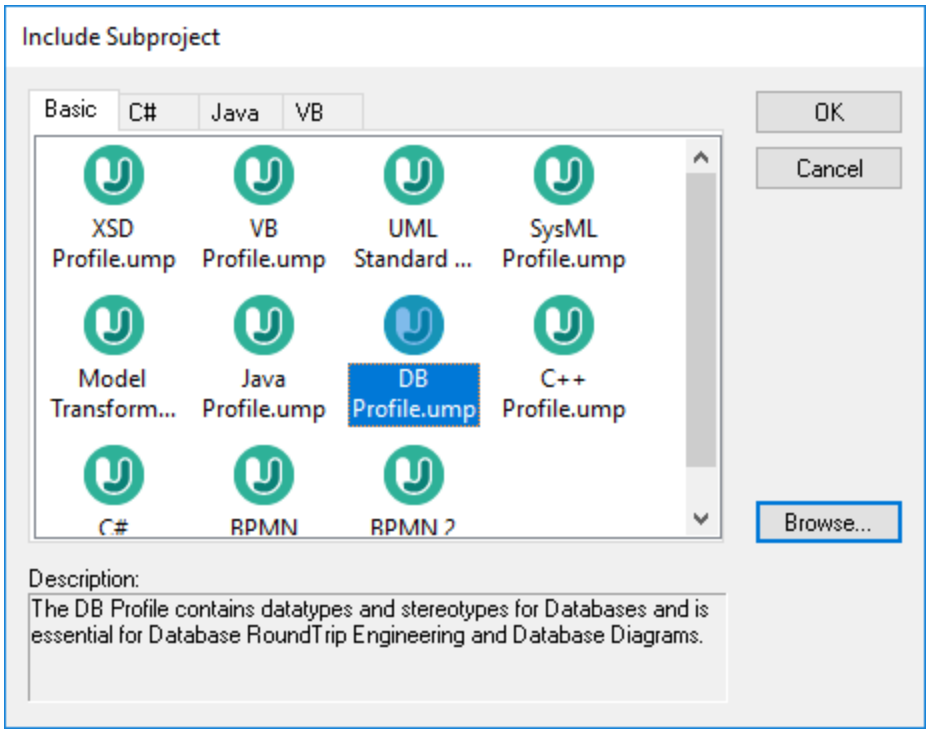
1. Without code engineering support. In this scenario, you model the database objects without connecting to a real database (for example, you just want to create a diagram that illustrates the potential structure of a database).
2. With code engineering support. In this scenario, you connect to a database, import its structure into the model, and then view the database object definitions directly in UModel. Upon reading the database structure, UModel can automatically generate database diagrams. Optionally, you can modify the database objects in the model (for example, add a new table, or delete an existing one) and then update the real database by means of scripts generated by UModel. Synchronization between your database and the UModel project works in both directions, similar to how it works for programming languages. You also have the option to synchronize only the changes (do a merge), or overwrite all existing data (either the database from the model, or the model from the database).

In either of the cases above, your project must contain the Database profile available with UModel. This profile provides all the required metadata (such as UML stereotypes) that enable you to view or design database objects in UModel.

If you are using the code engineering approach, the Database profile and all the required code engineering configuration will be added automatically to your project the first time when you import a database into the model. Otherwise, you will need to include the Database profile manually.

To add the database profile to a UModel project manually:

1. Create a new UModel project or open an existing one, see Creating, Opening, and Saving Projects 154.
2. On the **Project** menu, click **Include Subproject**.



3. On the **Basic** tab, select **DB Profile.ump**, and then click **OK** to confirm.

Alternatively, do the following:

1. In the Diagram Tree Window 87, right-click **Diagrams**, and select **New diagram | Database Diagram**.
2. When prompted, select a package where the new diagram should belong.
3. When prompted by UModel that the Database profile will be added to your project, click **OK** to confirm.

Now that the UModel DB profile has been added, you can start modeling your database objects. For example, when you right-click inside a database diagram, the context menu provides options to create a new table. Likewise, when you right-click a table, the context menu provides options to create a column, keys, indices, and so on. For further information, see Designing Database Objects 495.

To establish a connection to a database and use the code engineering approach, see Importing SQL Databases into UModel 488.

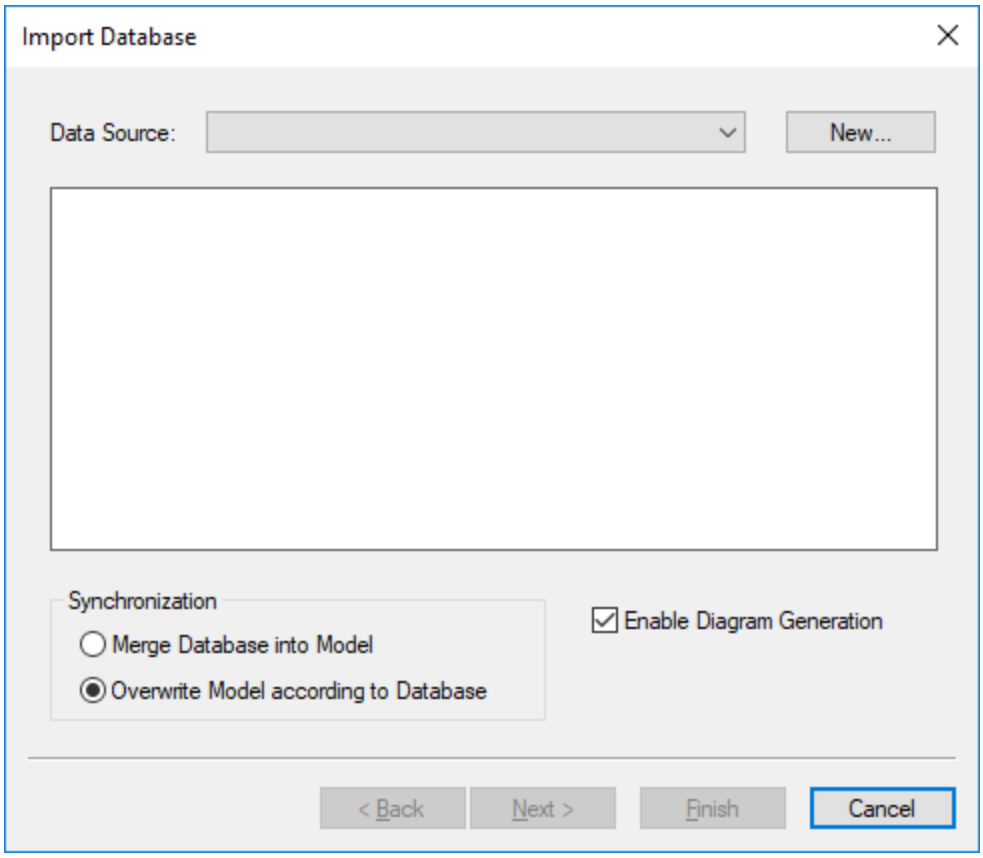
9.1.1 Importing SQL Databases into UModel

The instructions below show you how to import the structure of a database into UModel. You will also learn how to generate a UML diagram that illustrates the database structure. The database used in this tutorial is a sample Microsoft Access database; however, the steps are very similar for other database types supported by UModel.

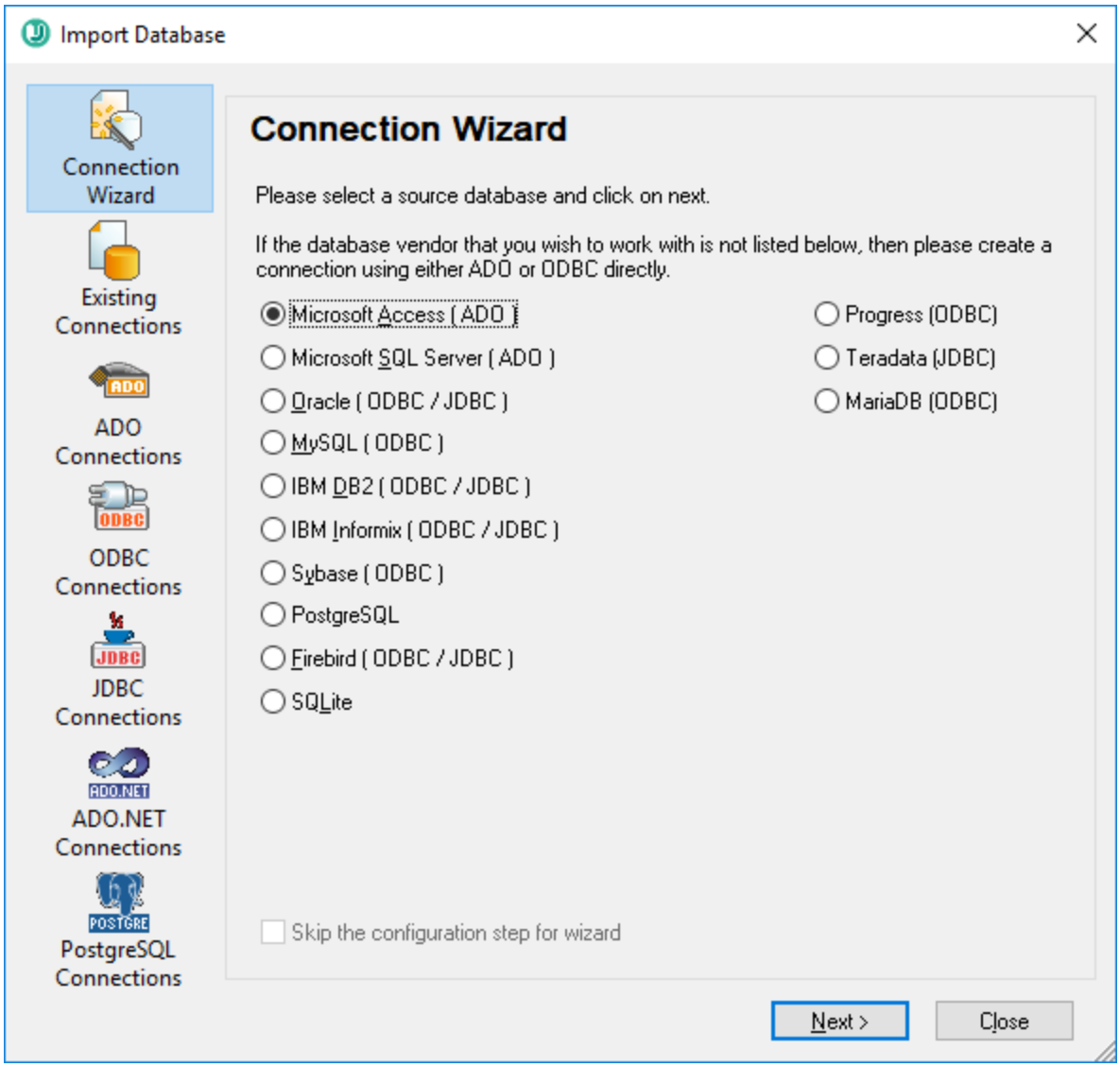
© 2019-2025 Altova GmbH

To import a database into UModel:

1. On the **Project** menu, click Import **SQL Database**.
2. If this is the first time you are importing a database into UModel, click **New**. Otherwise, you can select an existing database connection from the **Data Source** list.

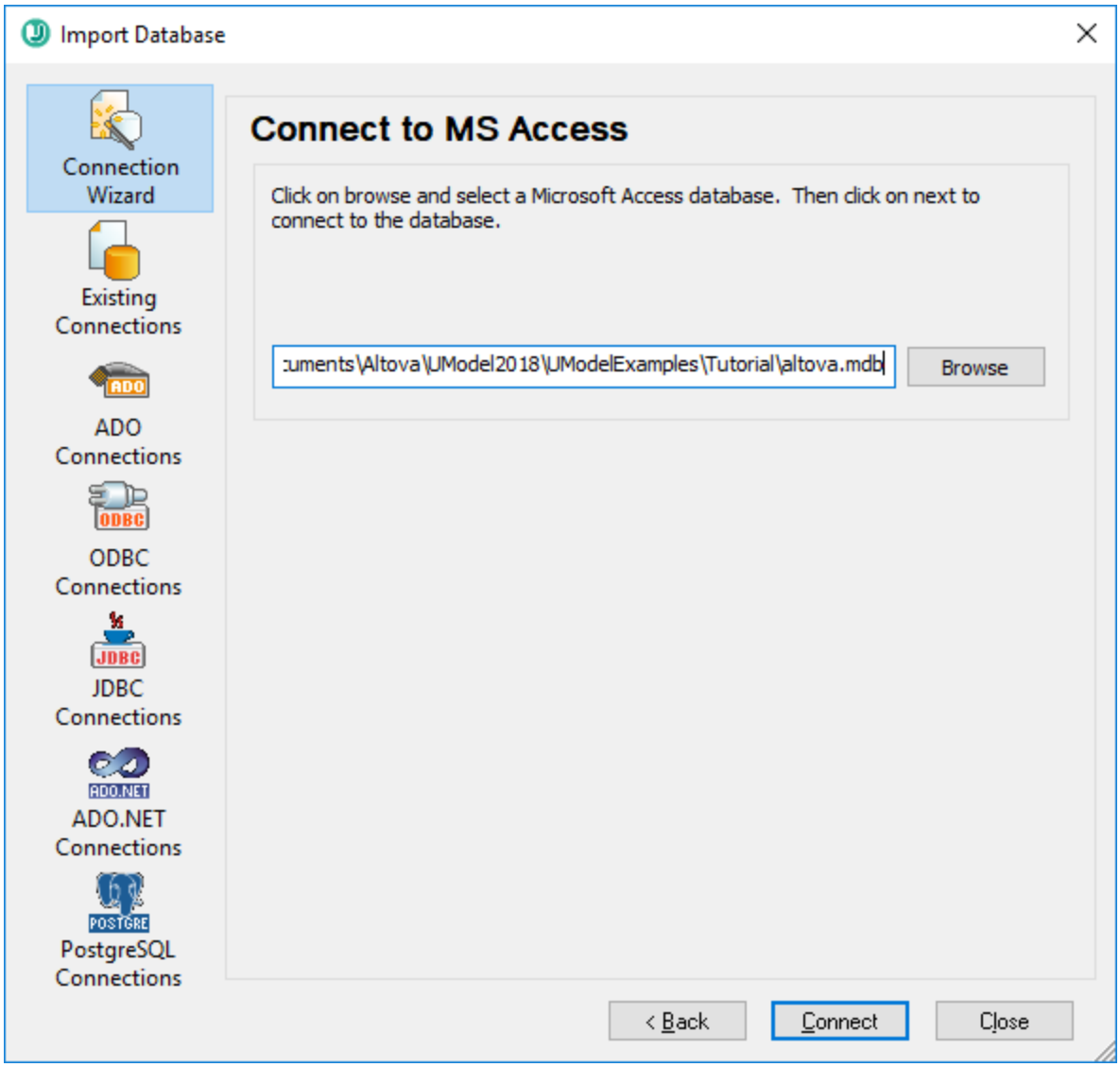


3. In this example, we are connecting to a local Microsoft Access database. Therefore, select **Microsoft Access (ADO)** as database kind, and then click **Next**. Otherwise, follow the wizard steps to connect to your preferred database. Depending on the database kind, you may need to install a database driver before you can connect. For specific examples, see Database Connection Examples 533.

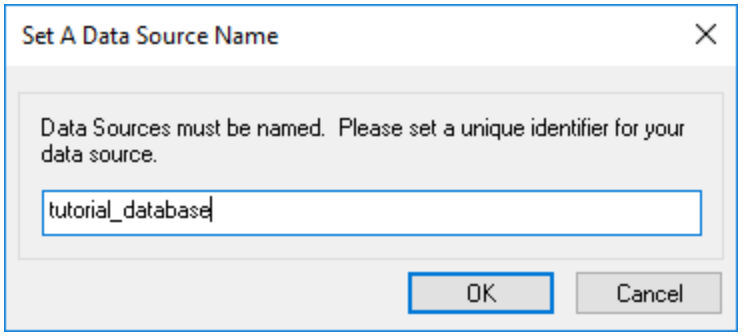


4. Browse for the following database file: **C:\Users\<username>\Documents\Altova\UModel2025\UModelExamples\Tutorial\altova.mdb**, and then click **Connect**.

© 2019-2025 Altova GmbH

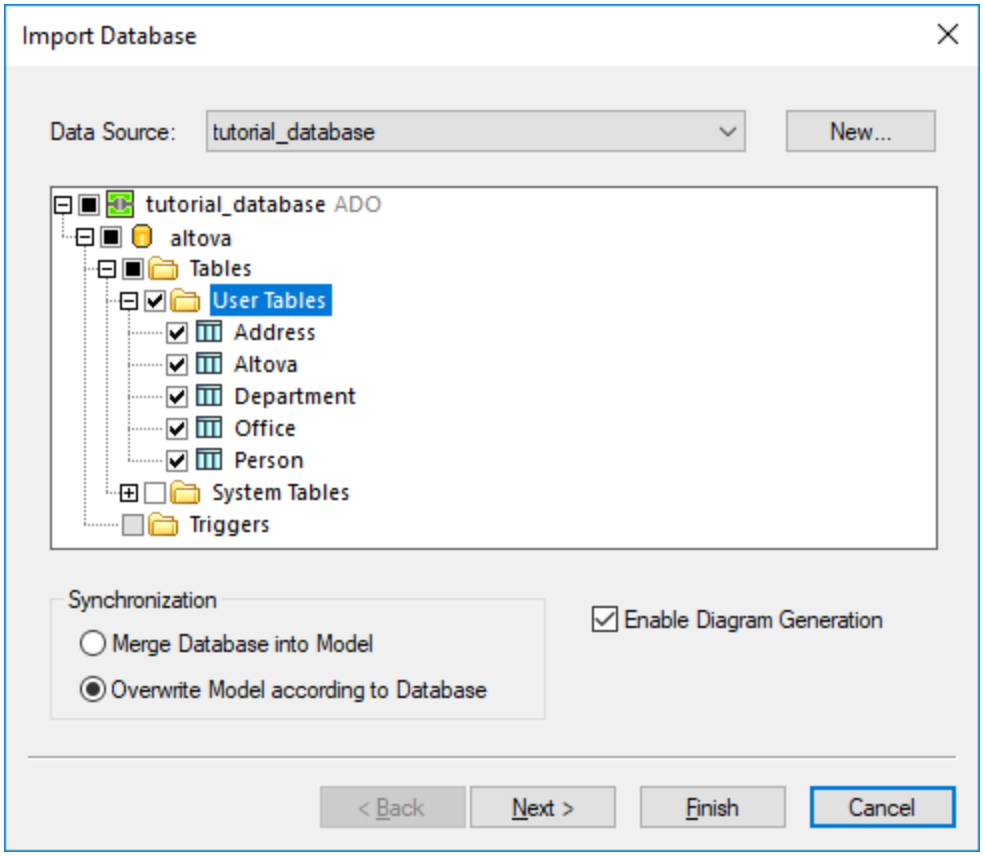


5. Enter a descriptive name for your data source. The data source name set here will later be available for selection when you want to connect to the same database again.



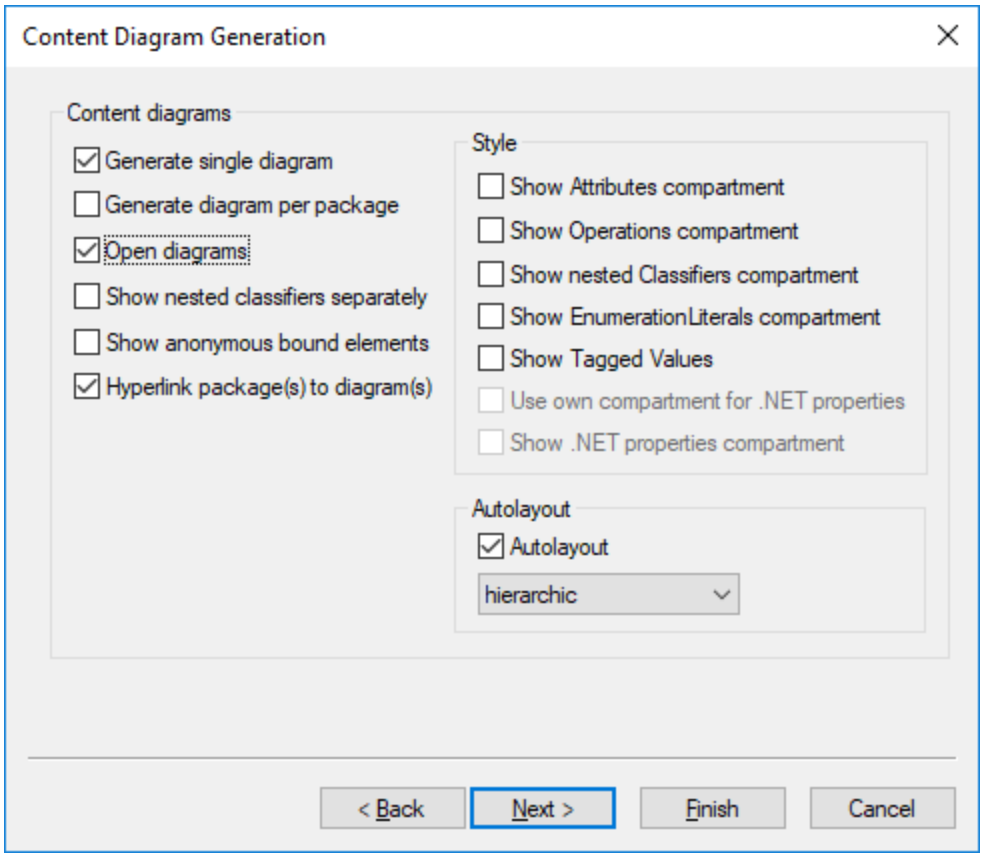
6. Select the database objects that you would like to import into the model. In this example, all user tables are imported. Also, notice the option **Overwrite Model according to Database** is selected

(which means all elements in the project will be replaced with those imported from the database). For existing projects, change this option to **Merge Database into Model**.



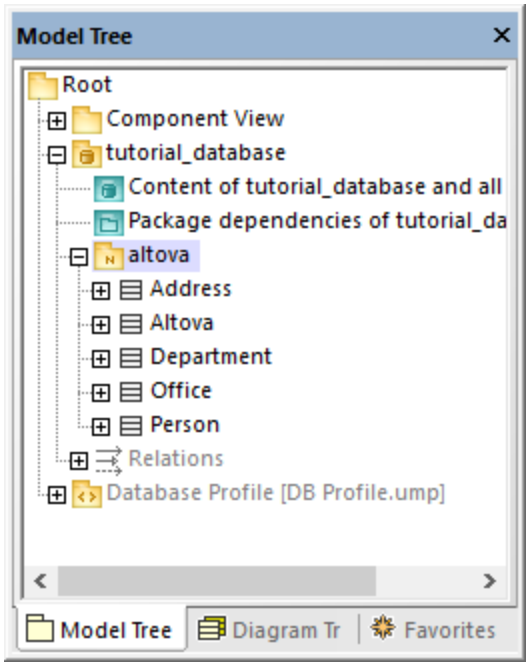
7. Click **Next**. Select the diagram generation options as shown below:

© 2019-2025 Altova GmbH

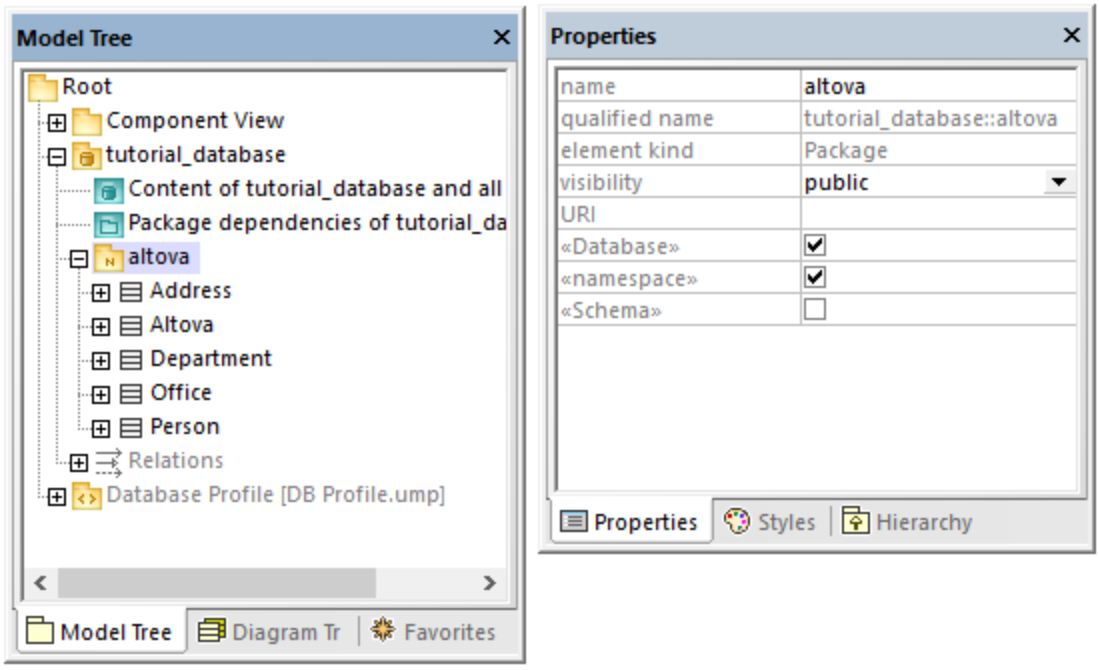


8. Click **Finish**.

After the import, the project contains all the objects imported from the database (tables and their structure). Two diagrams are also created, a database diagram that illustrates the database objects, and a package dependency diagram.



A illustrated above, the data source ("tutorial_database" in this example) has become a package in the model. The database itself ("altova") has also become a package that has both the «Database» stereotype and the «namespace» stereotype. To view the properties of a package, click the package and then look at the Properties window, for example:



Note: After a database import, UModel creates packages and applies stereotypes depending on the database kind. The model above is illustrative of Access databases.

© 2019-2025 Altova GmbH

All database tables become classes in the model, and get the «Table» stereotype. Notice also that, after the import, the Database profile (**DB Profile.ump**) has been automatically added to the project.

At this stage, the project is configured for code engineering from database to model. That is, whenever you want to update the UModel project with the latest database changes, run the following command:

- On the **Project** menu, click either **Merge UModel Project from Program Code** or **Overwrite UModel Project from Program Code**.

If you intend to synchronize from the model to the database, see Configuring Roundtrip Engineering for Databases 500.

9.1.2 Designing Database Objects

In UModel, you can create, edit, or delete database objects (such as tables, columns, foreign keys, and so on) either from a Database diagram, or from the Model Tree window.

When viewing or designing database objects In UModel, keep in mind the following basic rules:

- Tables are classes with the «Table» stereotype.
- Columns are class properties.
- Primary, foreign, and unique keys are classes with the «PrimaryKey», «ForeignKey», «UniqueKey» stereotypes, respectively.
- Check constraints are classes with the «CheckConstraint» stereotype.
- Indices are classes with the «Index» stereotype.

For an exhaustive table that illustrates how each database object maps to a UModel element, see Database Mappings 273 .

Adding tables

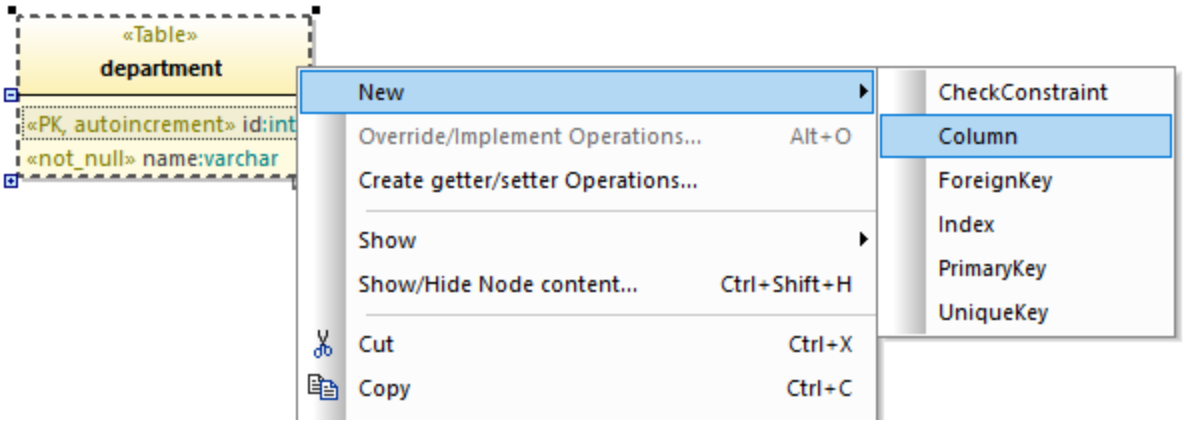
To add a table to the model, do one of the following:

1. Create a database diagram or open an existing one. To create a new Database diagram, right-click a package in the Model Tree 83 window, and select **New diagram | Database diagram** from the context menu.
2. Do one of the following:
   a. Right-click inside the diagram and select **New | Table** from the context menu.
   b. Click the **New Table** toolbar button, and then click inside the diagram to add the table.

Note: You can add a table class anywhere in the model. However, as best practice and especially if you intend to use code engineering, all table classes must belong under a package that has the «Database» stereotype. Such a package is created automatically whenever you import an existing database into the model, see Importing SQL Databases into UModel 488.

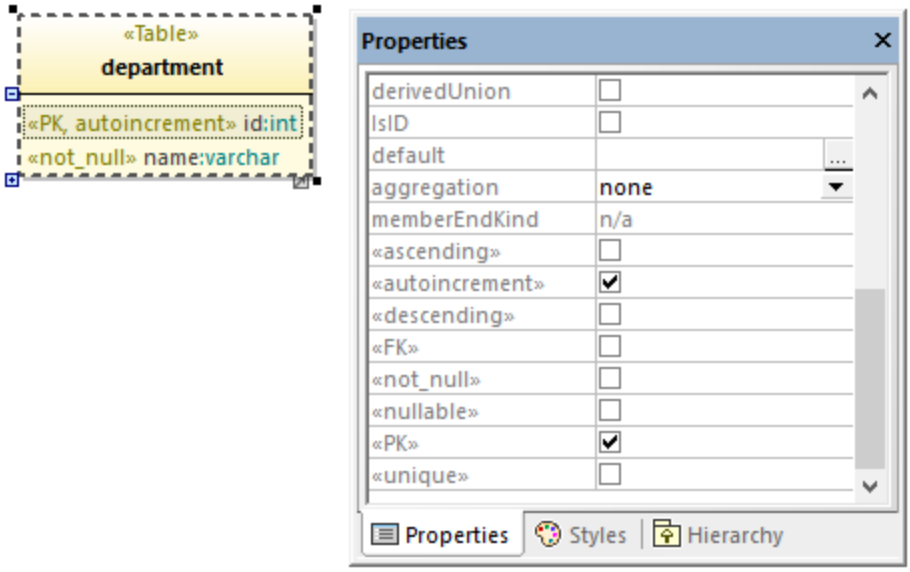
Adding other database objects

To add a column, index, foreign key, etc to a table, right-click the table on the diagram, and then select the respective command from the context menu, for example:



Alternatively, click a toolbar button in the diagram's toolbar, and then click inside the target table.

To set column attributes such as "autoincrement", "nullable", "primary key", first click the column, and then select the required checkbox (stereotype) in the Properties window:



You can also create the column and set all required attributes directly as you type. For example, to create a primary, autoincrement column with the name "id" and type "int", do the following:

1. Select a table on the diagram and press F7.
2. Start typing `<<PK, autoincrement>> id:int`. As you type, UModel assists you to pick up the required values automatically from a list.

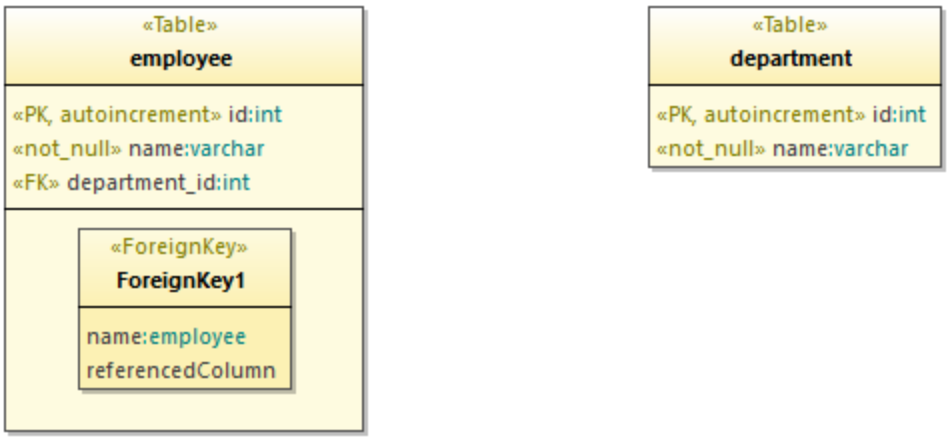
© 2019-2025 Altova GmbH

Adding database relationships

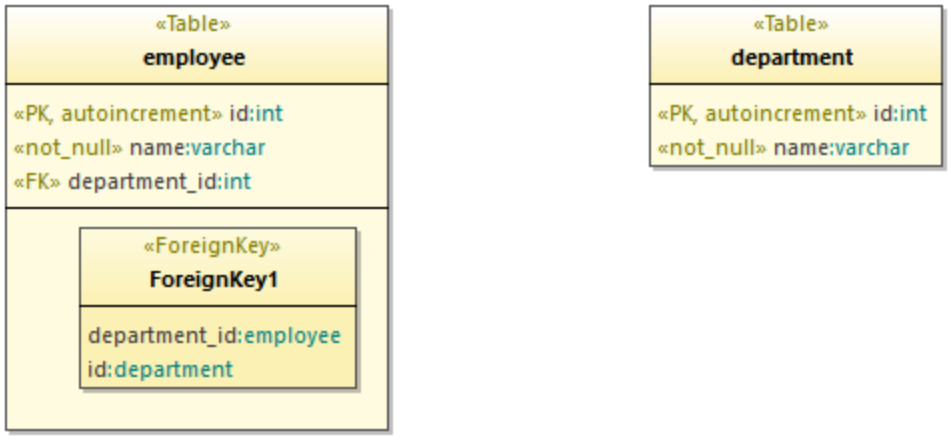
You typically add relationships to illustrate foreign key dependencies between columns of different tables. For example, let's assume that you have the following classes:

To add a foreign key relationship between the **department_id** column in the "employee" table and the **id** column in the "department" table, do the following:

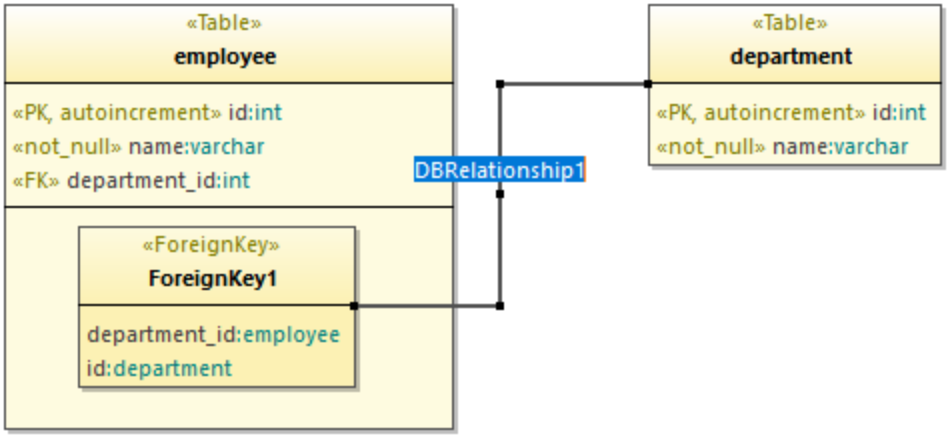
1. Right-click the "employee" table and select **New | ForeignKey** from the context menu. A new class called "ForeignKey1" is added inside the "employee" class.



2. In the "ForeignKey1" class, change the first column entry to correspond to the owner column and table (in this example, department_id:employee). Then change the second column entry to correspond to the referenced column and table (in this example, id:department).

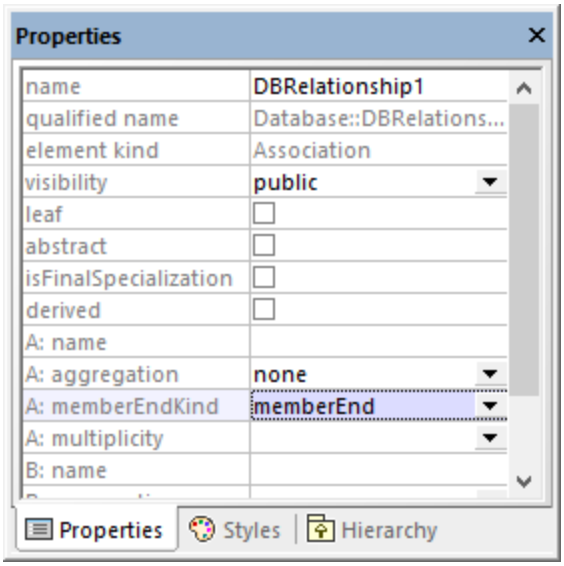


3. Click the **Database Relationship Association** toolbar button, and then drag from the "ForeignKey1" class onto the "department" class.



4. Select the relationship line, and, in the Properties window, change the **A :memberEndKind** property to **memberEnd**.

© 2019-2025 Altova GmbH



5. Press **F11** to check the project syntax for any errors (see below for more information).

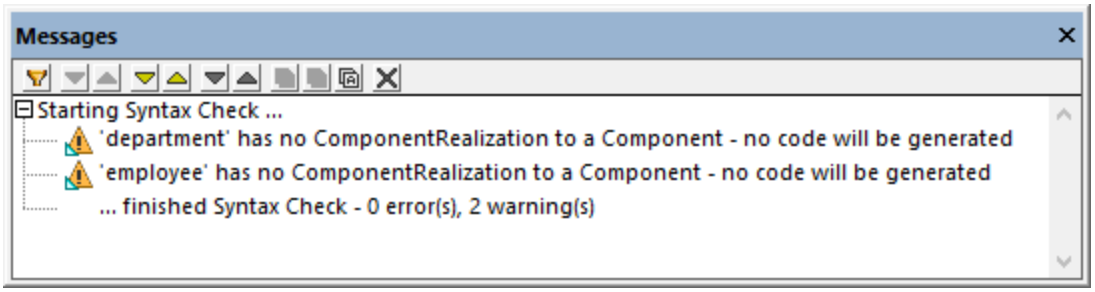
Note: If necessary, you can add multiple column entries per "ForeignKeys" class. You can also add multiple indices for the same table.

Checking project syntax

As you create or change database objects in UModel, it is good practice to periodically check the syntax of your project for any potential design issues (for example, tables that do not have at least one column, missing foreign key references, and so on). To check the project syntax, do one of the following:

- On the **Project** menu, click **Check Project Syntax**.
- Press **F11**.

UModel validates the project and displays any encountered problems in the Messages window, for example:



The two warnings in the image above indicate that no code will be generated for the "department" and "employee" tables. You can ignore such warnings if you do not need code engineering support in your UModel project. Otherwise, see Configuring Round-Trip Engineering for Databases 500.

9.1.3 Configuring Round-Trip Engineering for Databases

Whenever you import a database into UModel as shown in Importing SQL Databases into UModel 488, your project becomes bound with the database, and you can synchronize elements either from the database into the model, or vice versa.

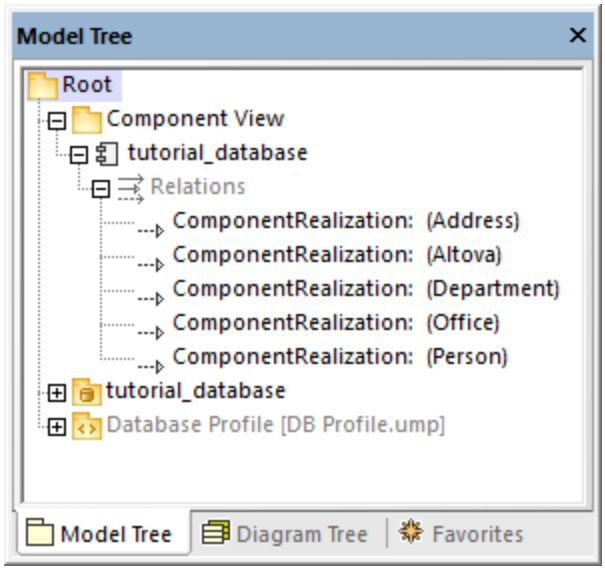
If you want to synchronize only from the database into the model, there is no need for any extra configuration—UModel takes care of all required mappings behind the curtains. For example, after each synchronization, new database tables will become new classes in the model, changed database column definitions will be updated in the model, and so on. All your database diagrams will also be updated automatically to reflect this.

However, if you make changes to the model and want to synchronize them back into the database, some additional configuration might be necessary in the UModel project. This configuration may also be necessary if you want to prevent the project (or certain tables) from synchronizing with the database.

> A synchronization can either merge or overwrite changes—you can always configure this by running the menu command **Project | Synchronization Settings**.

Note: Some database kinds do not allow changing the database structure by virtue of their design. For example, renaming tables and columns is not supported by Microsoft Access databases. Likewise, renaming columns is not supported in SQLite. Therefore, such changes in the model will not trigger a database update, and UModel may display warnings in the Messages window.

Round-trip engineering for databases is very similar to round-trip engineering for program code—it revolves around a component in the "Component View" package that binds your project to the real database. Specifically, whenever you import the database for the first time, a code engineering component is generated automatically under the "Component View" package. For example, if you followed all the steps in Importing SQL Databases into UModel 488, then a component called **tutorial_database** was generated:



As stated before, each class in the model corresponds to a database table. For code engineering to be possible, the code engineering component must realize all the classes (tables) from the model—notice all the

 © 2019-2025 Altova GmbH

ComponentRealization relationships in the image above. Classes that are not realized by this component will not be part of code engineering. If you do not intend to ever update the database from the model, you do not need to take any action—UModel will create all realizations automatically whenever you synchronize from database to model.

However, if you intend to synchronize from the model to the database, each new class (table) that you add must have a **ComponentRealization** relationship to the code engineering component. Otherwise, when you attempt to update the database from the model, UModel displays a warning similar to the following: `Table1 has no ComponentRealization to a component - no code will be generated`.

The easiest way to create a **ComponentRealization** from a class to a component is to drag the class and drop it onto the code engineering component. So, for example, if you created a new class (table), drag the class (in the Model Tree window) onto the **tutorial_database** component to create the relationship. You can also add or remove such relationships from a Component diagram (see Component Diagrams 53).

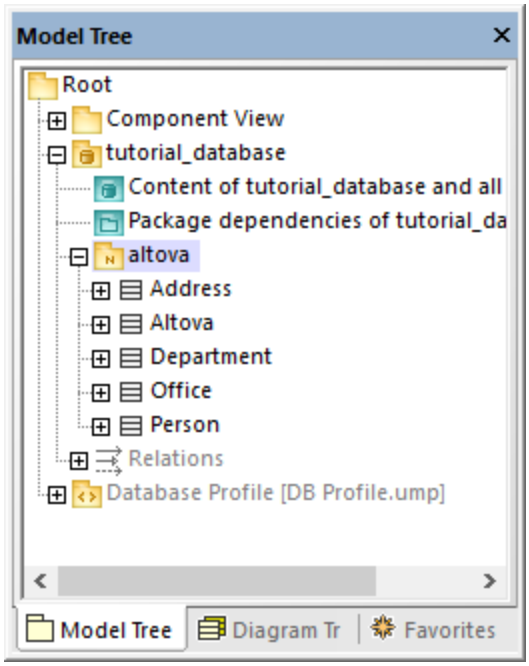
For a worked example, see Example: Update a Database from the Model 501.

9.1.4 Example: Update a Database from the Model

This example shows you how to update the structure of a database by means of scripts generated by UModel. The database used in this example is a local Access database available at the following path: **C:\Users\<username>\Documents\Altova\UModel2025\UModelExamples\Tutorial\altova.mdb**. In this example, we will add a new table to the database in UModel, and then generate a SQL script that updates the structure of the underlying Access database.

To proceed with this example, first import the database into the model, as shown in Importing SQL Databases into UModel 488. As illustrated below, after import, your project will include the following:

- A code engineering component responsible for code generation in both directions (from model to database, and vice versa). To view the code engineering component, expand the "Component View".
- A package that represents the structure of the imported database (for example, each database table is a class).
- The Database Profile required to work with database modeling projects.



Add a table

Let's now add a new table to the database in the model.

1. Double-click the "Content of tutorial_database..." diagram.
2. Right-click inside the diagram and select **New | Table** from the context menu.
3. Enter a table name, for example, "Products".

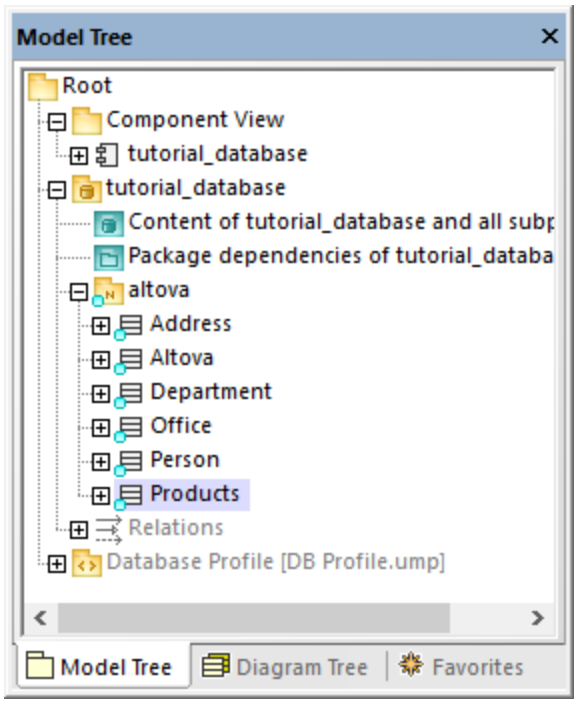
4. Click the table and press **F7** to add a new property (this will become a table column in the database).
5. Type `<<PK, autoincrement>> id:int` inside the property body.

6. Using the same steps as above, add a new column "title" of type "text".

© 2019-2025 Altova GmbH

Prepare the model for forward engineering

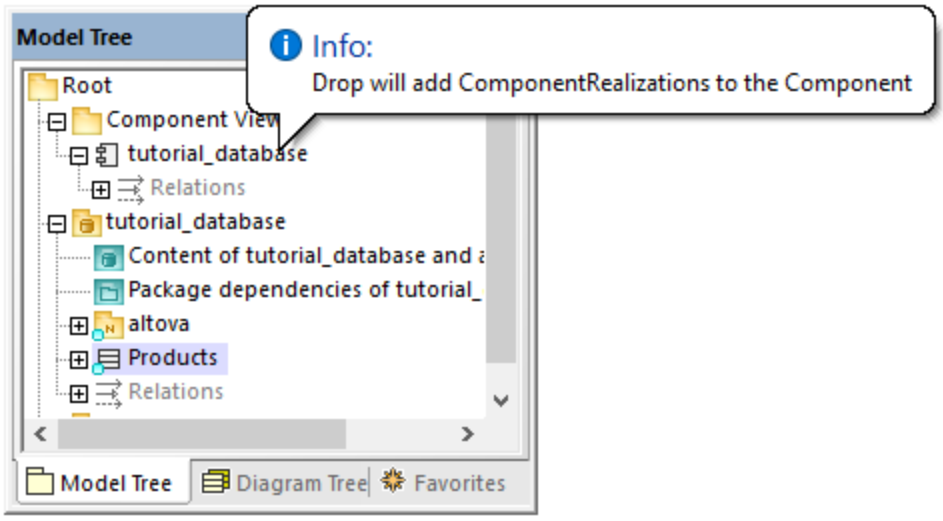
Before a table can be forward engineered from model to the database, it must belong to the correct namespace. To do this, in the Model Tree window, make sure that the class "Products" is under the "tutorial_database" namespace. If it is not, simply drag and drop it onto the "tutorial_database" namespace. Your model should now look as follows:



As explained in Configuring Round-Trip Engineering for Databases [500], it is good practice to validate the project syntax before attempting to update the database. If you press **F11** to check the project syntax at this time, a warning appears in the Messages window that table "Products" has no realization to a component.

You can quickly create a realization to a component as follows:

- In the Model Tree window, drag the class "Products" onto the "tutorial_database" component.

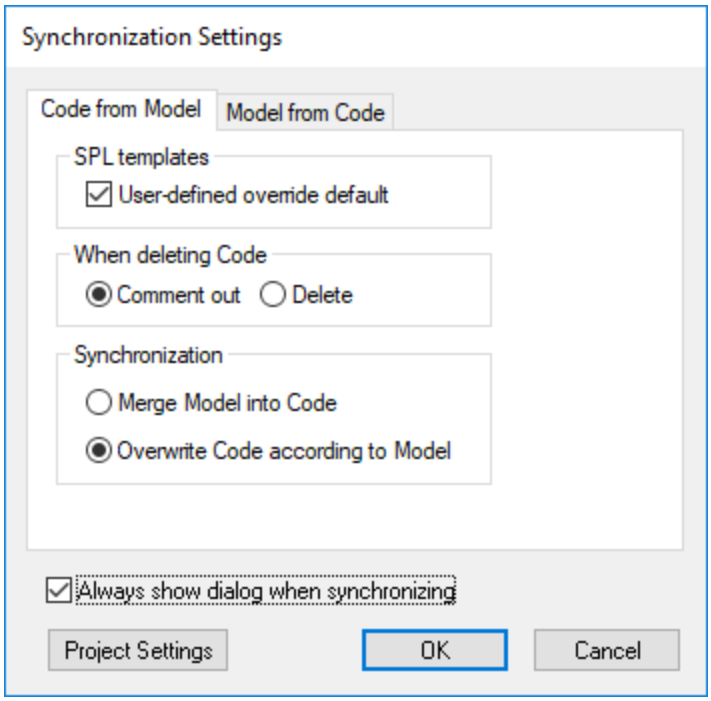


Generate the SQL script

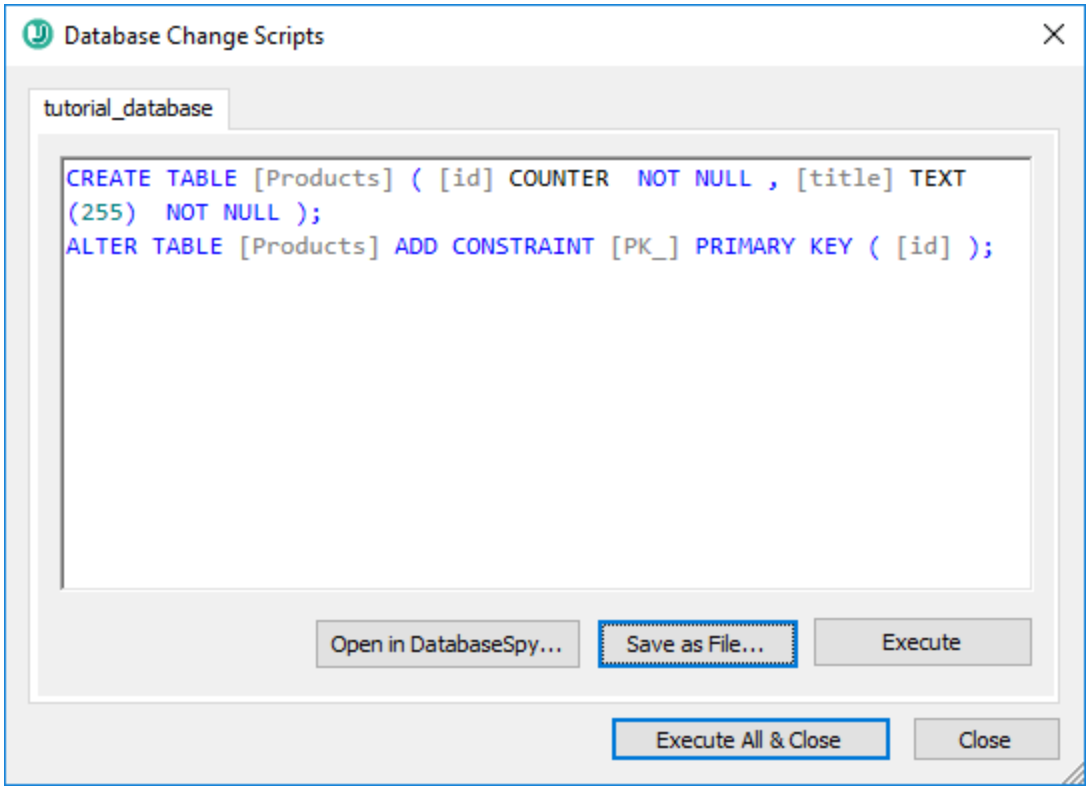
If the project has no more errors or warnings when you press **F11**, you can proceed to generating the database script:

1. On the **Project** menu, click **Overwrite Program Code from UModel Project**. ("Program Code" in the context of databases means the database itself)
2. In the dialog box below, you can choose between merging the changes to the database, or overwriting the database with the changes. For the scope of this example, we will select **Overwrite Code according to Model**. Otherwise, depending on the case, you may want to choose **Merge Model into Code**. For more information, see Code Synchronization Settings 221.

© 2019-2025 Altova GmbH



3. Click **OK**. A database script is generated with the changes you made to the model.



At this stage, you have the following options:

- Open the script in Altova DatabaseSpy for review or execution. For more information about DatabaseSpy, see https://www.altova.com/databasespy.
- Save the script to a file for storage or later execution.
- Click **Execute** and actually run the script against the database. Always take this action only if you fully understand the consequences (namely, the fact that the database will be updated with immediate effect).

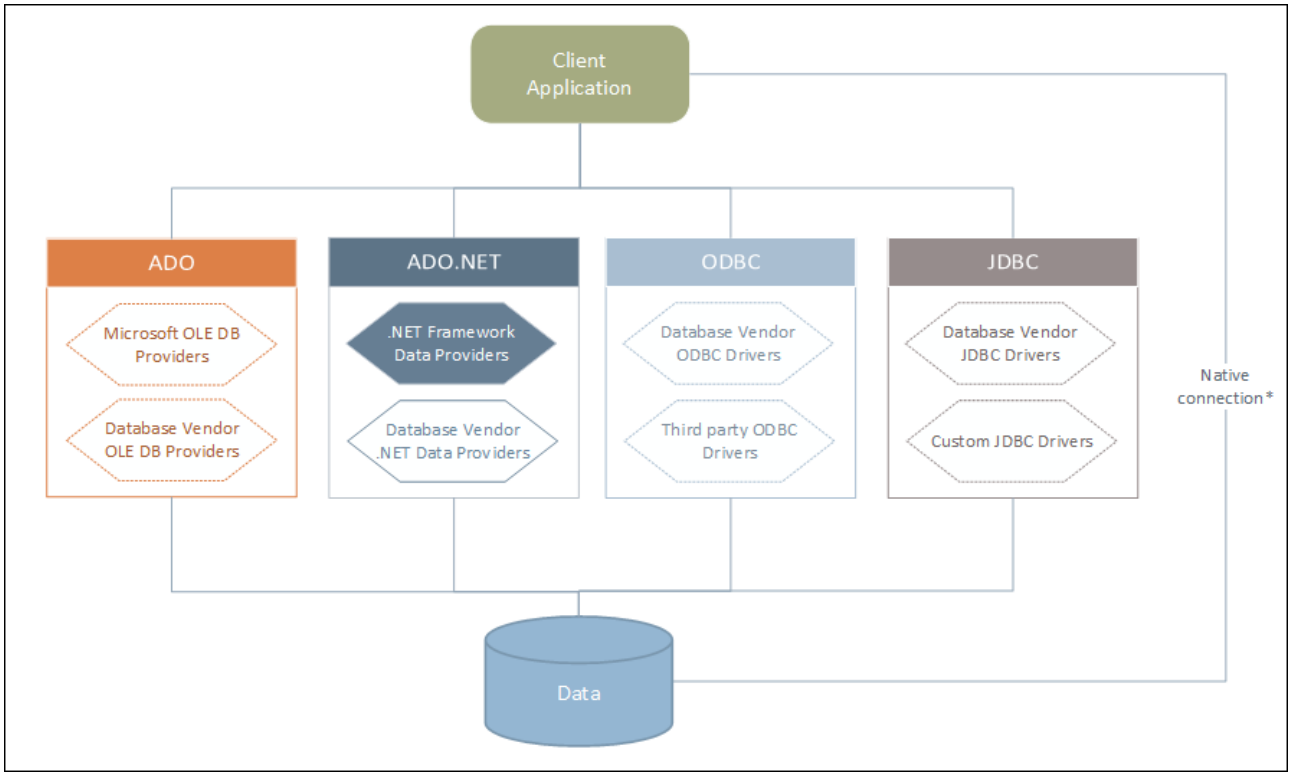
© 2019-2025 Altova GmbH

9.2 Connect to a Data Source

In the most simple case, a database can be a local file such as a Microsoft Access or SQLite database file. In a more advanced scenario, a database may reside on a remote or network database server which does not necessarily use the same operating system as the application that connects to it and consumes data. For example, while UModel runs on a Windows operating system, the database from which you want to access data (for example, MySQL) might run on a Linux machine.

To interact with various database types, both remote and local, UModel relies on the data connection interfaces and database drivers that are already available on your operating system or released periodically by the major database vendors. In the constantly evolving landscape of database technologies, this approach caters for better cross-platform flexibility and interoperability.

The diagram below illustrates data connectivity options available between UModel (illustrated as a generic client application) and a data store (which may be a database server or database file).



** Direct native connections* 532 *are supported for SQLite, MySQL, MariaDB, PostgreSQL databases. To connect to such databases, you do not need to install any additional drivers on your system.*

As shown in the diagram above, UModel can access any of the major database types through the following data access technologies:

- ADO (Microsoft® ActiveX® Data Objects), which, in its turn, uses an underlying OLE DB (Object Linking and Embedding, Database) provider
- ADO.NET (A set of libraries available in the Microsoft .NET Framework that enable interaction with data)

- JDBC (Java Database Connectivity)
- ODBC (Open Database Connectivity)

Note: Some ADO.NET providers are not supported or have limited support. See ADO.NET Support Notes 524.

About data access technologies

The data connection interface you should choose largely depends on your existing software infrastructure. You will typically choose the data access technology and the database driver which integrates tighter with the database system to which you want to connect. For example, to connect to a Microsoft Access 2013 database, you would build an ADO connection string that uses a native provider such as the **Microsoft Office Access Database Engine OLE DB Provider**. To connect to Oracle, on the other hand, you may want to download and install the latest JDBC, ODBC, or ADO.NET interfaces from the Oracle website.

While drivers for Windows products (such as Microsoft Access or SQL Server) may already be available on your Windows operating system, they may not be available for other database types. Major database vendors routinely release publicly available database client software and drivers which provide cross-platform access to the respective database through any combination of ADO, ADO.NET, ODBC, or JDBC. In addition to this, several third party drivers may be available for any of the above technologies. In most cases, there is more than one way to connect to the required database from your operating system, and, consequently, from UModel. The available features, performance parameters, and the known issues will typically vary based on the data access technology or drivers used.

9.2.1 Start Database Connection Wizard

UModel provides a Database Copnnection Wizard that guides you through the steps required to set up a connection to a data source. Before you go through the wizard steps, be aware that for some database types it is necessary to install and separately configure several database prerequisites, such as a database driver or database client software. These are normally provided by the respective database vendors, and include documentation tailored to your specific Windows version. For a list of database drivers grouped by database type, see Database Drivers Overview 510.

To start the Database Connection Wizard (*see screenshot below*), do the following:

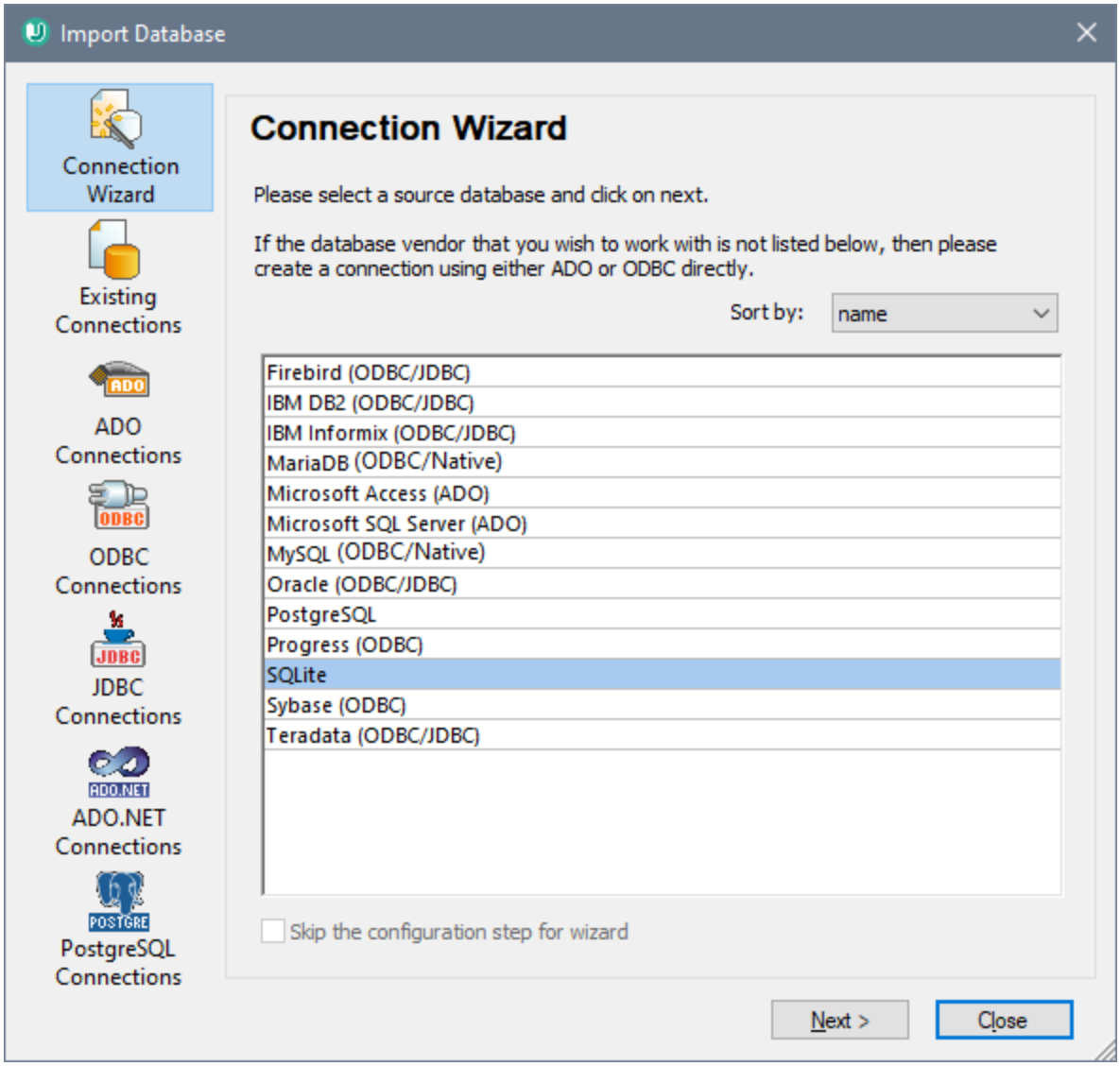
1. On the **Project** menu, click **Import SQL Database**.
2. Click **New**.

The Database Connection Wizard (*screenshot below*) is started. On the left hand side of the window, you can select the most suitable from the following ways to connect to your database:

- Connection Wizard, which prompts you to choose your database type and then guides you through the steps for connecting to a database of that type
- Select an existing connection
- Select a data access technology: ADO, ADO.NET, ODBC, or JDBC
- A native PostgreSQL connection

In the Connection Wizard pane (*see screenshot below*) databases can be sorted alphabetically by the name of the database type or by recent usage. Select the option you want in the *Sort By* combo box. After you have selected the database type to which you want to connect, click **Next**.

 © 2019-2025 Altova GmbH



The wizard will take you through the next steps according to the database type, connection technology (ADO, ADO.NET, ODBC, JDBC), and driver that will be used. For examples applicable to each database type, see Database Connection Examples [533].

Alternatively to using Connection Wizard, you can use one of the following database access technologies:

- Setting up an ADO Connection [514]
- Setting up an ADO.NET Connection [519]
- Setting up an ODBC Connection [528]
- Setting up a JDBC Connection [525]

9.2.2 Database Drivers Overview

This topic gives an overview of database drivers. Even though a number of database drivers might be already available on your Windows operating system, you may still need to download an alternative driver.

Database vendors may provide drivers either as separate downloadable packages, or bundled with database client software. In the latter case, the database client software normally includes any required database drivers, or provides you with an option during installation to select the drivers and components you wish to install. Database client software typically consists of administration and configuration utilities used to simplify database administration and connectivity, as well as documentation on how to install and configure the database client and any of its components.

Configuring the database client correctly is crucial for establishing a successful connection to the database. Before installing and using the database client software, we recommend that you carefully read the installation and configuration instructions of the database client; these may vary for each database version and for each Windows version.

To understand the capabilities and limitations of each data access technology with respect to each database type, refer to the documentation of that particular database product and also test the connection against your specific environment. To avoid common connectivity issues, note the following:

- Some ADO.NET providers are not supported or have limited support. See ADO.NET Support Notes 524.
- When installing a database driver, it is recommended that it has the same platform as the Altova application (32-bit or 64-bit). For example, if you are using a 32-bit Altova application on a 64-bit operating system, install the 32-bit driver, and set up your database connection using the 32-bit driver, see also Viewing the Available ODBC Drivers 530.
- When setting up an ODBC data source, it is recommended that you create the data source name (DSN) as a System DSN instead of as a User DSN. For more information, see Setting up an ODBC Connection 528.
- When setting up a JDBC data source, ensure that JRE (Java Runtime Environment) or Java Development Kit (JDK) is installed and that the `CLASSPATH` environment variable of the operating system is configured. For more information, see Setting up a JDBC Connection 525.
- For the installation instructions and support details of any drivers or database client software that you install from a database vendor, check the documentation provided with the installation package.

Available data-access technologies

The table below lists common database drivers you can use to connect to a particular database through a particular data access technology. Note that this list is neither exhaustive nor prescriptive; you can use other native or third party alternatives in addition to the drivers shown below.

Database	Interface	Drivers
Firebird	ADO.NET	Firebird ADO.NET Data Provider (https://www.firebirdsql.org/en/additional-downloads/)
	JDBC	Firebird JDBC driver (https://www.firebirdsql.org/en/jdbc-driver/)
	ODBC	Firebird ODBC driver (https://www.firebirdsql.org/en/odbc-driver/)

© 2019-2025 Altova GmbH

Database	Interface	Drivers
IBM DB2	ADO	IBM OLE DB Provider for DB2
	ADO.NET	IBM Data Server Provider for .NET
	JDBC	IBM Data Server Driver for JDBC and SQLJ
	ODBC	IBM DB2 ODBC Driver
IBM DB2 for i	ADO	• IBM DB2 for i5/OS IBMDA400 OLE DB Provider • IBM DB2 for i5/OS IBMDARLA OLE DB Provider • IBM DB2 for i5/OS IBMDASQL OLE DB Provider
	ADO.NET	.NET Framework Data Provider for IBM i
	JDBC	IBM Toolbox for Java JDBC Driver
	ODBC	iSeries Access ODBC Driver
IBM Informix	ADO	IBM Informix OLE DB Provider
	JDBC	IBM Informix JDBC Driver
	ODBC	IBM Informix ODBC Driver
Microsoft Access	ADO	• Microsoft Jet OLE DB Provider • Microsoft Access Database Engine OLE DB Provider
	ADO.NET	.NET Framework Data Provider for OLE DB
	ODBC	• Microsoft Access Driver
MariaDB	ADO.NET	In the absence of a dedicated .NET connector for MariaDB, use **Connector/NET** for MySQL (https://dev.mysql.com/downloads/connector/net/).
	JDBC	MariaDB Connector/J (https://downloads.mariadb.org/)

Database	Interface	Drivers
	ODBC	MariaDB Connector/ODBC (https://downloads.mariadb.org/)
	Native connection	Available. No drivers are required.
Microsoft SQL Server	ADO	• Microsoft OLE DB Driver for SQL Server (MSOLEDBSQL) • Microsoft OLE DB Provider for SQL Server (SQLOLEDB) • SQL Server Native Client (SQLNCLI)
	ADO.NET	• .NET Framework Data Provider for SQL Server • .NET Framework Data Provider for OLE DB
	JDBC	• Microsoft JDBC Driver for SQL Server (https://docs.microsoft.com/en-us/sql/connect/jdbc/microsoft-jdbc-driver-for-sql-server)
	ODBC	• ODBC Driver for Microsoft SQL Server (https://docs.microsoft.com/en-us/SQL/connect/odbc/download-odbc-driver-for-sql-server)
MySQL	ADO.NET	• Connector/NET (https://dev.mysql.com/downloads/connector/net/)
	JDBC	Connector/J (https://dev.mysql.com/downloads/connector/j/)
	ODBC	Connector/ODBC (https://dev.mysql.com/downloads/connector/odbc/)
	Native connection	Available for MySQL 5.7 and later. No drivers are required.
Oracle	ADO	• Oracle Provider for OLE DB • Microsoft OLE DB Provider for Oracle
	ADO.NET	Oracle Data Provider for .NET (http://www.oracle.com/technetwork/topics/dotnet/index-085163.html)
	JDBC	• JDBC Thin Driver • JDBC Oracle Call Interface (OCI) Driver

© 2019-2025 Altova GmbH

Database	Interface	Drivers
		These drivers are typically installed during the installation of your Oracle database client. Connect through the OCI Driver (not the Thin Driver) if you are using the Oracle XML DB component.
	ODBC	• Microsoft ODBC for Oracle • Oracle ODBC Driver (typically installed during the installation of your Oracle database client)
PostgreSQL	JDBC	PostgreSQL JDBC Driver (https://jdbc.postgresql.org/download.html)
	ODBC	psqlODBC (https://odbc.postgresql.org/)
	Native connection	Available. No drivers are required.
Progress OpenEdge	JDBC	JDBC Connector (https://www.progress.com/jdbc/openedge)
	ODBC	ODBC Connector (https://www.progress.com/odbc/openedge)
SQLite	Native connection	Available. No drivers are required.
Sybase	ADO	Sybase ASE OLE DB Provider
	JDBC	jConnect™ for JDBC
	ODBC	Sybase ASE ODBC Driver
Teradata	ADO.NET	.NET Data Provider for Teradata (https://downloads.teradata.com/download/connectivity/net-data-provider-for-teradata)
	JDBC	Teradata JDBC Driver (https://downloads.teradata.com/download/connectivity/jdbc-driver)
	ODBC	Teradata ODBC Driver for Windows (https://downloads.teradata.com/download/connectivity/odbc-driver/windows)

9.2.3 ADO Connection

Microsoft ActiveX Data Objects (ADO) is a data access technology that enables you to connect to a variety of data sources through OLE DB. OLE DB is an alternative interface to ODBC or JDBC; it provides uniform access to data in a COM (Component Object Model) environment. ADO is a precursor of the newer ADO.NET[519] and is still one of the possible ways to connect to Microsoft native databases such as Microsoft Access or SQL Server, although you can also use it for other data sources.

Importantly, you can choose between multiple ADO providers, and some of them must be downloaded and installed on your workstation before you can use them. For example, for connecting to SQL Server, the following ADO providers are available:

- Microsoft OLE DB *Driver* for SQL Server (MSOLEDBSQL)
- Microsoft OLE DB *Provider* for SQL Server (SQLOLEDB)
- SQL Server Native Client (SQLNCLI)

From the providers listed above, the recommended one is MSOLEDBSQL; you can download it from https://docs.microsoft.com/en-us/sql/connect/oledb/download-oledb-driver-for-sql-server?view=sql-server-ver15. Note that it must match the platform of UModel (32-bit or 64-bit). The SQLOLEDB and SQLNCLI providers are considered deprecated and thus are not recommended.

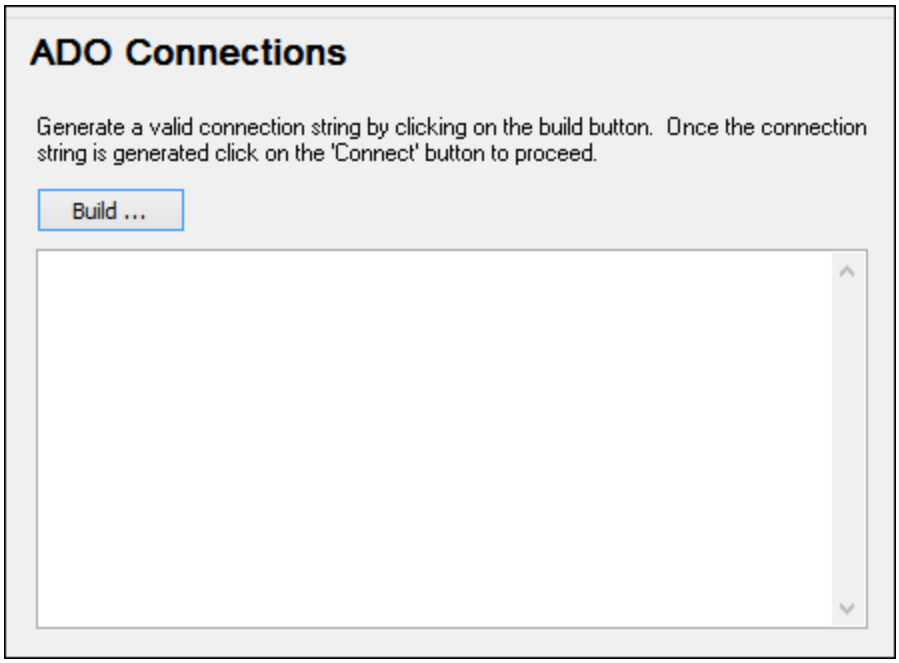
Note: The *Microsoft OLE DB Provider for SQL Server (SQLOLEDB)* is known to have issues with parameter binding of complex queries like Common Table Expressions (CTE) and nested SELECT statements.

Set up an ADO connection

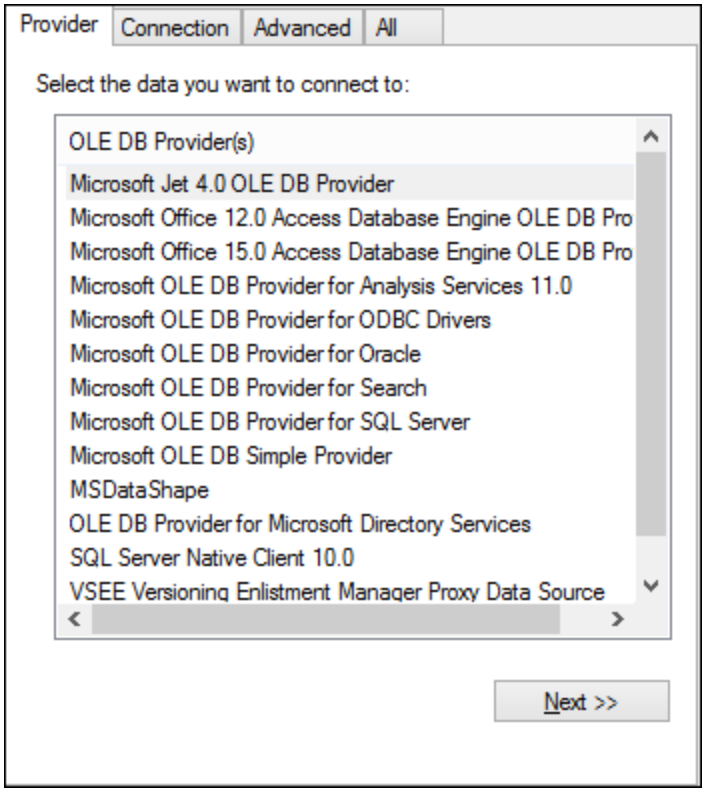
To set up an ADO connection, do the following:

1. Start the database connection wizard[508].
2. Click **ADO Connections**.

© 2019-2025 Altova GmbH



3. Click **Build**.



4. Select the data provider through which you want to connect. The table below lists a few common scenarios.

To connect to this database...	Use this provider...
Microsoft Access	• **Microsoft Office Access Database Engine OLE DB Provider** (recommended) • **Microsoft Jet OLE DB Provider** If the **Microsoft Office Access Database Engine OLE DB Provider** is not available in the list, make sure that you have installed either Microsoft Access or the Microsoft Access Database Engine Redistributable (https://www.microsoft.com/en-us/download/details.aspx?id=54920) on your computer.
SQL Server	• **Microsoft OLE DB Driver for SQL Server (MSOLEDBSQL)** - this is the recommended OLE DB provider. In order for this provider to appear in the list, it must be downloaded from https://docs.microsoft.com/en-us/sql/connect/oledb/download-oledb-driver-for-sql-server?view=sql-server-ver15 and installed. • **Microsoft OLE DB Provider for SQL Server (OLEDBSQL)** • **SQL Server Native Client (SQLNCLI)**
Other database	Select the provider applicable to your database. If an OLE DB provider to your database is not available, install the required driver from the database vendor (see Database Drivers Overview 510). Alternatively, set up an ADO.NET, ODBC, or JDBC connection. If the operating system has an ODBC driver to the required database, you could also use the **Microsoft OLE DB Provider for ODBC Drivers**, or preferably opt for an ODBC connection 528.

5. Having selected the provider of choice, click **Next** and complete the wizard.

The subsequent wizard steps are specific to the provider you chose. For SQL Server, you will need to provide or select the host name of the database server, the authentication method, the database name, as well as the database username and password. For an example, see Connecting to Microsoft SQL Server (ADO) 548. For Microsoft Access, you will be asked to browse for or provide the path to the database file. For an example, see Connecting to Microsoft Access (ADO) 546.

The complete list of initialization properties (connection parameters) is available in the **All** tab of the connection dialog box—these properties vary depending on the chosen provider and may need to be set explicitly in order for the connection to be possible. The following sections provide guidance on configuring the basic initialization properties for Microsoft Access and SQL Server databases:

- Setting up the SQL Server Data Link Properties 517
- Setting up the Microsoft Access Data Link Properties 518

© 2019-2025 Altova GmbH

9.2.3.1 Connect to an Existing MS Access Database

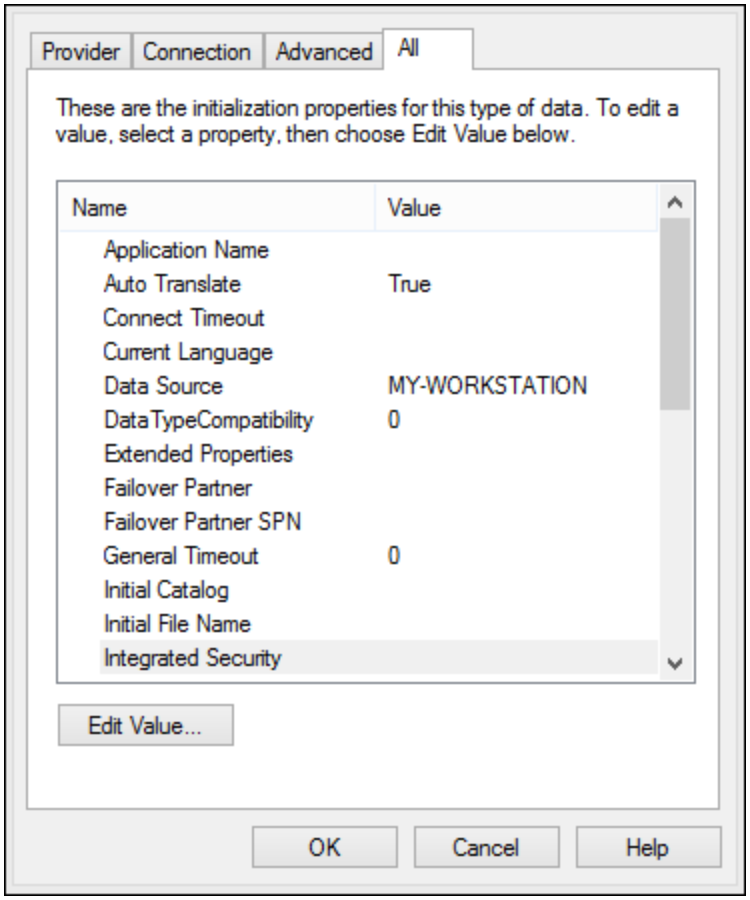
The procedure given below is suitable if you want to connect to a Microsoft Access database that is not password-protected. If the database is password-protected, use the method given in the Connection Example for Microsoft Access (ADO) 546.

1. Run the database connection wizard (see Starting the Database Connection Wizard 508).
2. Select **Microsoft Access (ADO)**, and then click **Next**.
3. Browse for the database file, or enter the path to it (either relative or absolute).
4. Click **Connect**.

9.2.3.2 Set Up SQL Server Data Link Properties

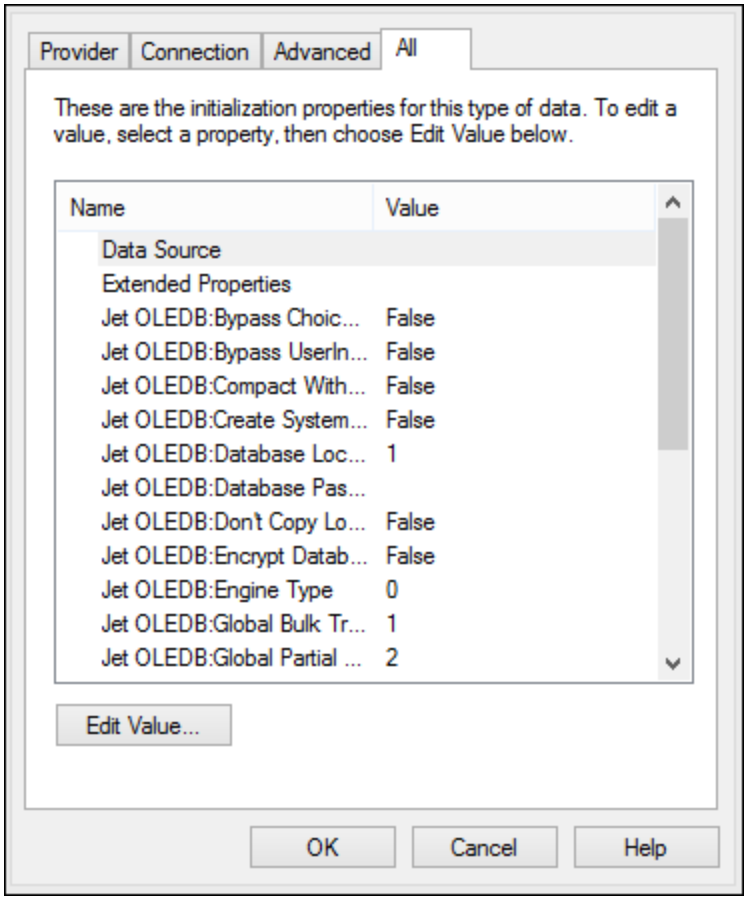
If you connect to a Microsoft SQL Server database through ADO 514, check the following properties in the *All* tab of the Data Link Properties dialog box (*screenshot below*).

- *Integrated Security:* If SQL Server Native Client is the data provider on the Provider tab, set this property to a space character.
- *Persist Security Info:* Set this property to *True*.



9.2.3.3 Set Up MS Access Data Link Properties

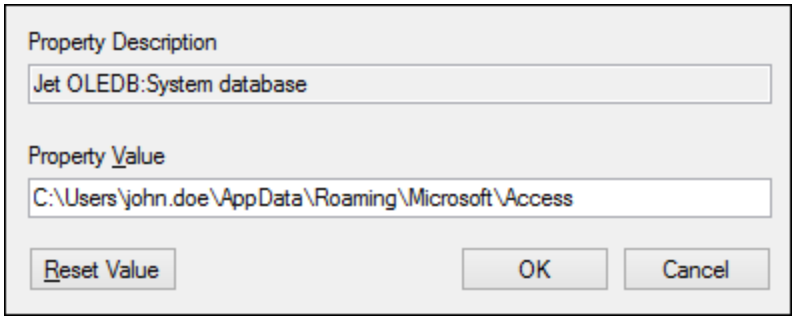
If you connect to a Microsoft Access database through ADO[514], then in the *All* tab of the Data Link Properties dialog box (*screenshot below*), check the properties listed below the screenshot.



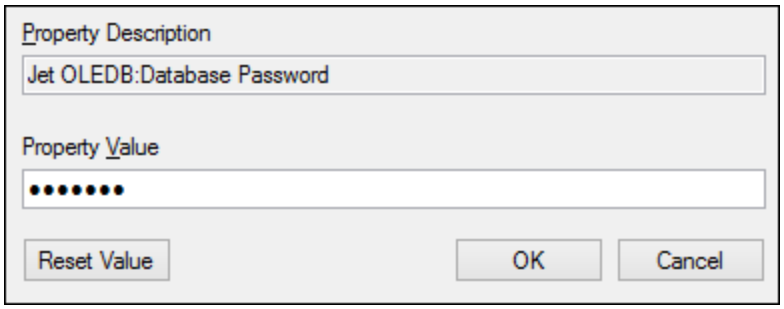
Data Source
This property stores the path to the Microsoft Access database file. We recommend using the UNC (Universal Naming Convention) path format, for example: `\\anyserver\share$\filepath`

Jet OLEDB:System Database
This property stores the path to the workgroup information file. You may need to explicitly set the value of this property before you can connect to a Microsoft Access database. If you cannot connect due to a "workgroup information file" error, locate the workgroup information file (`System.MDW`) applicable to your user profile, and set the property value to the path of the `System.MDW` file.

© 2019-2025 Altova GmbH



Jet OLEDB:Database Password
If the database is password-protected, set the value of this property to the database password.



9.2.4 ADO.NET Connection

ADO.NET is a set of Microsoft .NET Framework libraries designed to interact with data, including data from databases. To connect to a database from UModel through ADO.NET, Microsoft .NET Framework 4 or later is required. Connect to a database through ADO.NET by selecting a .NET provider and supplying a connection string.

A .NET data provider is a collection of classes that enables connecting to a particular type of data source (for example, a SQL Server, or an Oracle database), executing commands against it, and fetching data from it. So, when you use ADO.NET, UModel interacts with a database through a data provider—one that is optimized to work with the specific type of data source that it is designed for.

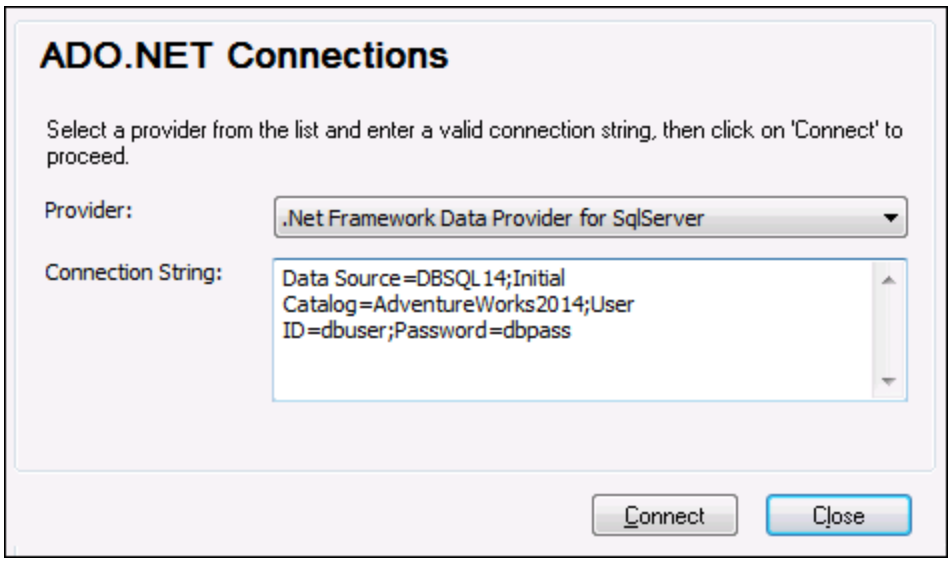
There are two types of .NET providers:

- Supplied by default with Microsoft .NET Framework.
- Supplied by major database vendors, as an extension to the .NET Framework. Such ADO.NET providers must be installed separately and can typically be downloaded from the website of the respective database vendor.

Note: Certain ADO.NET providers are not supported or have limited support. See ADO.NET Support Notes 524.

Set up the connection as follows.

1. Start the database connection wizard [508].
2. Click **ADO.NET Connections**.
3. Select a .NET data provider from the list. The list of providers available by default with the .NET Framework appears in the "Provider" list. Vendor-specific .NET data providers are available in the list only if they are already installed on your system. To become available, vendor-specific .NET providers must be installed into the GAC (Global Assembly Cache) by running the `.msi` or `.exe` file supplied by the database vendor.
4. Enter a database connection string. A connection string defines the database connection information as semicolon-delimited key/value pairs of connection parameters. For example, a connection string such as `Data Source=DBSQLSERV;Initial Catalog=ProductsDB;User ID=dbuser;Password=dbpass` connects to the SQL Server database `ProductsDB` on server `DBSQLSERV`, with the user name `dbuser` and password `dbpass`. You can create a connection string by typing the key/value pairs directly into the "Connection String" dialog box. Another option is to create it with Visual Studio (see Creating a Connection String in Visual Studio [520]). The syntax of the connection string depends on the provider selected from the "Provider" list. For examples, see Sample ADO.NET Connection Strings [523].



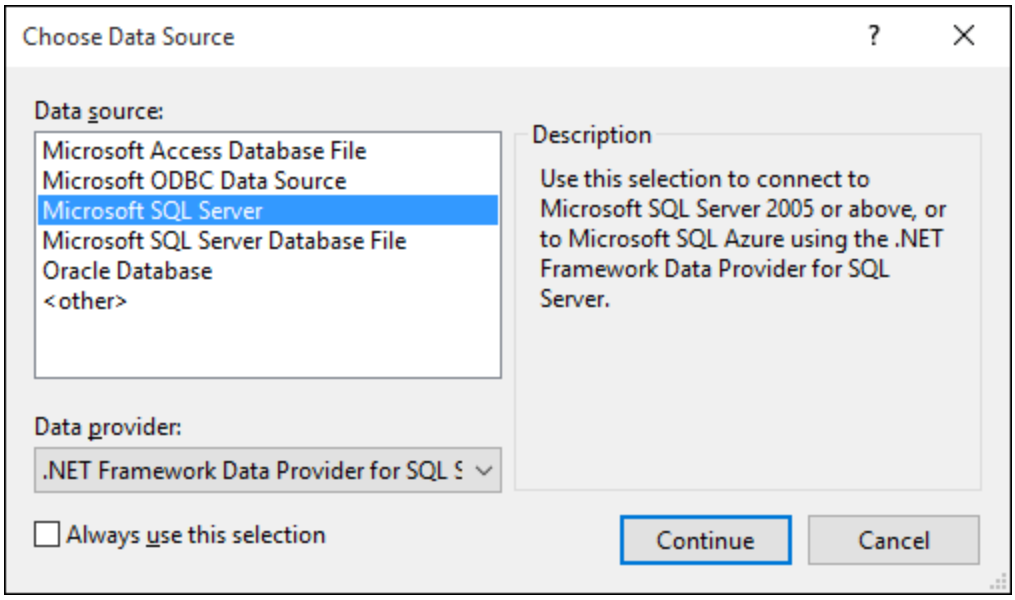
5. Click Connect.

9.2.4.1 Creating a Connection String in Visual Studio

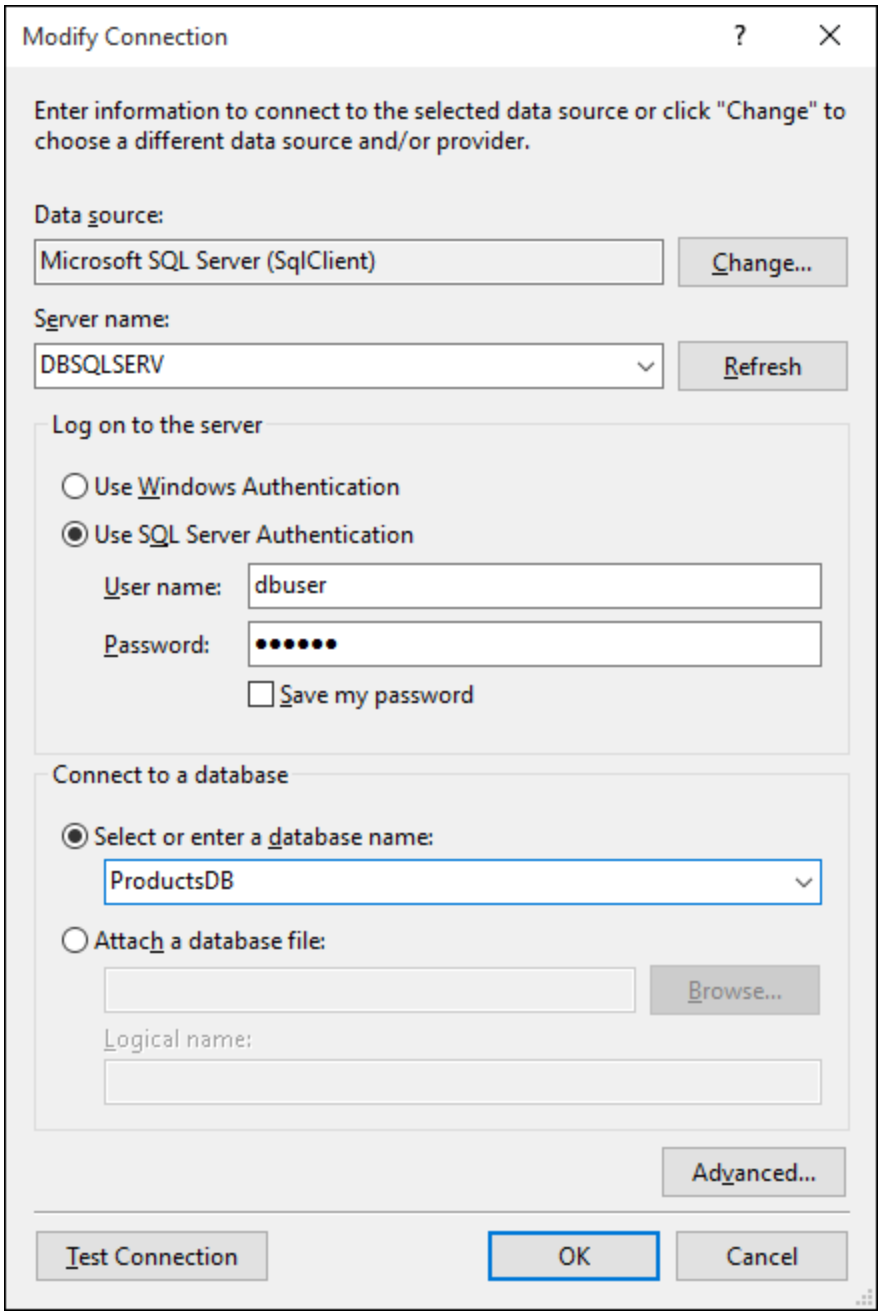
In order to connect to a data source using ADO.NET, a valid database connection string is required. The following instructions show you how to create a connection string from Visual Studio.

1. On the **Tools** menu, click **Connect to Database**.
2. Select a data source from the list (in this example, Microsoft SQL Server). The Data Provider is filled automatically based on your choice.

© 2019-2025 Altova GmbH



3. Click **Continue**.



4. Enter the server host name and the user name and password to the database. In this example, we are connecting to the database `ProductsDB` on server `DBSQLSERV`, using SQL Server authentication.
5. Click **OK**.

If the database connection is successful, it appears in the Server Explorer window. You can display the Server Explorer window using the menu command **View | Server Explorer**. To obtain the database connection string, right-click the connection in the Server Explorer window, and select **Properties**. The connection string is now displayed in the Properties window of Visual Studio. Note that, before pasting the string into the "Connection String" box of UModel, you will need to replace the asterisk (*) characters with the actual password.

© 2019-2025 Altova GmbH

9.2.4.2 Sample ADO.NET Connection Strings

To set up an ADO.NET connection, you need to select an ADO.NET provider from the database connection dialog box and enter a connection string (see also Setting up an ADO.NET Connection 519). Sample ADO.NET connection strings for various databases are listed below under the .NET provider where they apply.

.NET Data Provider for Teradata
This provider can be downloaded from Teradata website (https://downloads.teradata.com/download/connectivity/net-data-provider-for-teradata). A sample connection string looks as follows:

```
Data Source=ServerAddress;User Id=user;Password=password;
```

.NET Framework Data Provider for IBM i
This provider is installed as part of *IBM i Access Client Solutions - Windows Application Package*. A sample connection string looks as follows:

```
DataSource=ServerAddress;UserID=user;Password=password;DataCompression=True;
```

For more information, see the ".NET Provider Technical Reference" help file included in the installation package above.

.NET Framework Data Provider for MySQL
This provider can be downloaded from MySQL website (https://dev.mysql.com/downloads/connector/net/). A sample connection string looks as follows:

```
Server=127.0.0.1;Uid=root;Pwd=12345;Database=test;
```

See also: https://dev.mysql.com/doc/connector-net/en/connector-net-programming-connecting-connection-string.html

.NET Framework Data Provider for SQL Server
A sample connection string looks as follows:

```
Data Source=DBSQLSERV;Initial Catalog=ProductsDB;User ID=dbuser;Password=dbpass
```

See also: https://msdn.microsoft.com/en-us/library/ms254500(v=vs.110).aspx

IBM DB2 Data Provider 10.1.2 for .NET Framework 4.0
A sample connection string looks as follows:

```
Database=PRODUCTS;UID=user;Password=password;Server=localhost:50000;
```

Note: This provider is typically installed with the IBM DB2 Data Server Client package. If the provider is missing from the list of ADO.NET providers after installing IBM DB2 Data Server Client package, refer to the following technical note: https://www-01.ibm.com/support/docview.wss?uid=swg21429586. See also: https://www.ibm.com/support/knowledgecenter/en/SSEPGG_10.1.0/com.ibm.swg.im.dbclient.adonet.ref.doc/doc/DB2ConnectionClassConnectionStringProperty.html

Oracle Data Provider for .NET (ODP.NET)
The installation package which includes the ODP.NET provider can be downloaded from the Oracle website (see http://www.oracle.com/technetwork/topics/dotnet/downloads/index.html). A sample connection string looks as follows:

```
Data Source=DSORCL;User Id=user;Password=password;
```

where `DSORCL` is the name of the data source which points to an Oracle service name defined in the **`tnsnames.ora`** file, as described in Connecting to Oracle (ODBC) 557.

To connect without configuring a service name in the **`tnsnames.ora`** file, use a string such as:

```
Data Source=(DESCRIPTION=(ADDRESS_LIST=(ADDRESS=(PROTOCOL=TCP)(HOST=host)(PORT=port)))
(CONNECT_DATA=(SERVER=DEDICATED)(SERVICE_NAME=MyOracleSID)));User
Id=user;Password=password;
```

See also: https://docs.oracle.com/cd/B28359_01/win.111/b28375/featConnecting.htm.

9.2.4.3 ADO.NET Support Notes

The following table lists known ADO.NET database drivers that are currently not supported or have limited support in UModel.

Database	Driver	Support notes
All databases	**.Net Framework Data Provider for ODBC**	Limited support. Known issues exist with Microsoft Access connections. It is recommended to use ODBC direct connections instead.
	.Net Framework Data Provider for OleDb	Limited support. Known issues exist with Microsoft Access connections. It is recommended to use ADO direct connections instead.
Firebird	**Firebird ADO.NET Data Provider**	Limited support. It is recommended to use ODBC or JDBC instead.
Informix	**IBM Informix Data Provider for .NET Framework 4.0**	Not supported. Use **DB2 Data Server Provider** instead.
IBM DB2 for i (iSeries)	**.Net Framework Data Provider for i5/OS**	Not supported. Use **.Net Framework Data Provider for IBM i** instead, installed as part of the *IBM i Access Client Solutions - Windows Application Package*.
Oracle	**.Net Framework Data Provider for Oracle**	Limited support. Although this driver is provided with the .NET Framework, its usage is discouraged by Microsoft, because it is deprecated.
PostgreSQL	—	No ADO.NET drivers for this vendor are supported. Use a native connection instead.

© 2019-2025 Altova GmbH

Database	Driver	Support notes
Sybase	—	No ADO.NET drivers for this vendor are supported.

9.2.5 JDBC Connection

JDBC (Java Database Connectivity) is a database access interface which is part of the Java software platform from Oracle. JDBC connections are generally more resource-intensive than ODBC connections but may provide features not available through ODBC.

Prerequisites

- JRE (Java Runtime Environment) or Java Development Kit (JDK) must be installed. This may be either Oracle JDK or an open source build such as Oracle OpenJDK. UModel will determine the path to the Java Virtual Machine (JVM) from the following locations, in this order: (i) the custom JVM path you may have set in application **Options**; see Java Settings 694; (ii) the JVM path found in the Windows registry; (iii) the `JAVA_HOME` environment variable.
- Make sure that the platform of UModel (32-bit, 64-bit) matches that of the JRE/JDK.
- The JDBC drivers from the database vendor must be installed. These may be JDBC drivers installed as part of a database client installation, or supported JDBC libraries (`.jar` files) that are downloaded separately. See also Database Connection Examples 533.
- The `CLASSPATH` environment variable must include the path to the JDBC driver (one or several `.jar` files) on your Windows operating system. When you install some database clients, the installer may configure this variable automatically. See also Configuring the CLASSPATH 527.

Connect to SQL Server via JDBC with Windows credentials

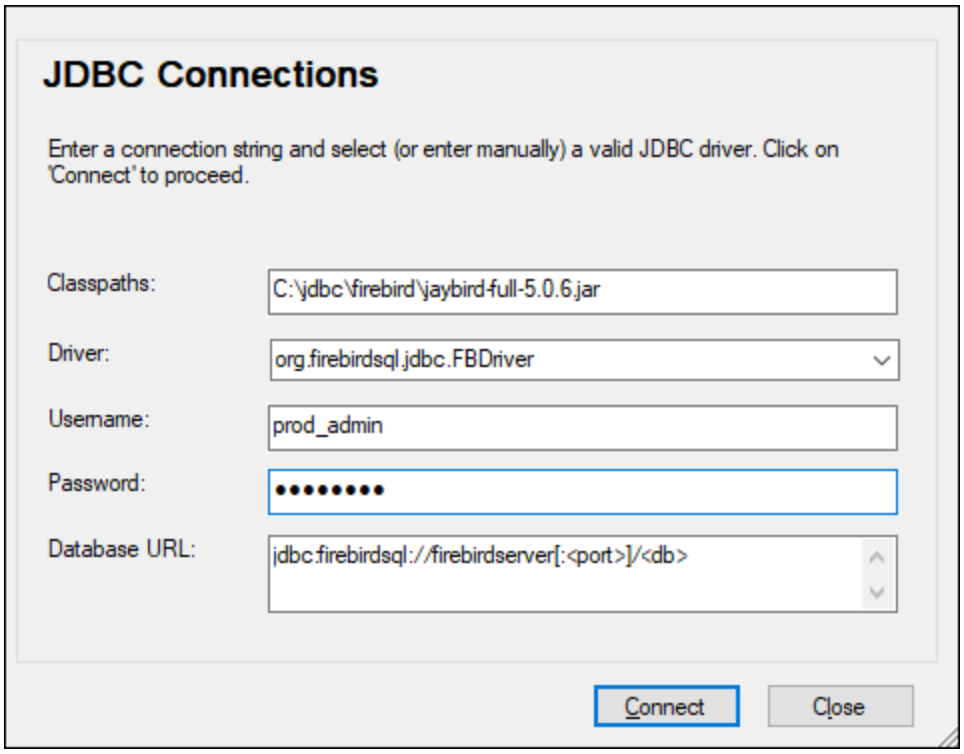
If you connect to SQL Server through JDBC with Windows credentials (integrated security), note the following:

- The **`sqljdbc_auth.dll`** file included in the JDBC driver package must be copied to a directory that is on the system `PATH` environment variable. There are two such files, one for the x86 and one for x64 platform. Make sure that you add to the `PATH` the one that corresponds to your JDK platform.
- The JDBC connection string must include the property **`integratedSecurity=true`**.

For further information, refer to *Microsoft JDBC driver for SQL Server* documentation, https://docs.microsoft.com/en-us/sql/connect/jdbc/building-the-connection-url.

Set up a JDBC connection

1. Start the database connection wizard 508 and click JDBC Connections.
2. If required, enter a semicolon-separated list of `.jar` file paths in the *Classpaths* field. The `.jar` libraries entered here will be loaded into the environment in addition to those already defined in the `CLASSPATH` 527 environment variable. JDBC drivers found in the source .jar libraries referenced via the *Classpaths* field and the system's `CLASSPATH` 527 are listed in the Driver dropdown list (see next step).



3. In the *Driver* field, select a JDBC driver from the list or enter a Java class name. The list will contain the JDBC drivers configured through in the *Classpaths* field (see above) and the `CLASSPATH` [527] environment variable.

> The JDBC driver paths defined in the `CLASSPATH` variable, as well as any .jar file paths entered directly in the database connection dialog box are all supplied to the Java Virtual Machine (JVM). The JVM then decides which drivers to use in order to establish a connection. It is recommended that you keep track of Java classes loaded into the JVM so as not to create potential JDBC driver conflicts and avoid unexpected results when connecting to the database.

4. Enter the username and password of the database in the corresponding fields.
5. In the *Database URL* field, enter the JDBC connection URL (JDBC connection string) in the format specific to your database type. The following table describes the syntax of JDBC connection strings for common database types.

Database	JDBC Connection URL
Firebird	`jdbc:firebirdsql://<host>[:<port>]/<database path or alias>`
IBM DB2	`jdbc:db2://hostName:port/databaseName`
IBM DB2 for i	`jdbc:as400://[host]`
IBM Informix	`jdbc:informix-sqli://hostName:port/databaseName:INFORMIXSERVER=myserver`

© 2019-2025 Altova GmbH

Database	JDBC Connection URL
MariaDB	`jdbc:mariadb://hostName:port/databaseName`
Microsoft SQL Server	`jdbc:sqlserver://hostName:port;databaseName=name`
MySQL	`jdbc:mysql://hostName:port/databaseName`
Oracle	`jdbc:oracle:thin:@hostName:port:SID` `jdbc:oracle:thin:@//hostName:port/service`
Oracle XML DB	`jdbc:oracle:oci:@//hostName:port:service`
PostgreSQL	`jdbc:postgresql://hostName:port/databaseName`
Progress OpenEdge	`jdbc:datadirect:openedge://host:port;databaseName=db_name`
Sybase	`jdbc:sybase:Tds:hostName:port/databaseName`
Teradata	`jdbc:teradata://databaseServerName`

Note that syntax variations of the formats listed above are also possible. For example, the database URL may exclude the port or may include the username and password of the database. Check the documentation of the database vendor for further details.

6. Click **Connect**.

9.2.5.1 Configuring the CLASSPATH

The `CLASSPATH` environment variable is used by the Java Runtime Environment (JRE) or the Java Development Kit (JDK) to locate Java classes and other resource files on your operating system. When you connect to a database through JDBC, this variable must be configured to include the path to the JDBC driver on your operating system, and, in some cases, the path to additional library files specific to the database type you are using.

The following table lists sample file paths that must be typically included in the `CLASSPATH` variable. Importantly, you may need to adjust this information based on the location of the JDBC driver on your system, the JDBC driver name, as well as the JRE/JDK version present on your operating system. To avoid connectivity problems, check the installation instructions and any pre-installation or post-installation configuration steps applicable to the JDBC driver installed on your operating system.

Database	Sample CLASSPATH entries
Firebird	`C:\Program Files\Firebird\Jaybird-2.2.8-JDK_1.8\jaybird-full-2.2.8.jar`
IBM DB2	`C:\Program Files (x86)\IBM\SQLLIB\java\db2jcc.jar;C:\Program Files (x86)\IBM\SQLLIB\java\db2jcc_license_cu.jar;`
IBM DB2 for i	`C:\jt400\jt400.jar;`
IBM Informix	`C:\Informix_JDBC_Driver\lib\ifxjdbc.jar;`

Database	Sample CLASSPATH entries
Microsoft SQL Server	C:\Program Files\Microsoft JDBC Driver 4.0 for SQL Server\sqljdbc_4.0\enu\sqljdbc.jar
MariaDB	**<installation directory>**\mariadb-java-client-2.2.0.jar
MySQL	**<installation directory>**\mysql-connector-java-*version*-bin.jar;
Oracle	**ORACLE_HOME**\jdbc\lib\ojdbc6.jar;
Oracle (with XML DB)	**ORACLE_HOME**\jdbc\lib\ojdbc6.jar;**ORACLE_HOME**\LIB\xmlparserv2.jar; **ORACLE_HOME**\RDBMS\jlib\xdb.jar;
PostgreSQL	**<installation directory>**\postgresql.jar
Progress OpenEdge	%DLC%\java\openedge.jar;%DLC%\java\pool.jar; Note: Assuming the Progress OpenEdge SDK is installed on the machine, % DLC% is the directory where OpenEdge is installed.
Sybase	C:\sybase\jConnect-7_0\classes\jconn4.jar
Teradata	**<installation directory>**\tdgssconfig.jar;**<installation directory>**\terajdbc4.jar

Note the following points:

- The CLASSPATH setting is available in the Environment Variables setting of your Windows system. Include in the CLASSPATH the path where the JDBC driver is located, separating it from other paths by a semi-colon.
- Changing the CLASSPATH variable may affect the behavior of Java applications on your machine. To understand possible implications before you proceed, refer to the Java documentation.
- Environment variables can be user or system. To change system environment variables, you need administrative rights on the operating system.
- After you change the environment variable, restart any running programs for settings to take effect. Alternatively, log off or restart your operating system.

9.2.6 ODBC Connection

ODBC (Open Database Connectivity) is a widely used data access technology that enables you to connect to a database from UModel. It can be used either as primary means to connect to a database, or as an alternative to native, OLE DB, or JDBC-driven connections.

To connect to a database through ODBC, first you need to create an ODBC data source name (DSN) on the operating system. This step is not required if the DSN has already been created, perhaps by another user of the operating system. The DSN represents a uniform way to describe the database connection to any ODBC-aware client application on the operating system, including UModel. DSNs can be of the following types:

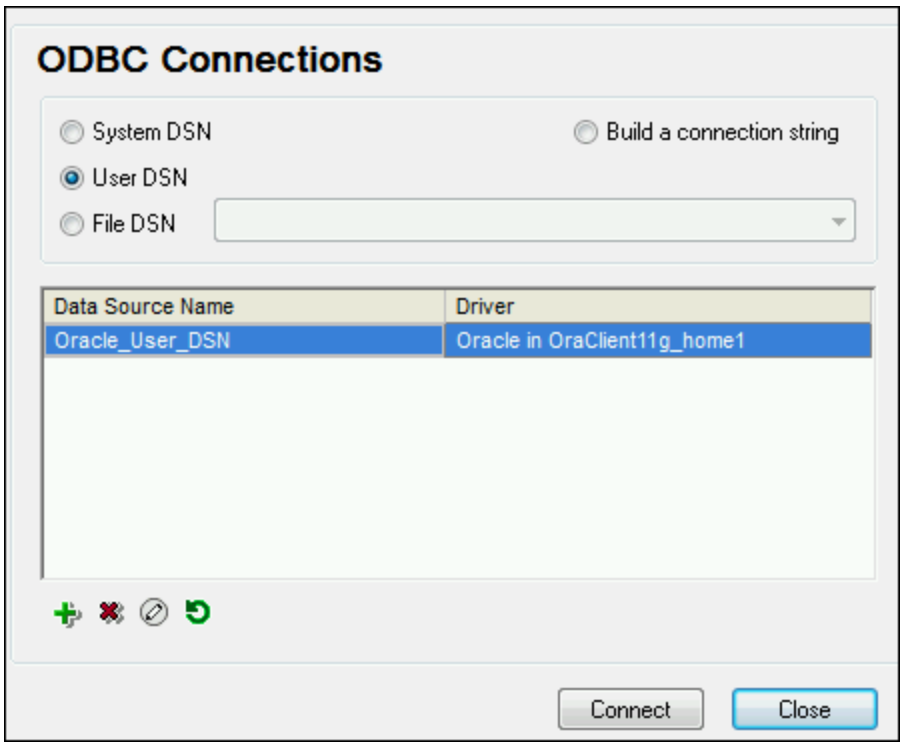
- System DSN

© 2019-2025 Altova GmbH

- User DSN
- File DSN

A *System* data source is accessible by all users with privileges on the operating system. A *User* data source is available to the user who created it. Finally, if you create a *File DSN*, the data source will be created as a file with the .dsn extension which you can share with other users, provided that they have installed the drivers used by the data source.

Any DSNs already available on your machine are listed by the database connection dialog box when you click **ODBC connections** on the ODBC connections dialog box (*screenshot below*).



If a DSN to the required database is not available, the UModel database connection wizard will assist you to create it; however, you can also create it directly on your Windows operating system. In either case, before you proceed, ensure that the ODBC driver applicable for your database is in the list of ODBC drivers available to the operating system (see Viewing the Available ODBC Drivers 530).

Connect with a new DSN

To connect with a new DSN, carry out the following steps. Note that, to create a System DSN, you need administrative rights on the operating system and UModel must be run as administrator..

1. Start the database connection wizard 508.
2. On the database connection dialog box, click **ODBC Connections**.
3. Select a data source type (User DSN, System DSN, File DSN).
4. Click **Add** .
5. Select a driver, and then click **User DSN** or **System DSN** (depending on the type of the DSN you want to create). If the driver applicable to your database is not listed, download it from the database vendor and install it (see Database Drivers Overview 510).

6. On the dialog box that pops up, fill in any driver specific connection information to complete the setup.

For the connection to be successful, you will need to provide the host name (or IP address) of the database server, as well as the database username and password. There may be other optional connection parameters—these parameters vary between database providers. For detailed information about the parameters specific to each connection method, consult the documentation of the driver provider. Once created, the DSN becomes available in the list of data source names. This enables you to reuse the database connection details any time you want to connect to the database. Note that User DSNs are added to the list of User DSNs whereas System DSNs are added to the list of System DSNs.

Connect with an existing DSN

To connect with an existing DSN, do the following.

1. Start the database connection wizard[508].
2. Click **ODBC Connections**.
3. Choose the type of the existing data source (User DSN, System DSN, File DSN).
4. Click the existing DSN record, and then click **Connect**.

Build a connection string based on an existing .dsn file

Do this as follows.

1. Start the database connection wizard[508].
2. Click **ODBC Connections**.
3. Select **Build a connection string** and then click **Build**.
4. If you want to build the connection string using a File DSN, click the *File Data Source* tab. Otherwise, click the *Machine Data Source* tab. (System DSNs and User DSNs are known as "Machine" data sources.)
5. Select the required `.dsn` file, and then click **OK**.

Connect by using a prepared connection string

Do this as follows.

1. Start the database connection wizard[508].
2. Click **ODBC Connections**.
3. Select **Build a connection string**.
4. Paste the connection string into the provided box, and then click **Connect**.

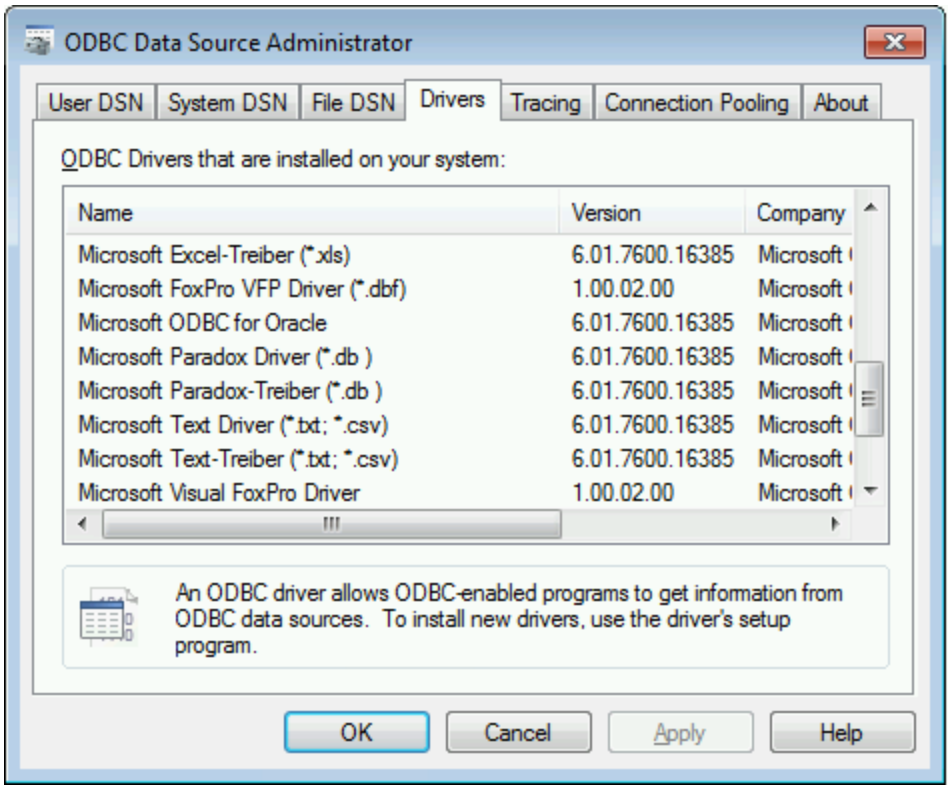
9.2.6.1 Available ODBC Drivers

You can view the ODBC drivers available on your operating system in the ODBC Data Source Administrator. You can access the ODBC Data Source Administrator (`Odbcad32.exe`) from the Windows Control Panel, under Administrative Tools. On 64-bit operating systems, there are two versions of this executable:

- The 32-bit version of the `Odbcad32.exe` file is located in the `C:\Windows\SysWoW64` directory (assuming that `C:` is your system drive).
- The 64-bit version of the `Odbcad32.exe` file is located in the `C:\Windows\System32` directory.

© 2019-2025 Altova GmbH

Any installed 32-bit database drivers are visible in the 32-bit version of ODBC Data Source Administrator (*screenshot below*), while 64-bit drivers—in the 64-bit version. Therefore, ensure that you check the database drivers from the relevant version of ODBC Data Source Administrator.



If the driver to your target database does not exist in the list, or if you want to add an alternative driver, you will need to download it from the database vendor (see Database Drivers Overview 510). Once the ODBC driver is available on your system, you are ready to create ODBC connections with it (see Setting up an ODBC Connection 528).

9.2.7 SQLite Connection

SQLite is a file-based, self-contained database type, which makes it ideal in scenarios where portability and ease of configuration is important. Since SQLite databases are natively supported by UModel, you do not need to install any drivers to connect to them.

SQLite database support notes

- On Linux, statement execution timeout for SQLite databases is not supported.
- Full text search tables are not supported.
- SQLite allows values of different data types in each row of a given table. All processed values must be compatible with the declared column type; therefore, unexpected values can be retrieved and run-time errors may occur if your SQLite database has row values which are not the same as the declared column type.

Important: It is recommended to create tables with the `STRICT` keyword to ensure more predictable behavior of your data. Otherwise, the data may not be read or written correctly when values of different types are mixed in one column. To find out more about STRICT tables, see the SQLite documentation.

9.2.7.1 Connect to an Existing SQLite Database

To connect to an existing SQLite database, do the following.

1. Run the database connection wizard (see Starting the Database Connection Wizard[508]).
2. Select **SQLite**, and then click **Next**.
3. Browse for the SQLite database file, or enter the path (either relative or absolute) to the database. The **Connect** button becomes enabled once you enter the path to a SQLite database file.
4. Click **Connect**.

9.2.8 Native Connections

Native connections are direct connections to a database's own network protocols and drivers. Therefore, no additional drivers need to be installed. Native connections provide the most efficient method of interacting with the database and full feature support.

You can set up native connections to the following DBs:

- MariaDB
- MySQL
- SQLite
- PostgreSQL

Connection setup

To set up a native connection, follow the steps below:

1. Start the database connection wizard[508].
2. Select the DB you want to connect to.
3. In the dialog that appears, enter the relevant connection details, for example, the host (e.g., *localhost*), optionally the port (typically 5432), SSL Mode in the case of MySQL, the database name, username, and password in the corresponding boxes.
4. Click **Connect**.

SQLite conections

For detailed information about SQLite connections, see the topic SQLite Connection[531].

Notes for PostrgreSQL

If the PostgreSQL database server is on a different machine, note the following:

- The PostgreSQL database server must be configured to accept connections from clients. Specifically, the **pg_hba.conf** file must be configured to allow non-local connections. Secondly, the

© 2019-2025 Altova GmbH

postgresql.conf file must be configured to listen on specified IP address(es) and port. For more information, check the PostgreSQL documentation (https://www.postgresql.org/docs/9.5/static/client-authentication-problems.html).
- The server machine must be configured to accept connections on the designated port (typically, 5432) through the firewall. For example, on a database server running on Windows, a rule may need to be created to allow connections on port 5432 through the firewall, from **Control Panel > Windows Firewall > Advanced Settings > Inbound Rules**.

9.2.9 Database Connection Examples

This section includes examples for connecting to a database from UModel through ADO, ODBC, or JDBC. The ADO.NET connection examples are listed separately, see Sample ADO.NET Connection Strings 523. For instructions about establishing a native connection to PostgreSQL and SQLite, see Setting up a PostgreSQL Connection 532 and Setting up a SQLite Connection 531, respectively.

Note the following points:

- The instructions may differ if your Windows configuration, network environment and the database client or server software are not the same as the ones described in each example.
- For most database types, it is possible to connect using more than one data access technology (ADO, ADO.NET, ODBC, JDBC) or driver. The performance of the database connection, as well as its features and limitations will depend on the selected driver, database client software (if applicable), and any additional connectivity parameters that you may have configured outside UModel.

9.2.9.1 Firebird (JDBC)

This example illustrates how to connect to a Firebird database via JDBC.

Prerequisites

- JRE (Java Runtime Environment) or Java Development Kit (JDK) must be installed. This may be either Oracle JDK or an open source build such as Oracle OpenJDK. UModel will determine the path to the Java Virtual Machine (JVM) from the following locations, in this order: (i) the custom JVM path you may have set in application **Options**; see Java Settings 694; (ii) the JVM path found in the Windows registry; (iii) the `JAVA_HOME` environment variable.
- Make sure that the platform of UModel (32-bit, 64-bit) matches that of the JRE/JDK.
- The Firebird JDBC driver must be available on your operating system (it takes the form of a .jar file which provides connectivity to the database). The driver can be downloaded from the Firebird website (https://www.firebirdsql.org/). This example uses *Jaybird 5.0.6*.
- You have the following database connection details: host, port, database path, name, or alias, username, and password.

Connection

1. Start the database connection wizard 508 and click **JDBC Connections** (*see JDBC Connection 525 for a screenshot of the dialog*).
2. In the *Classpaths* field, enter the path to the `.jar` file that provides connectivity to the database. If necessary, you can also enter a semicolon-separated list of `.jar` file paths. If you have added the

filepath to the CLASSPATH[527] of the system, you can leave this field empty.
3. In the *Driver* field, select the appropriate driver. Relevant drivers will be available only if a valid `.jar` file path is in the *Classpaths* field or in the `CLASSPATH` environment variable.
4. Enter the username and password for the database in the corresponding fields.
5. Enter the JDBC connection string in the *Database URL* field according to the pattern in the table below, replacing the highlighted values with the ones applicable to your database server.
6. Click **Connect**.

Connection details of the Firebird example

Field	Value
Classpaths	`C:\jdbc\firebird\jaybird-full-5.0.6.jar`
Driver	*org.firebirdsql.jdbc.FBDriver*
Database URL	`jdbc:firebirdsql://<host>[:<port>]/<database path or alias>`

9.2.9.2 Firebird (ODBC)

This example illustrates how to connect to a Firebird 2.5.4 database running on a Linux server.

Prerequisites

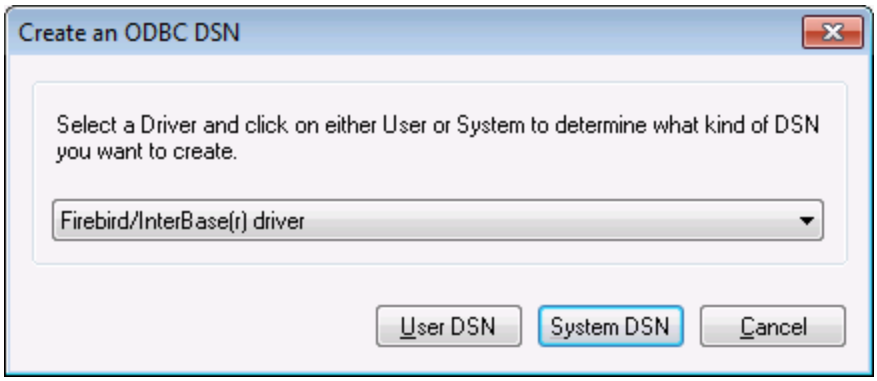
- The Firebird database server is configured to accept TCP/IP connections from clients.
- The Firebird ODBC driver must be installed on your operating system. This example uses the Firebird ODBC driver version 2.0.3.154 downloaded from the Firebird website.
- The Firebird client must be installed on your operating system. Note that there is no standalone installer available for the Firebird 2.5.4 client; the client is part of the Firebird server installation package. You can download the Firebird server installation package from the Firebird website. Look for *Windows executable installer for full Superclassic/Classic or Superserver*. To install only the client files, choose *Minimum client install - no server, no tools* when going through the wizard steps.
- You have the following database connection details: server host name or IP address, database path (or alias) on the server, user name, and password.

Important: The platform of both the Firebird ODBC driver and client (32-bit or 64-bit) must correspond to that of UModel. Additionally, the version of the Firebird client must correspond to the version of Firebird server to which you are connecting.

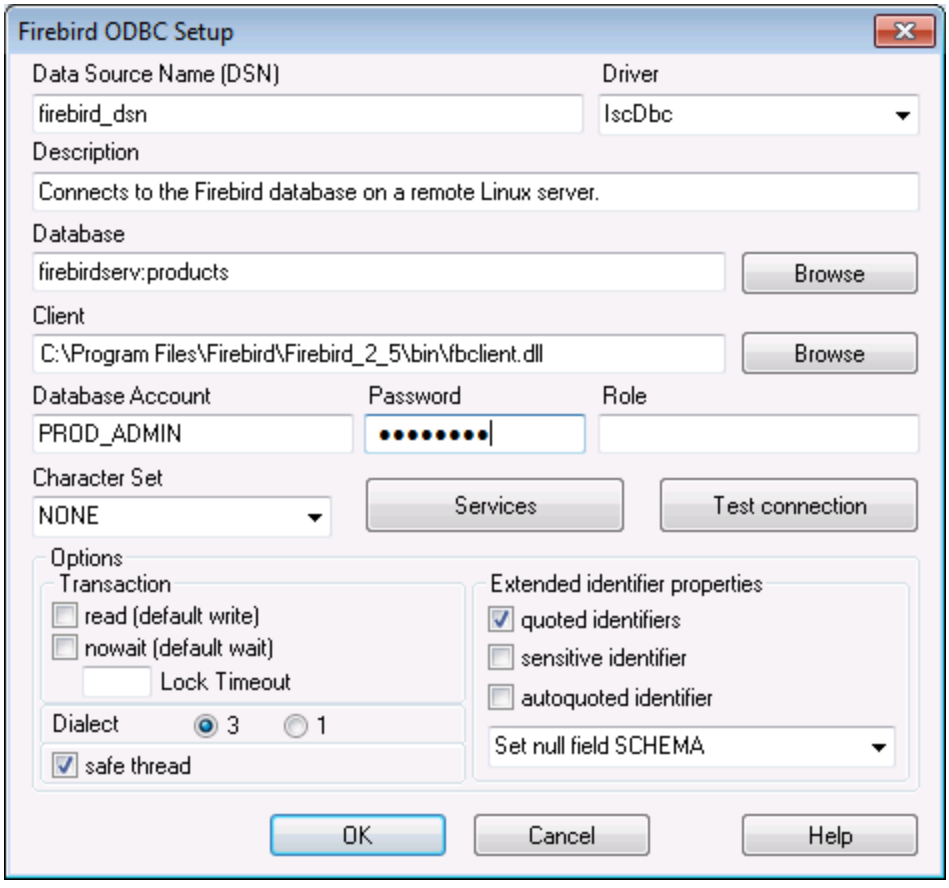
Connection

1. Start the database connection wizard[508] and click **ODBC Connections**.
2. Select *User DSN* (or *System DSN* if you have administrative privileges), and then click **Create a New DSN** .

© 2019-2025 Altova GmbH



3. Select the Firebird driver, and then click *User DSN* or *System DSN*, depending on what you selected in the previous step. If the Firebird driver is not available in the list, make sure that it is installed on your operating system (see also Viewing the Available ODBC Drivers 530).



4. Enter the database connection details as follows:

Data Source Name (DSN)	Enter a descriptive name for the data source you are creating.
Database	Enter the server host name or IP address, followed by a colon, followed by the database alias (or path). In this example, the host

	name is `firebirdserv`, and the database alias is `products`, as follows:
	
	`firebirdserv:products`
	
	Using a database alias assumes that, on the server side, the database administrator has configured the alias *products* to point to the actual Firebird (.fdb) database file on the server (see the Firebird documentation for more details).
	
	You can also use the server IP address instead of the host name, and a path instead of an alias; therefore, any of the following sample connection strings are valid:
	
	`firebirdserver:/var/Firebird/databases/butterflies.fdb`
	`127.0.0.1:D:\Misc\Lenders.fdb`
	
	If the database is on the local Windows machine, click **Browse** and select the Firebird (.fdb) database file directly.

Client	Enter the path to the **fbclient.dll** file. By default, this is the `bin` subdirectory of the Firebird installation directory.
Database Account	Enter the database user name supplied by the database administrator (in this example, `PROD_ADMIN`).
Password	Enter the database password supplied by the database administrator.

5. Click **OK** to finish.

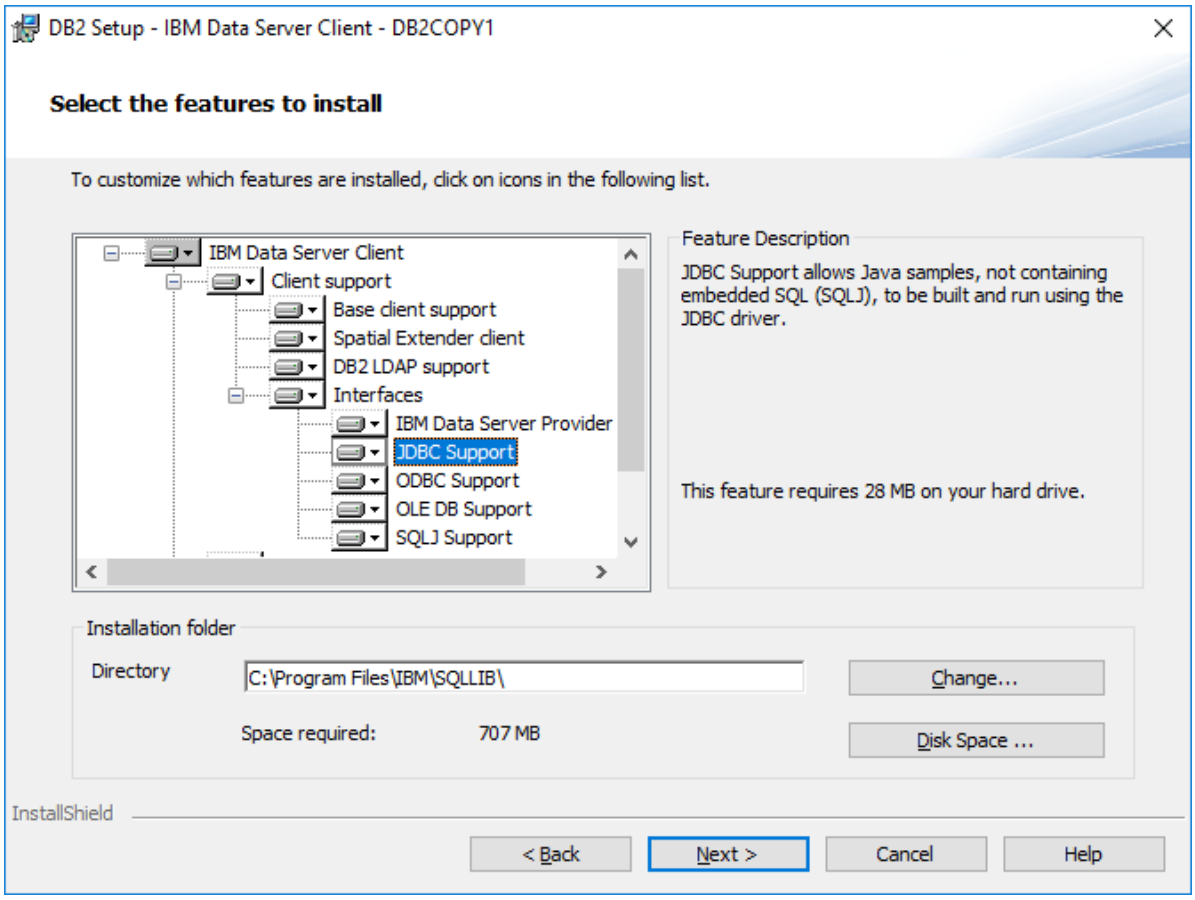
9.2.9.3 IBM DB2 (JDBC)

This example illustrates how to connect to an IBM DB2 database server via JDBC.

Prerequisites

- JRE (Java Runtime Environment) or Java Development Kit (JDK) must be installed. This may be either Oracle JDK or an open source build such as Oracle OpenJDK. UModel will determine the path to the Java Virtual Machine (JVM) from the following locations, in this order: (i) the custom JVM path you may have set in application **Options**; see Java Settings 694; (ii) the JVM path found in the Windows registry; (iii) the `JAVA_HOME` environment variable.
- Make sure that the platform of UModel (32-bit, 64-bit) matches that of the JRE/JDK.
- The JDBC driver (one or several .jar files that provide connectivity to the database) must be available on your operating system. This example uses the JDBC driver available after installing the IBM Data Server Client version 10.1 (64-bit). For the JDBC drivers to be installed, choose a *Typical* installation, or select this option explicitly on the installation wizard.

© 2019-2025 Altova GmbH



If you did not change the default installation path, the required `.jar` files will be in the `C:\Program Files\IBM\SQLLIB\java` directory after installation.

- You have the following database connection details: host, port, database path, name, or alias, username, and password.

Connection

1. Start the database connection wizard[508] and click **JDBC Connections** (*see JDBC Connection*[525] *for a screenshot of the dialog*).
2. In the *Classpaths* field, enter the path to the `.jar` file that provides connectivity to the database. If necessary, you can also enter a semicolon-separated list of `.jar` file paths. If you have added the filepath to the CLASSPATH[527] of the system, you can leave this field empty.
3. In the *Driver* field, select the appropriate driver. Relevant drivers will be available only if a valid `.jar` file path is in the *Classpaths* field or in the `CLASSPATH` environment variable.
4. Enter the username and password for the database in the corresponding fields.
5. Enter the JDBC connection string in the *Database URL* field according to the pattern in the table below, replacing the highlighted values with the ones applicable to your database server.
6. Click **Connect**.

Connection details of the IBM DB2 example

Field	Value

Classpaths	`C:\Program Files\IBM\SQLLIB\java\db2jcc.jar`
Driver	*com.ibm.db2.jcc.DB2Driver*
Database URL	`jdbc:db2://hostName:port/databaseName`

9.2.9.4 IBM DB2 (ODBC)

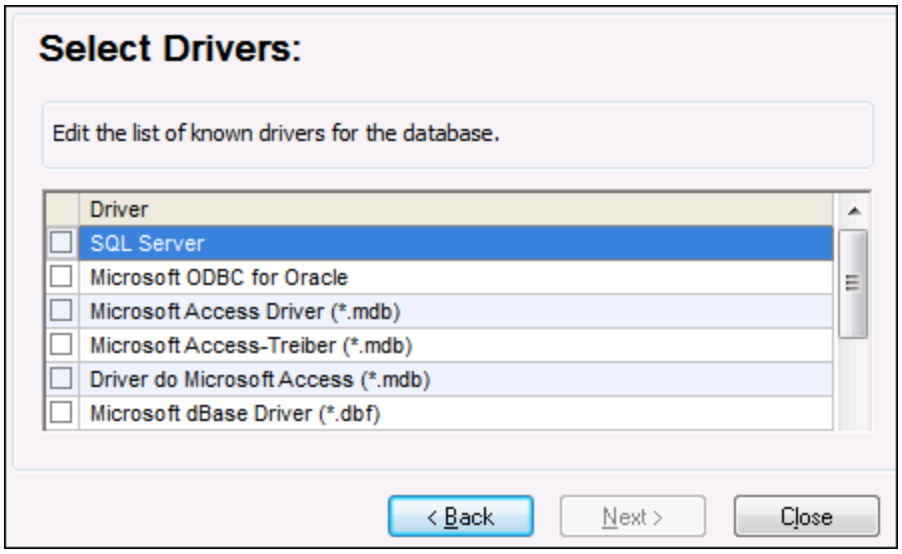
This example illustrates how to connect to an IBM DB2 database via ODBC.

Prerequisites

- IBM Data Server Client must be installed and configured on your operating system (this example uses IBM Data Server Client 9.7). For installation instructions, check the documentation supplied with your IBM DB2 software. After installing the IBM Data Server Client, check if the ODBC drivers are available on your machine (see Viewing the Available ODBC Drivers 530).
- Create a database alias. There are several ways to do this:
  - From IBM DB2 Configuration Assistant
  - From IBM DB2 Command Line Processor
  - From the ODBC data source wizard (for this case, the instructions are shown below)
- You have the following database connection details: host, database, port, username, and password.

Connection

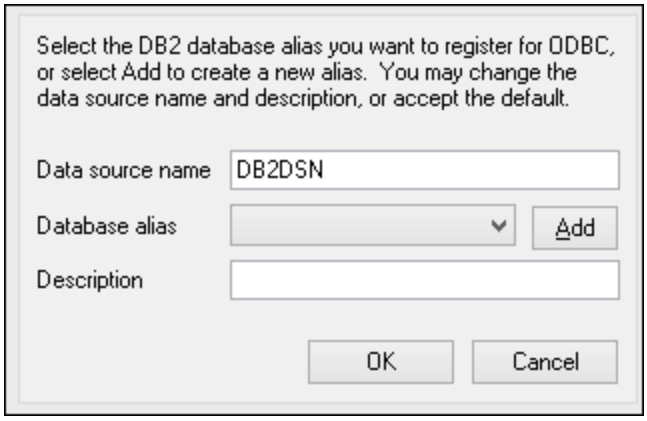
1. Start the database connection wizard 508, select **Connection Wizard** and then *IBM DB2 (ODBC/JDBC)*. Click **Next**.
2. Select *ODBC* and click **Next**.
3. If prompted to edit the list of known drivers for the database, select the database drivers applicable to IBM DB2 (see *Prerequisites* above) and click **Next**.



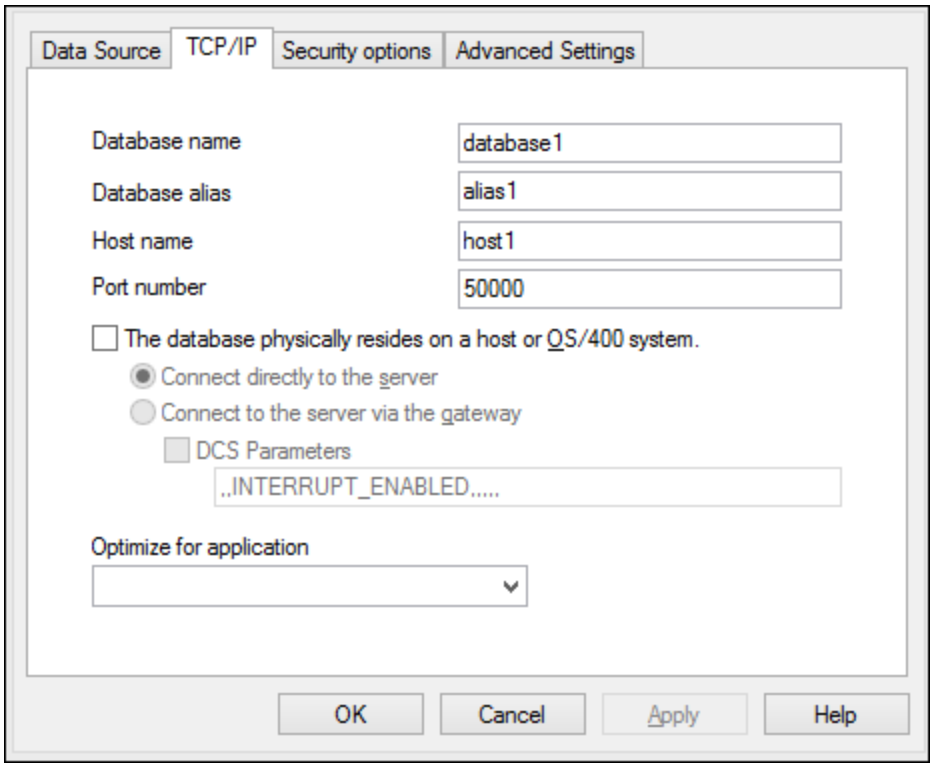
 © 2019-2025 Altova GmbH

4. Select the IBM DB2 driver you want from the list, and then click **Connect**. (To edit the list of available drivers, click **Edit Drivers**, and then check or uncheck the IBM DB2 drivers you wish to add or remove, respectively.)

5. Enter a data source name (in the screenshot below, DB2DSN), and then click **Add**.



6. On the *Data Source* tab, enter the user name and password for the database.
7. On the *TCP/IP* tab, enter the database name, a name for the alias, the host name, and the port number, and then click **OK**.



8. Enter the username and password again, and then click **OK**.

9.2.9.5 IBM DB2 for i (JDBC)

This example illustrates how to connect to an IBM DB2 for i database server via JDBC.

Prerequisites

- JRE (Java Runtime Environment) or Java Development Kit (JDK) must be installed. This may be either Oracle JDK or an open source build such as Oracle OpenJDK. UModel will determine the path to the Java Virtual Machine (JVM) from the following locations, in this order: (i) the custom JVM path you may have set in application **Options**; see Java Settings 694; (ii) the JVM path found in the Windows registry; (iii) the `JAVA_HOME` environment variable.
- Make sure that the platform of UModel (32-bit, 64-bit) matches that of the JRE/JDK.
- The JDBC driver (one or several .jar files that provide connectivity to the database) must be available on your operating system. This example uses the open source **Toolbox for Java/JTOpen** version 9.8 (http://jt400.sourceforge.net/). After you download the package and unpack to a local directory, the required .jar files will be available in the **lib** subdirectory.
- You have the following database connection details: host, port, database path, name, or alias, username, and password.

Connection

1. Start the database connection wizard 508 and click **JDBC Connections** (*see JDBC Connection* 525 *for a screenshot of the dialog*).

© 2019-2025 Altova GmbH

2. In the *Classpaths* field, enter the path to the `.jar` file that provides connectivity to the database. If necessary, you can also enter a semicolon-separated list of `.jar` file paths. If you have added the filepath to the CLASSPATH[527] of the system, you can leave this field empty.
3. In the *Driver* field, select the appropriate driver. Relevant drivers will be available only if a valid `.jar` file path is in the *Classpaths* field or in the `CLASSPATH` environment variable.
4. Enter the username and password for the database in the corresponding fields.
5. Enter the JDBC connection string in the *Database URL* field according to the pattern in the table below, replacing the highlighted values with the ones applicable to your database server.
6. Click **Connect**.

Connection details of the IBM DB2 for i example

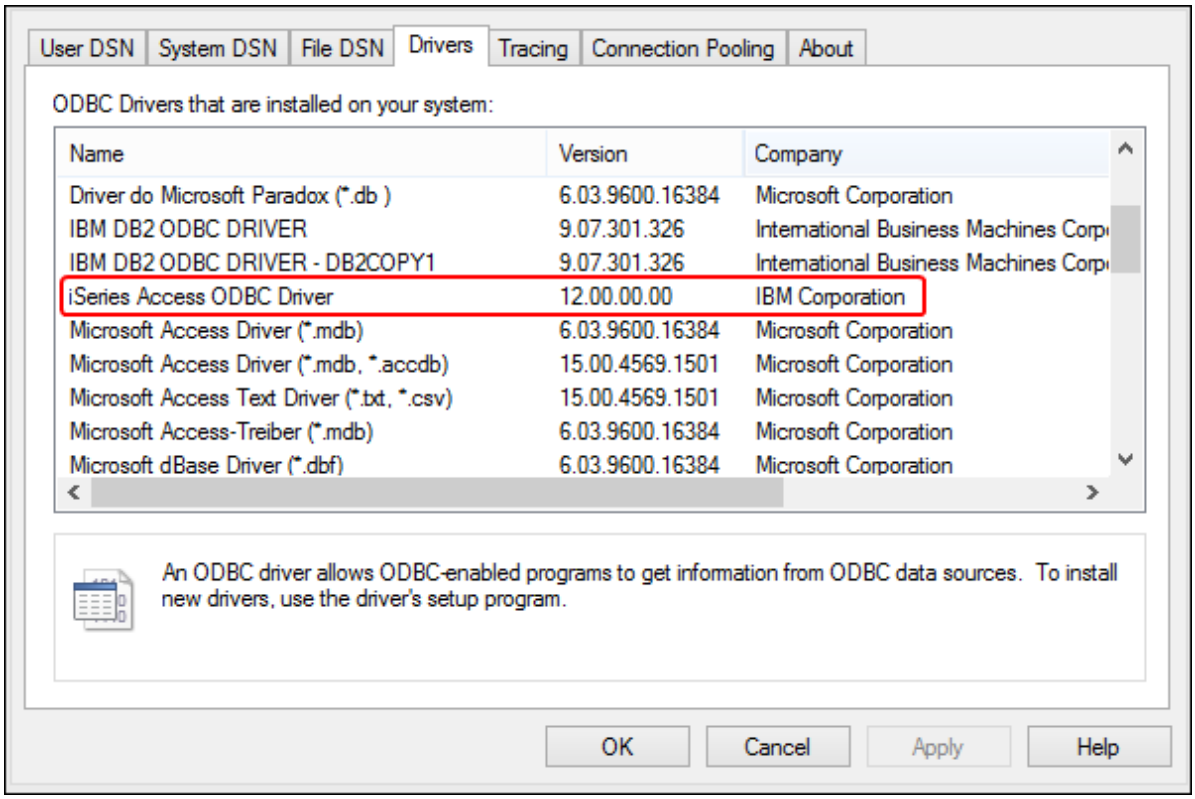
Field	Value
Classpaths	`C:\jdbc\jtopen_9_8\jt400.jar`
Driver	*com.ibm.as400.access.AS400JDBCDriver*
Database URL	`jdbc:as400://host[:<port>]/<database path or alias>`

9.2.9.6 IBM DB2 for i (ODBC)

This example illustrates how to connect to an *IBM DB2 for i* database via ODBC.

Prerequisites

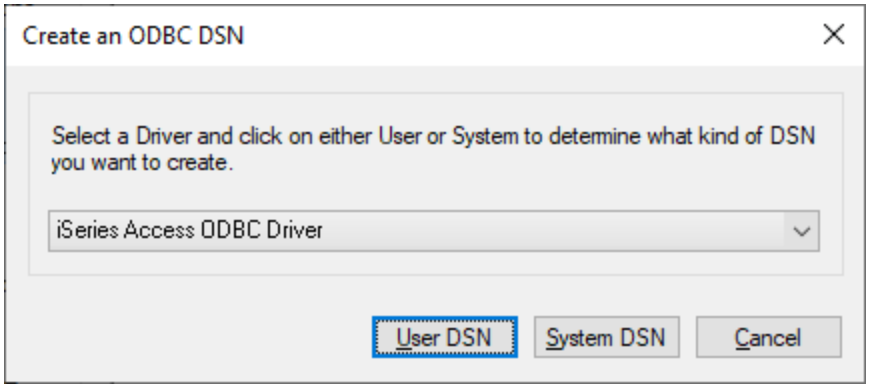
- *IBM System i Access for Windows* must be installed on your operating system. For installation instructions, check the documentation supplied with your *IBM DB2 for i* software. After installation, check if the ODBC driver is available on your machine[530].



- You have the following database connection details: the IP address of the database server, and the database user name and password.
- Run System i Navigator and follow the wizard to create a new connection. When prompted to specify a system, enter the IP address of the database server. After creating the connection, verify the connection by click it and selecting **File | Diagnostics | Verify Connection**.

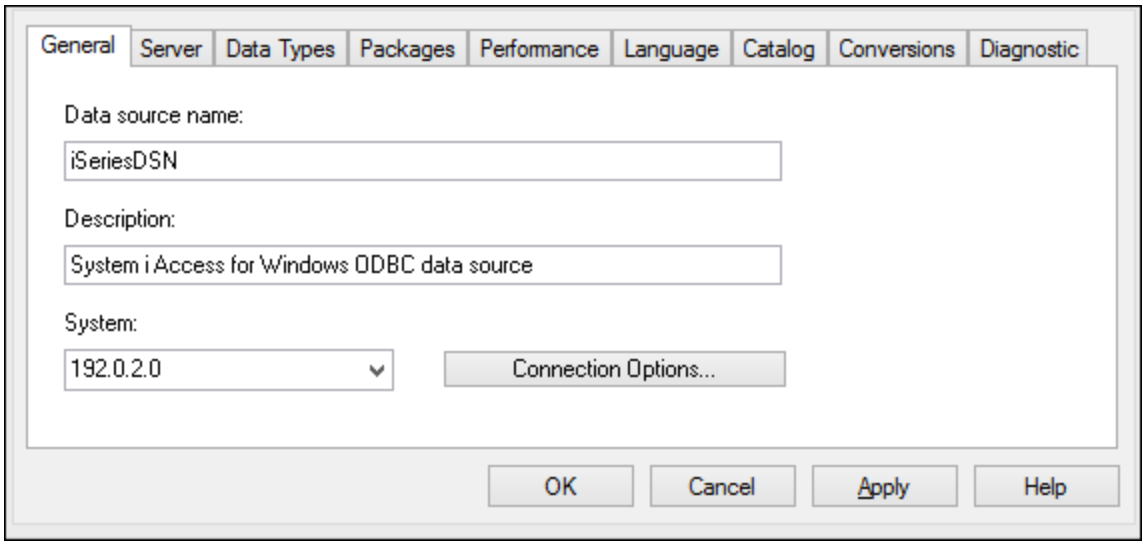
Connection

1. Start the database connection wizard[508] and click **ODBC Connections**.
2. Click *User DSN*. and click **Create a New DSN** .
3. Select *iSeries Access ODBC Driver* from the list, and click **User DSN** (or **System DSN** if applicable).

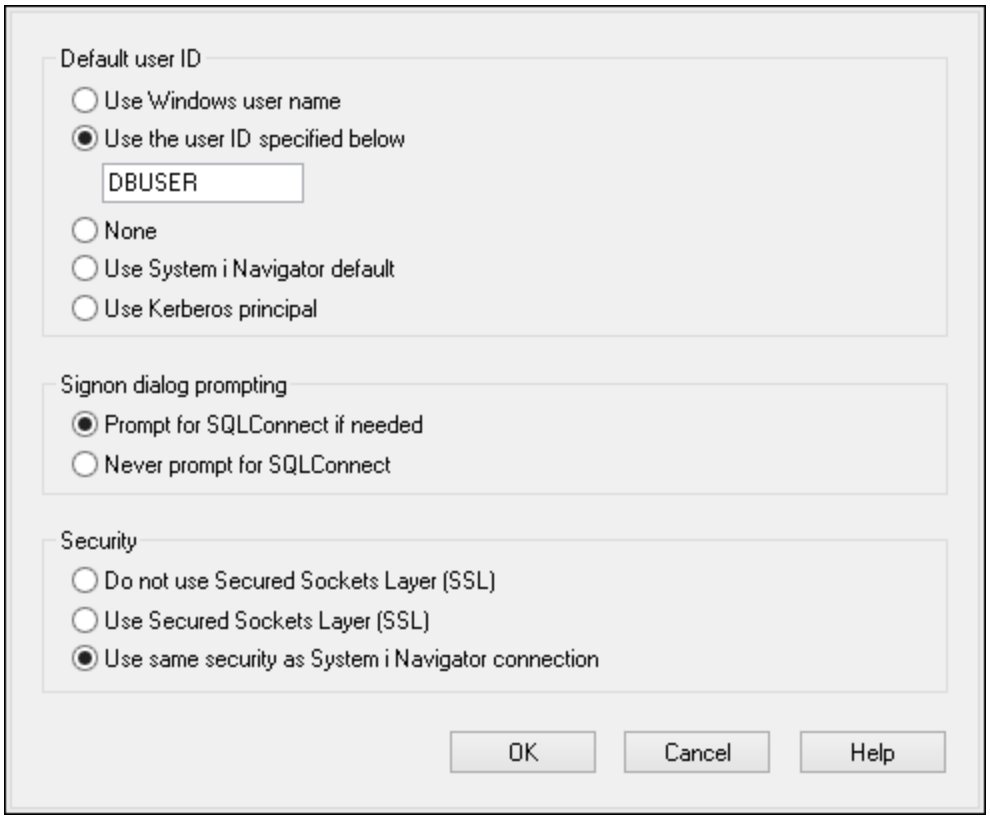


4. Enter a data source name and select the connection from the System combo box. In this example, the data source name is iSeriesDSN and the System is 192.0.2.0.

© 2019-2025 Altova GmbH



5. Click **Connection Options**, select *Use the User ID Specified Below* and enter the name of the database user (in this example, *DBUSER*).



6. Click **OK**. The new data source becomes available in the list of DSNs.
7. Click **Connect**.
8. Enter the user name and password to the database when prompted, and then click **OK**.

9.2.9.7 IBM Informix (JDBC)

This example illustrates how to connect to an IBM Informix database server via JDBC.

Prerequisites

- JRE (Java Runtime Environment) or Java Development Kit (JDK) must be installed. This may be either Oracle JDK or an open source build such as Oracle OpenJDK. UModel will determine the path to the Java Virtual Machine (JVM) from the following locations, in this order: (i) the custom JVM path you may have set in application **Options**; see Java Settings 694; (ii) the JVM path found in the Windows registry; (iii) the `JAVA_HOME` environment variable.
- Make sure that the platform of UModel (32-bit, 64-bit) matches that of the JRE/JDK.
- The JDBC driver (one or several .jar files that provide connectivity to the database) must be available on your operating system. In this example, IBM Informix JDBC driver version 3.70 is used. For the driver's installation instructions, see the documentation accompanying the driver or the "IBM Informix JDBC Driver Programmer's Guide").
- You have the following database connection details: host, name of the Informix server, database, port, username, and password.

Connection

1. Start the database connection wizard 508 and click **JDBC Connections** (*see JDBC Connection 525 for a screenshot of the dialog*).
2. In the *Classpaths* field, enter the path to the `.jar` file that provides connectivity to the database. If necessary, you can also enter a semicolon-separated list of `.jar` file paths. If you have added the filepath to the CLASSPATH 527 of the system, you can leave this field empty.
3. In the *Driver* field, select the appropriate driver. Relevant drivers will be available only if a valid `.jar` file path is in the *Classpaths* field or in the `CLASSPATH` environment variable.
4. Enter the username and password for the database in the corresponding fields.
5. Enter the JDBC connection string in the *Database URL* field according to the pattern in the table below, replacing the highlighted values with the ones applicable to your database server.
6. Click **Connect**.

Connection details of the IBM Informix example

Field	Value
Classpaths	`C:\Informix_JDBC_Driver\lib\ifxjdbc.jar`
Driver	*com.informix.jdbc.IfxDriver*
Database URL	`jdbc:informix-sqli://hostName:port/databaseName:INFORMIXSERVER=myserver;`

© 2019-2025 Altova GmbH

9.2.9.8 MariaDB (ODBC)

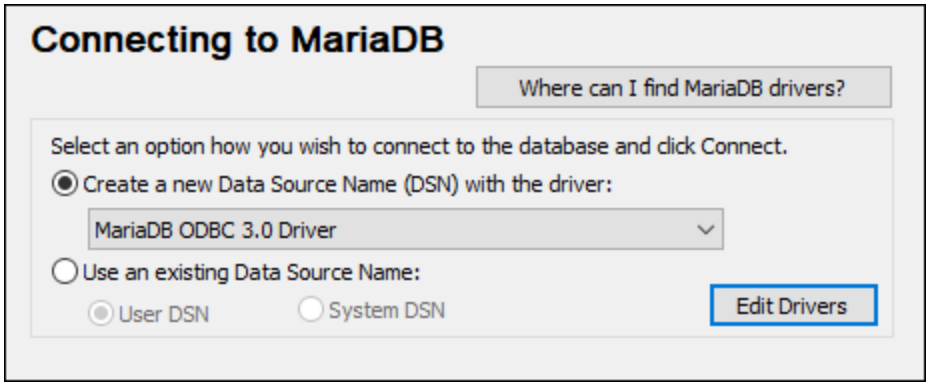
This example illustrates how to connect to a MariaDB database server via ODBC.

Prerequisites

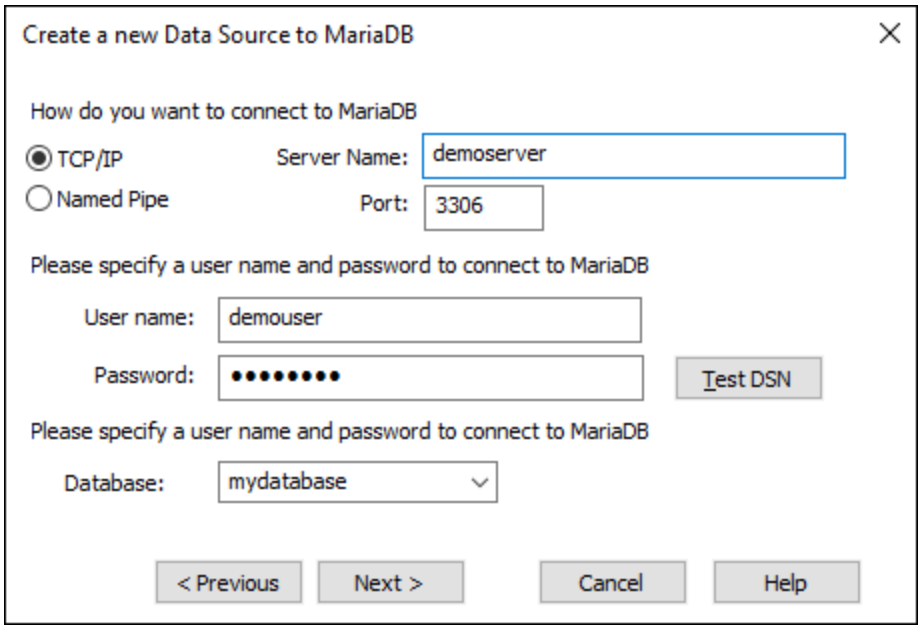
- The MariaDB ODBC Connector must be installed.
- You have the following database connection details: host, database, port, username, and password.

Connection

1. Start the database connection wizard[508], select **Connection Wizard** and then *MariaDB (ODBC/Native)*. Click **Next**.
2. Select *Create a new Data Source Name (DSN) with the driver*, and choose *MariaDB ODBC 3.0 Driver*. (If no such driver is available in the list, click **Edit Drivers**, and select any available MariaDB driver from the the list of ODBC drivers installed on your system.) Click **Connect**. to start the New Data Source Wizard.



3. In the first screen of the wizard, enter a name for the data source, optionally, a description to help you identify this ODBC data source in the future, and click **OK**.
4. In the next screen (screenshot below), fill in the database connection credentials (TCP/IP Server, User Name, and Password), select a database, and click **Test DSN**.



5. Upon successful connection, a message to that effect is displayed. Click **Next** and complete the wizard. Other parameters may be required, for example, SSL certificates if you are connecting to MariaDB via a secure connection.

Note: If the database server is remote, it must be configured by the server administrator to accept remote connections from your machine's IP address.

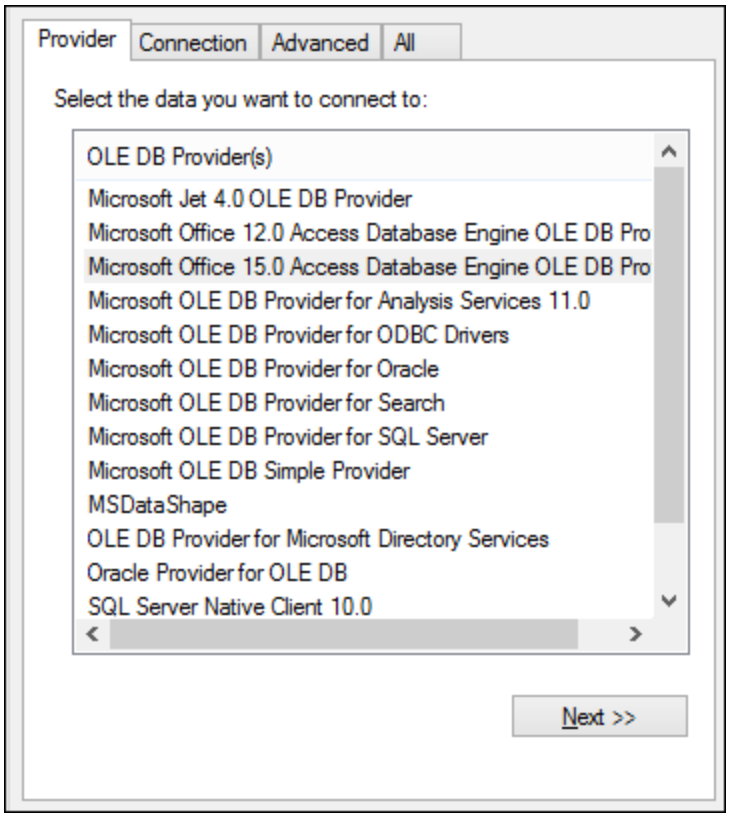
9.2.9.9 Microsoft Access (ADO)

A simple way to connect to a Microsoft Access database is to follow the wizard and browse for the database file, as shown in Connecting to an Existing Microsoft Access Database[517]. An alternative approach is to set up an ADO connection explicitly, as shown in this topic. This approach is useful if your database is password-protected. (It is also possible to connect to Microsoft Access through an ODBC connection, but it has limitations, so it is best to avoid it.)

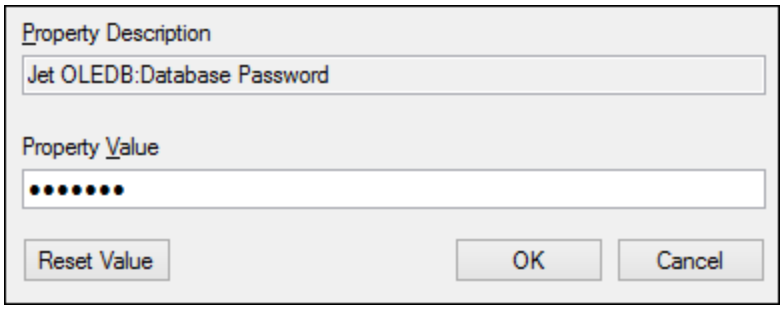
Connection

1. Start the database connection wizard[508].
2. Click **ADO Connections**, then click **Build**.
3. In the dialog that appears (*screenshot below*), go to the *Provider* tab, select *Microsoft Office 15.0 Access Database Engine OLE DB Provider*, and then click **Next**.

© 2019-2025 Altova GmbH



4. Go to the dialog's *Connection* tab, and in the *Data Source* field, enter the path to the Microsoft Access file in UNC format, for example, `\\myserver\\mynetworkshare\Reports\Revenue.accdb`, where `myserver` is the name of the server and `mynetworkshare` is the name of the network share.
5. On the *All* tab, double click the *Jet OLEDB:Database Password* property and enter the database password as property value (*see screenshot below*)



If you are still unable to connect, locate the workgroup information file (`System.MDW`) applicable to your user profile, and set the value of the *Jet OLEDB: System database* property to the path of the `System.MDW` file.

9.2.9.10 Microsoft Azure SQL (ODBC)

In order to connect properly to an Azure SQL database, you must install the latest SQL Server Native Client.

For information about connecting to an Azure SQL database in the cloud, see this Altova blog entry.

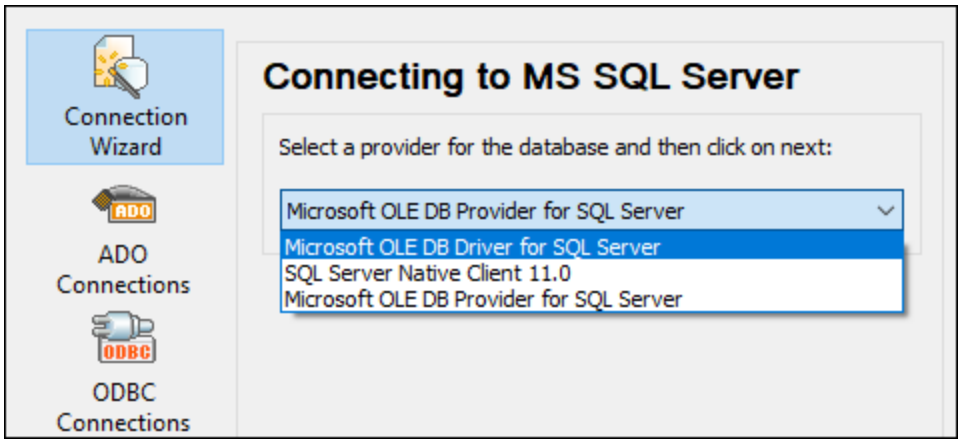
9.2.9.11 Microsoft SQL Server (ADO)

This example illustrates how to connect to a SQL Server database through ADO. These instructions work with the recommended Microsoft OLE DB Driver for SQL Server (MSOLEDBSQL) that matches your UModel platform (32-bit or 64-bit). For other ADO providers, the instructions would be similar but you may need to set additional connection properties as described in Setting up the SQL Server Data Link Properties [517].

Note: The *Microsoft OLE DB Provider for SQL Server (SQLOLEDB)* is known to have issues with parameter binding of complex queries like Common Table Expressions (CTE) and nested SELECT statements.

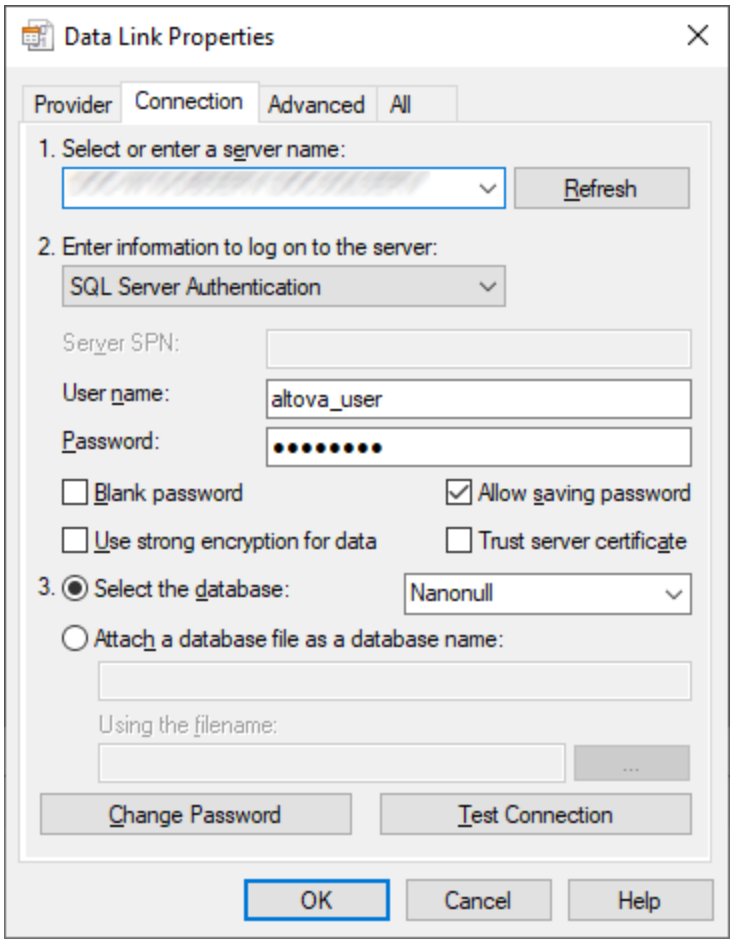
Connection

1. Start the database connection wizard [508], select **Connection Wizard** and then *Microsoft SQL Server (ADO).* Click **Next**.
2. A list of available ADO providers is displayed (*screenshot below*). In this example, we use *Microsoft OLE DB Driver for SQL Server*. If it's not in the list, make sure that it is installed on your computer.



3. Click **Next**. The Data Link Properties dialog appears (*screenshot below*).

© 2019-2025 Altova GmbH



4. In the *Connection* tab, select or enter the name of the database server, for example, *SQLSERV01*. If you are connecting to a named SQL Server instance, the server name will be something like *SQLSERV01\SOMEINSTANCE*.
5. If the database server was configured to allow connections from users authenticated on the Windows domain, select Windows Authentication. Otherwise, select SQL Server Authentication, clear the Blank password check box, and enter the database credentials in the relevant boxes.
6. Select the *Allow saving password* check box and the database to which you are connecting (in this example, *Nanonull*).
7. You can test the connection by clicking **Test Connection**.
8. Click **OK** when done.

9.2.9.12 Microsoft SQL Server (ODBC)

This example illustrates how to connect to a SQL Server database via ODBC.

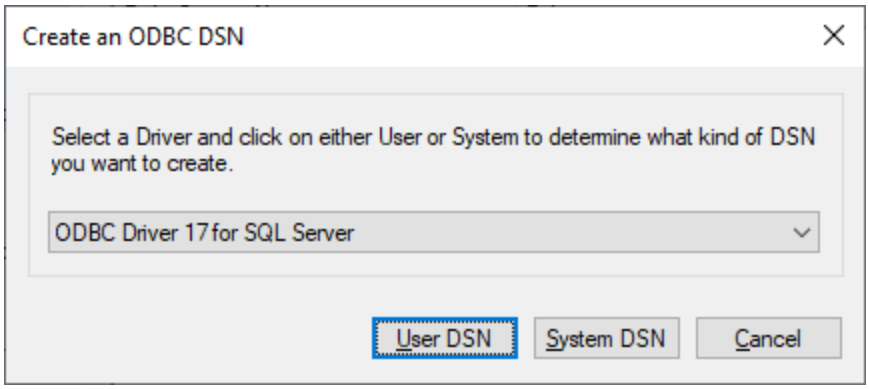
Prerequisites

Download and install the Microsoft ODBC Driver for SQL Server from the Microsoft website. This example uses Microsoft ODBC Driver 17 for SQL Server to connect to a SQL Server 2016 database. You might need a

different ODBC driver version, depending on your SQL Server version. For information about compatibility, see the driver's system requirements.

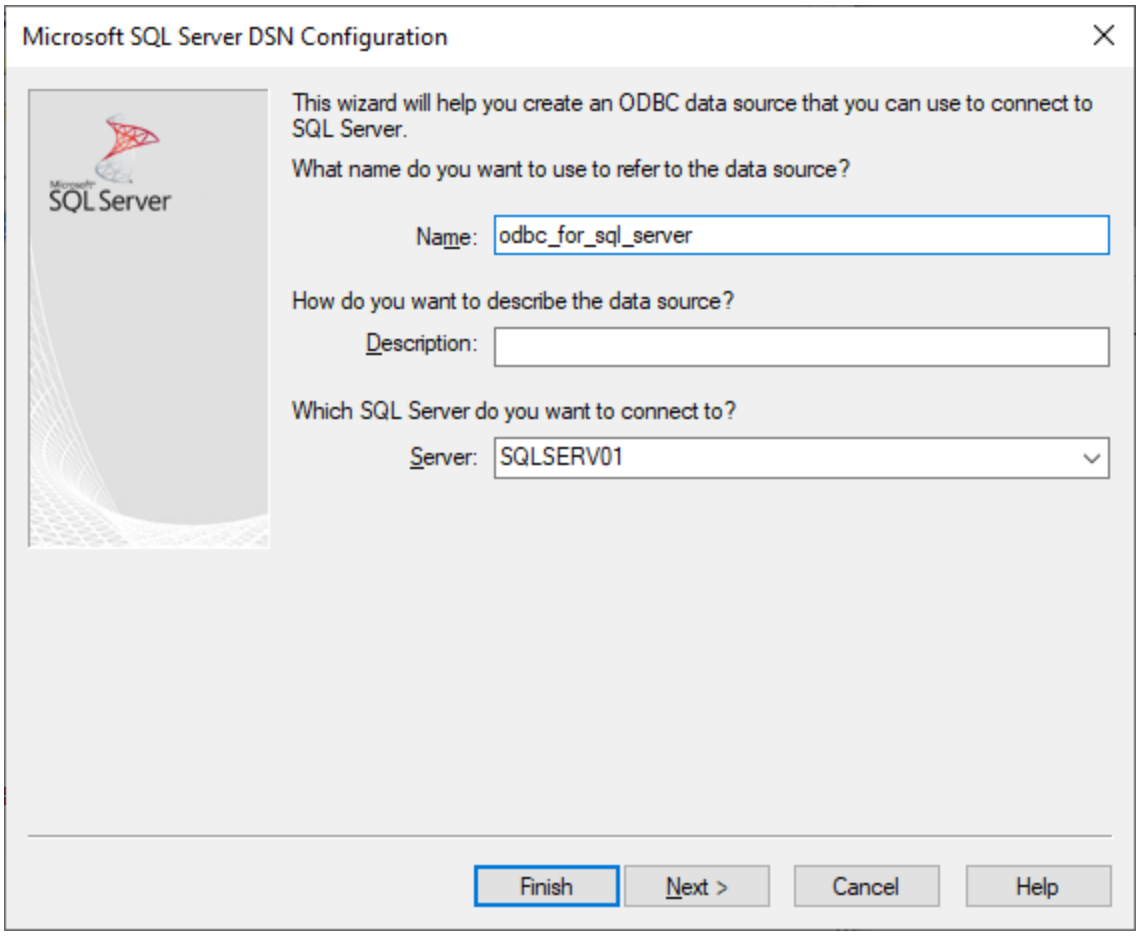
Connection

1. Start the database connection wizard[508] and click **ODBC Connections**.
2. Select *User DSN* (or *System DSN* if you have administrative privileges), and then click **Create a New DSN** .
3. Select the driver from the list. Note that the driver appears in the list only after it has been installed.

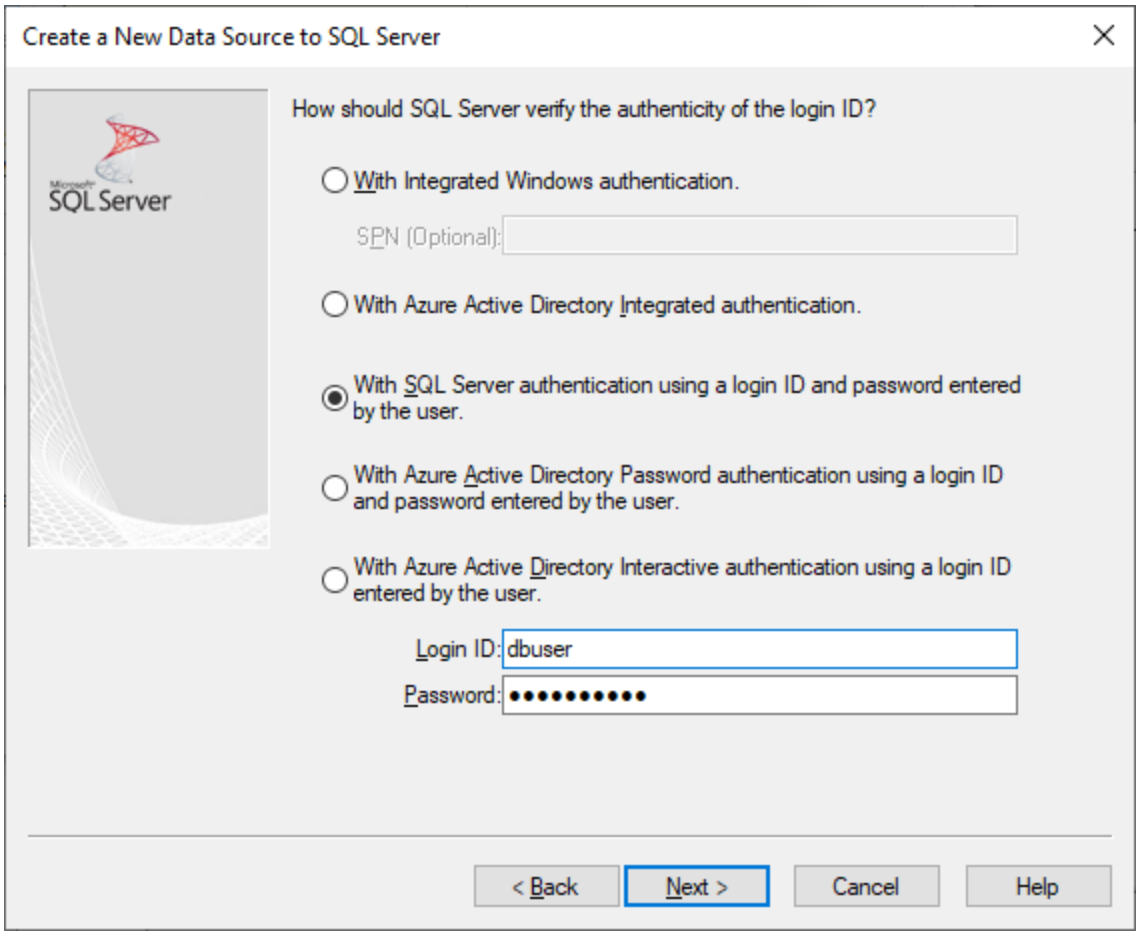


4. Click *User DSN* (or *System DSN* if you are creating a System DSN). Creating a System DSN requires that UModel be run as an administrator. Therefore, in order to create a System DSN, cancel the wizard, make sure that you run UModel as an administrator, and perform the steps above again.
5. Enter a name for the data source, enter, optionally, a description to identify this connection, and select from the list the SQL Server to which you are connecting. Click **Next** to go to the next screen.

© 2019-2025 Altova GmbH

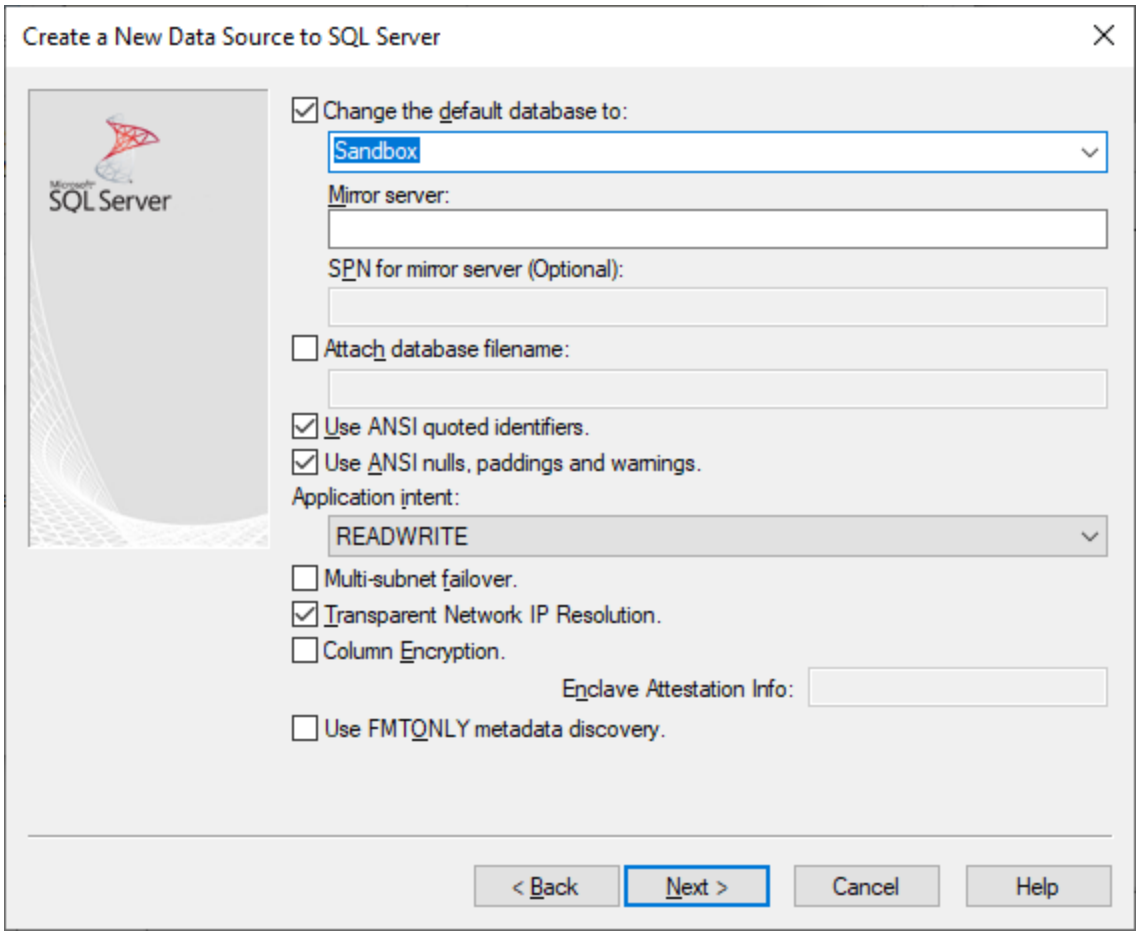


6. In the next screen, select the authentication method to use. If the database server was configured to allow connections from users authenticated on the Windows domain, select *With Integrated Windows authentication*. Otherwise, select one of the other options, as applicable. This example uses *With SQL Server Authentication*, which requires that the user name and password be entered in the relevant boxes. Click **Next** when done.

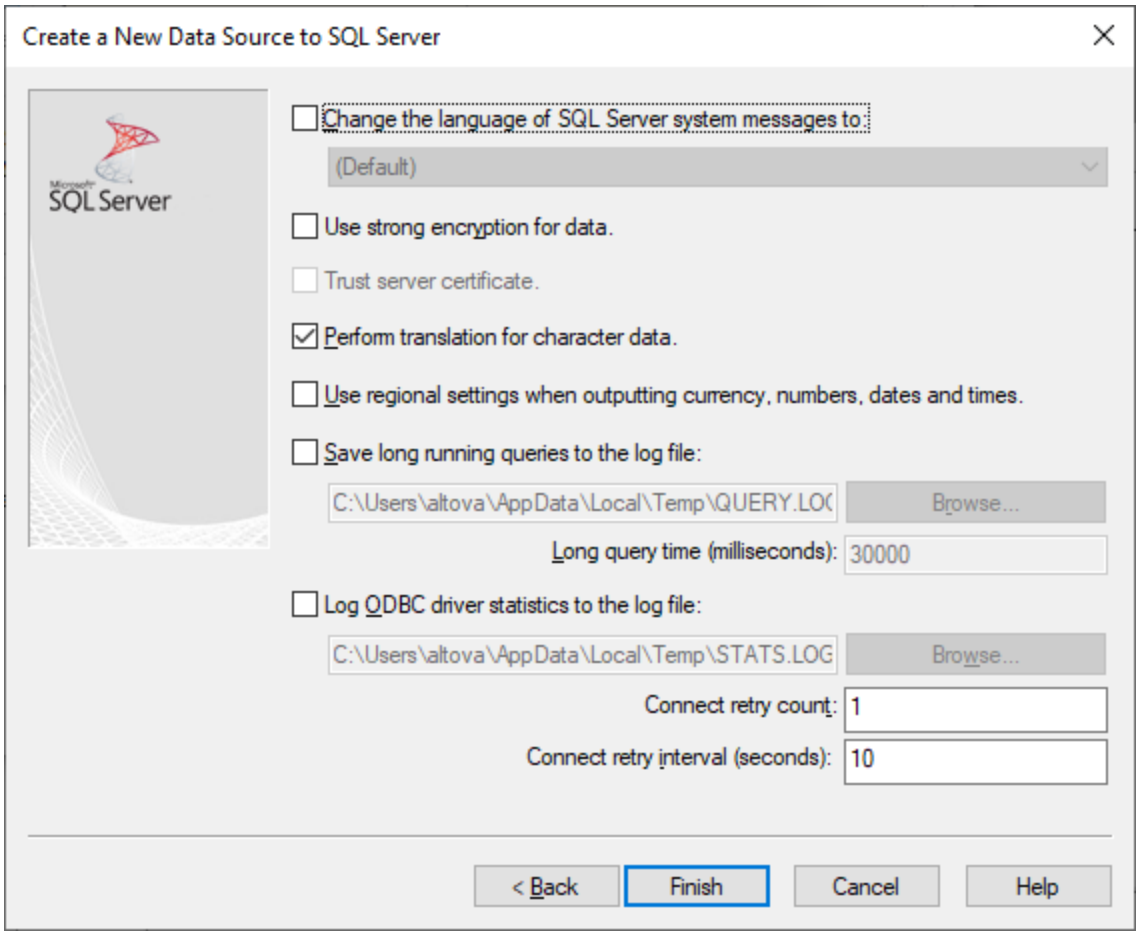


7. In the next screen of our example (*screenshot below*), we have selected *Change the default database* and entered *Sandbox* (the name of the database to which to connect). Click **Next** after you have entered the settings you want.

© 2019-2025 Altova GmbH



8. Click **Next** and, optionally, configure additional parameters for this connection. Click **Finish** when done.



9. A confirmation dialog box listing the connection details opens. Click **OK**.
10. The data source now appears in the list of *User* or *System* data sources.

9.2.9.13 MySQL (ODBC)

This example illustrates how to connect to a MySQL database server from a Windows machine via the MySQL ODBC, which driver must be downloaded and installed separately. This example uses MySQL Connector/ODBC 8.0.

Prerequisites

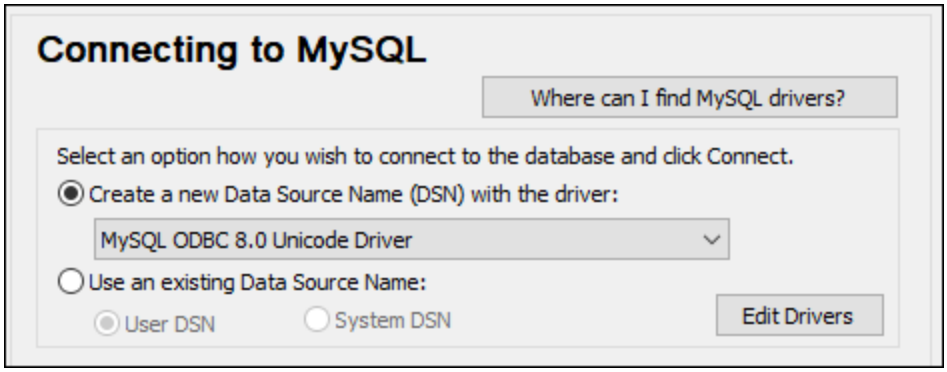
- MySQL ODBC driver must be installed on your operating system. Check the MySQL documentation for the driver version recommended for your database server version.
- Make sure that the platform of UModel (32-bit, 64-bit) matches that of the MySQL ODBC driver.
- You have the following database connection details: host, database, port, username, and password.

Connection

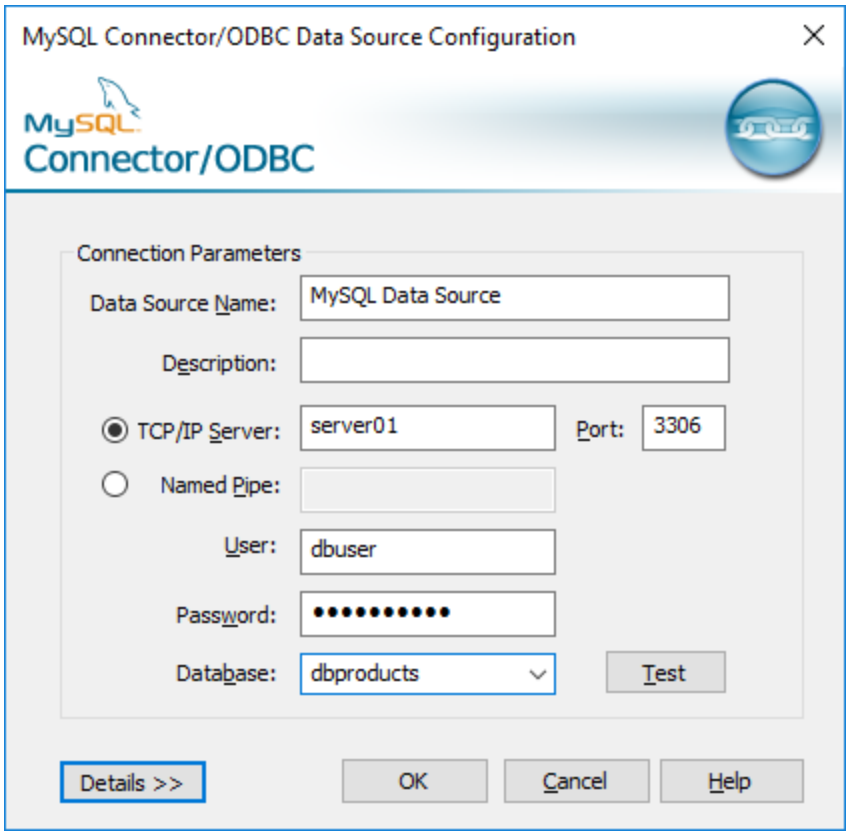
1. Start the database connection wizard[508], select **Connection Wizard** and then *MySQL (ODBC/Native)*. Click **Next**.

© 2019-2025 Altova GmbH

2. Select *Create a new Data Source Name (DSN) with the driver*, and select a MySQL driver. (If no such driver is available in the list, click **Edit Drivers**, and select any available MySQL driver from the the list of ODBC drivers installed on your system. See also Viewing the Available ODBC Drivers 530.)



3. Click **Connect**. to open the Data Source Configuration dialog (*screenshot below*).



4. In the Data Source Name box, enter a descriptive name that will help you identify this ODBC data source in future. Optionally enter a description.
5. Fill in the database connection credentials (TCP/IP Server, User, Password), select a database, and then click **OK**.

Note: If the database server is remote, it must be configured by the server administrator to accept remote connections from your machine's IP address. Also, if you click **Details >>**, there are several additional parameters available for configuration. Check the driver's documentation before changing the default values.

9.2.9.14 Oracle (JDBC)

This example shows how to connect to an Oracle database server using the JDBC interface. The connection is created as a pure Java connection, using the Oracle Instant Client Package (Basic) available from the Oracle website. The advantage of this connection type is that it requires only the Java environment and the .jar libraries supplied by the Oracle Instant Client Package.

Prerequisites

- JRE (Java Runtime Environment) or Java Development Kit (JDK) must be installed. This may be either Oracle JDK or an open source build such as Oracle OpenJDK. UModel will determine the path to the Java Virtual Machine (JVM) from the following locations, in this order: (i) the custom JVM path you may have set in application **Options**; see Java Settings 694; (ii) the JVM path found in the Windows registry; (iii) the `JAVA_HOME` environment variable.
- Make sure that the platform of UModel (32-bit, 64-bit) matches that of the JRE/JDK.
- The Oracle Instant Client Package (Basic) must be available on your operating system. The package can be downloaded from the official Oracle website. This example uses Oracle Instant Client Package version 12.1.0.2.0, for Windows 32-bit and, consequently, Oracle JDK 32-bit.
- You have the following database connection details: host, port, service name, username, and password.

Connection

1. Start the database connection wizard 508 and click **JDBC Connections** (*see JDBC Connection 525 for a screenshot of the dialog*).
2. In the *Classpaths* field, enter the path to the `.jar` file that provides connectivity to the database. If necessary, you can also enter a semicolon-separated list of `.jar` file paths. If you have added the filepath to the CLASSPATH 527 of the system, you can leave this field empty.
3. In the *Driver* field, select the appropriate driver. Relevant drivers will be available only if a valid `.jar` file path is in the *Classpaths* field or in the `CLASSPATH` environment variable.
4. Enter the username and password for the database in the corresponding fields.
5. Enter the JDBC connection string in the *Database URL* field according to the pattern in the table below, replacing the highlighted values with the ones applicable to your database server.
6. Click **Connect**.

Connection details of the Oracle example

Field	Value
Classpaths	`C:\jdbc\instantclient_12_1\ojdbc7.jar`
Driver	*oracle.jdbc.OracleDriver* or *oracle.jdbcdriver.OracleDriver*
Database URL	`jdbc:oracle:thin:@//`**`host`**`:`**`port`**`:`**`service`**

© 2019-2025 Altova GmbH

9.2.9.15 Oracle (ODBC)

This example shows how to connect from UModel to an Oracle database server on a network machine, via an Oracle database client installed on the local operating system.

The example includes instructions for setting up an ODBC data source (DSN) using the database connection wizard in UModel. If you have already created a DSN, or if you prefer to create it directly from the *ODBC Data Source Administrator* in Windows, you can do so, and then select it when prompted by the wizard. For more information about ODBC data sources, see Setting up an ODBC Connection 528.

Prerequisites

- The Oracle database client (which includes the ODBC Oracle driver) must be installed and configured on your system. For instructions, see the Oracle documentation.
- The **tnsnames.ora** file located in Oracle home directory contains an entry that describes the database connection parameters, in a format similar to this:

```
ORCL =
  (DESCRIPTION =
    (ADDRESS_LIST =
      (ADDRESS = (PROTOCOL = TCP)(HOST = server01)(PORT = 1521))
    )
    (CONNECT_DATA =
      (SID = orcl)
      (SERVER = DEDICATED)
    )
  )
```

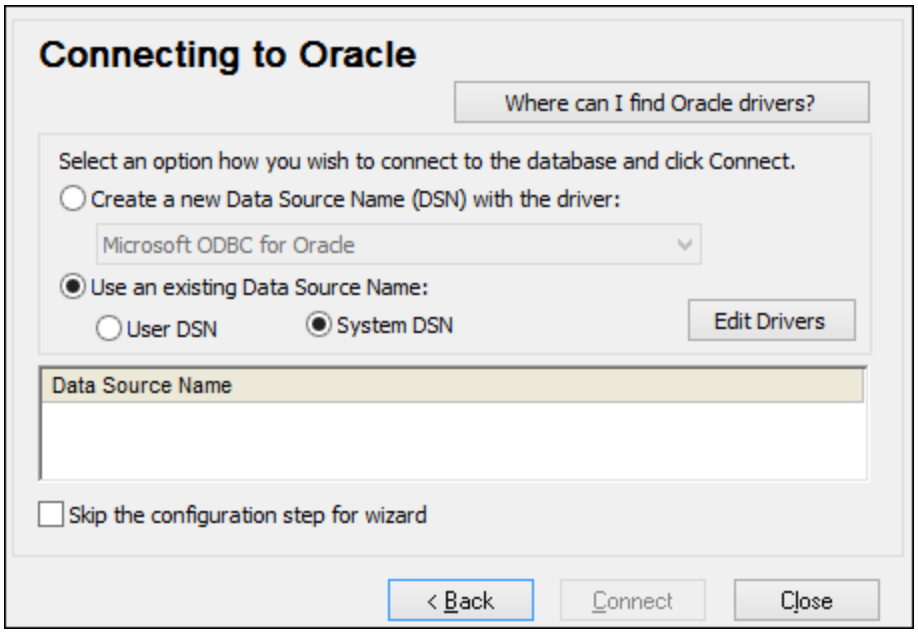
  For Oracle database client 11.2.0, the default Oracle home directory path could be as follows:

```
C:\app\username\product\11.2.0\client_1\network\admin\tnsnames.ora
```

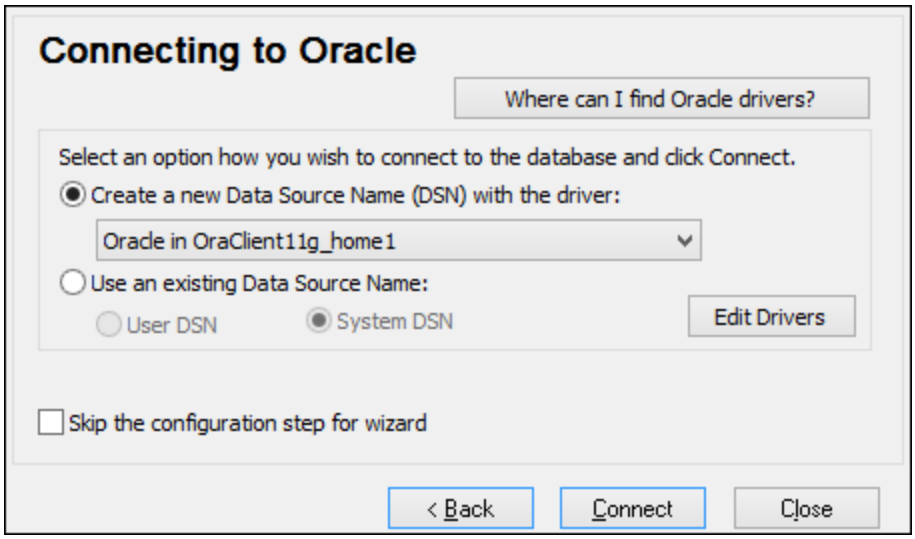
  You can add new entries to **tnsnames.ora**, either by pasting the connection details and saving the file, or by running the Oracle Net Configuration Assistant, if available. If you want these values to appear in dropdown lists during the configuration process, then you may need to add the path to the admin folder as a **TNS_ADMIN** environment variable.

Connection

1. Start the database connection wizard 508, select **Connection Wizard** and then *Oracle (ODBC/JDBC)*. Click **Next**.
2. Select *ODBC* and click **Next**.
3. In the dialog that appears (*screenshot below*), click **Edit Drivers**.

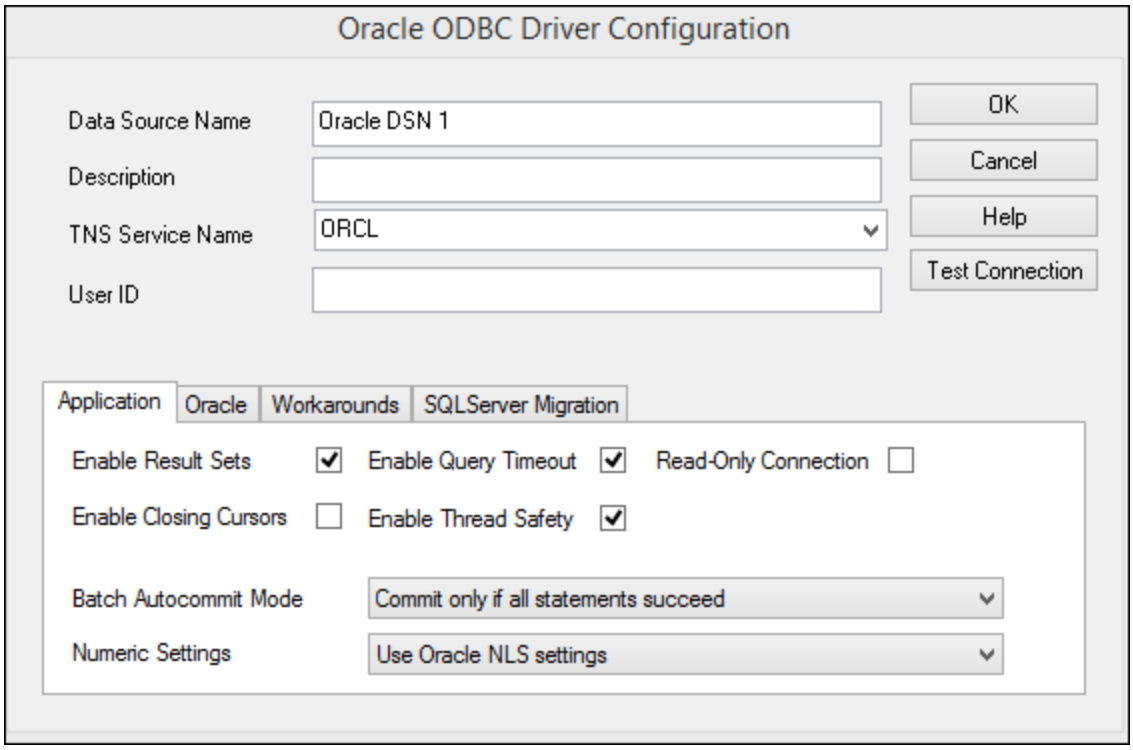


4. From the list of Oracle drivers available on your system, select the Oracle driver you wish to use and click **Back**.
5. Select *Create a new data source name (DSN) with the driver*, and then select the Oracle driver chosen in step 4. (Avoid using the Microsoft-supplied driver called Microsoft ODBC for Oracle driver. Microsoft recommends using the ODBC driver provided by Oracle.)



6. Click **Connect**.
7. In the dialog that appears fill in the required fields (*shown filled in the screenshot below*). The connection name must be entered in the *TNS Service Name* field as it is defined in the `tnsnames.ora` file (see *Prerequisites* above). *Note:* If you wish to have the dropdown list of the combo box automatically filled with the values of the **`tnsnames.ora`** file, then you may need to add the path to the admin folder as a **`TNS_ADMIN`** environment variable.

© 2019-2025 Altova GmbH



8. Click **OK** when done.
9. In the dialog that appears, enter the username and password to the database, and then click **OK**.

9.2.9.16 PostgreSQL (ODBC)

This example shows how to connect to a PostgreSQL database server from a Windows machine via a PostgreSQL ODBC driver installed on the local machine. This example uses the *psqlODBC* driver (version 11.0) downloaded from the official website (see also Database Drivers Overview[510]).

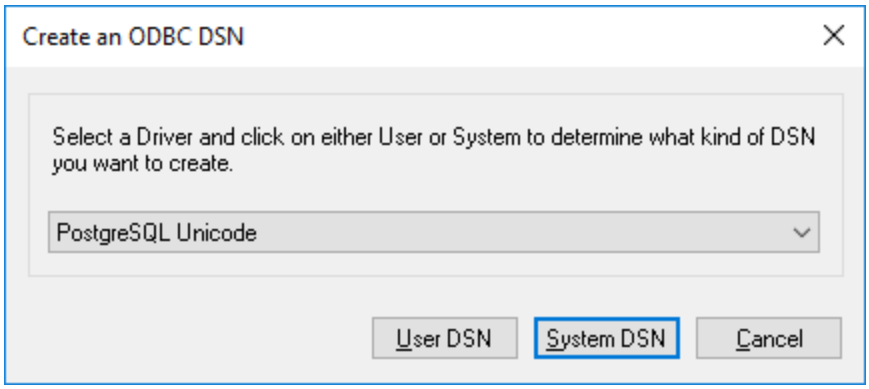
Note: You can also connect to a PostgreSQL database server directly (without the ODBC driver), see Native Connections[532].

Prerequisites

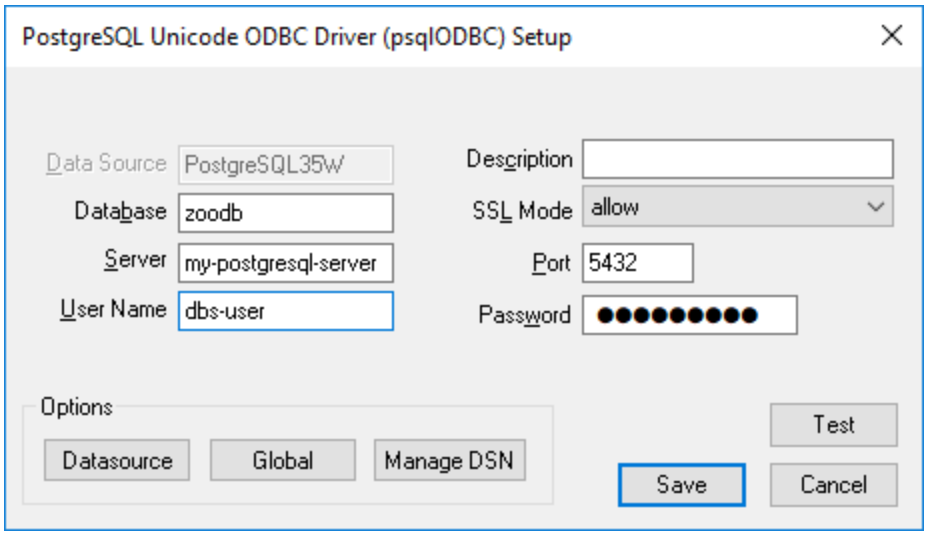
- *psqlODBC* driver must be installed on your system.
- You have the following database connection details: server, port, database, user name, and password.

Connection

1. Start the database connection wizard[508] and click **ODBC Connections**.
2. Select *User DSN* (or *System DSN* if you have administrative privileges), and then click **Create a New DSN** .
3. Select the driver from the drop-down list (*screenshot below*) and click **User DSN**. (If no PostgreSQL driver is available in the list, make sure that the PostgreSQL ODBC driver is installed on your operating system.)



4. In the dialog that appears (*screenshot below*), fill in the database connection (supplied by the database administrator).



5. Click **Save**.

The connection will become now available in the list of ODBC connections. To connect to the database, you can either double-click the connection or select it and then click **Connect**.

9.2.9.17 Progress OpenEdge (JDBC)

This example illustrates how to connect to a Progress OpenEdge 11.6 database server via JDBC.

Prerequisites

- JRE (Java Runtime Environment) or Java Development Kit (JDK) must be installed. This may be either Oracle JDK or an open source build such as Oracle OpenJDK. UModel will determine the path to the Java Virtual Machine (JVM) from the following locations, in this order: (i) the custom JVM path you may have set in application **Options**; see Java Settings 694; (ii) the JVM path found in the Windows registry; (iii) the `JAVA_HOME` environment variable.
- Make sure that the platform of UModel (32-bit, 64-bit) matches that of the JRE/JDK.

 © 2019-2025 Altova GmbH

- The operating system's `PATH` environment variable must include the path to the `bin` directory of the JRE or JDK installation directory, for example `C:\Program Files (x86)\Java\jre1.8.0_51\bin`.
- The Progress OpenEdge JDBC driver must be available on your operating system. In this example, JDBC connectivity is provided by the openedge.jar and pool.jar driver component files available in `C:\Progress\OpenEdge\java` as part of the OpenEdge SDK installation.
- You have the following database connection details: host, port, database path, name, or alias, username, and password.

Connection

1. Start the database connection wizard[508] and click **JDBC Connections** (*see JDBC Connection*[525] *for a screenshot of the dialog*).
2. In the *Classpaths* field, enter the path to the `.jar` file that provides connectivity to the database. If necessary, you can also enter a semicolon-separated list of `.jar` file paths. If you have added the filepath to the CLASSPATH[527] of the system, you can leave this field empty.
3. In the *Driver* field, select the appropriate driver. Relevant drivers will be available only if a valid `.jar` file path is in the *Classpaths* field or in the `CLASSPATH` environment variable.
4. Enter the username and password for the database in the corresponding fields.
5. Enter the JDBC connection string in the *Database URL* field according to the pattern in the table below, replacing the highlighted values with the ones applicable to your database server.
6. Click **Connect**.

Connection details of the Progress OpenEdge example

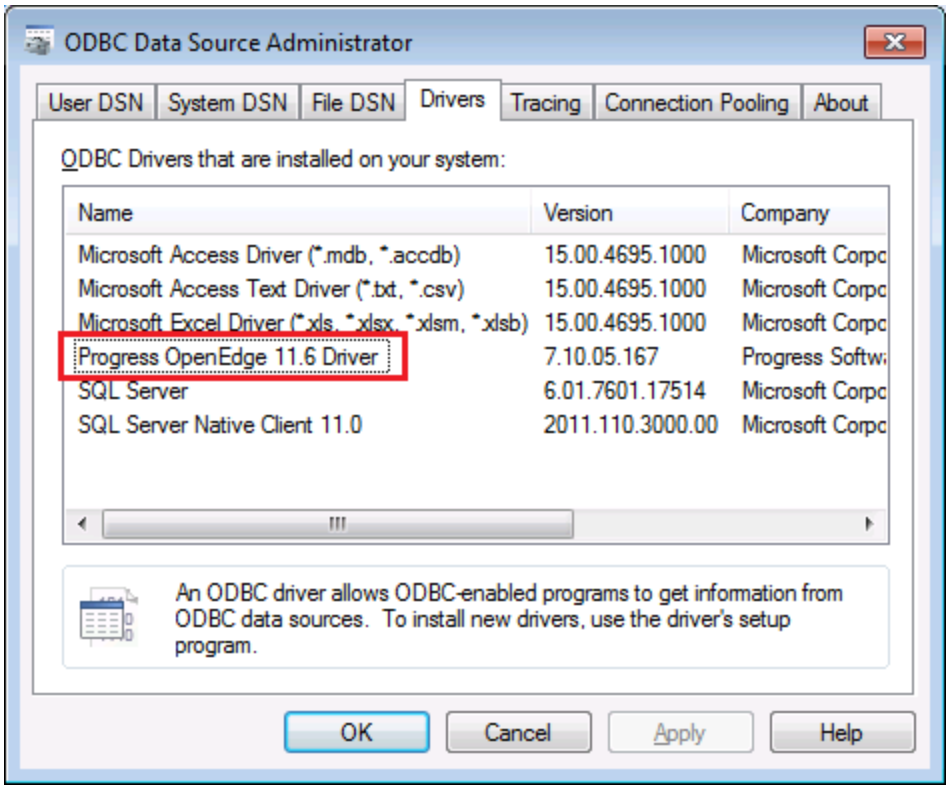
Field	Value
Classpaths	`C:\Progress\OpenEdge\java\openedge.jar;C:\Progress\OpenEdge\java\pool.jar;`
Driver	*com.ddtek.jdbc.openedge.OpenEdgeDriver*
Database URL	`jdbc:datadirect:openedge://host:port;databaseName=db_name`

9.2.9.18 Progress OpenEdge (ODBC)

This example shows how to connect to a Progress OpenEdge database server via the Progress OpenEdge 11.6 ODBC driver.

Prerequisites

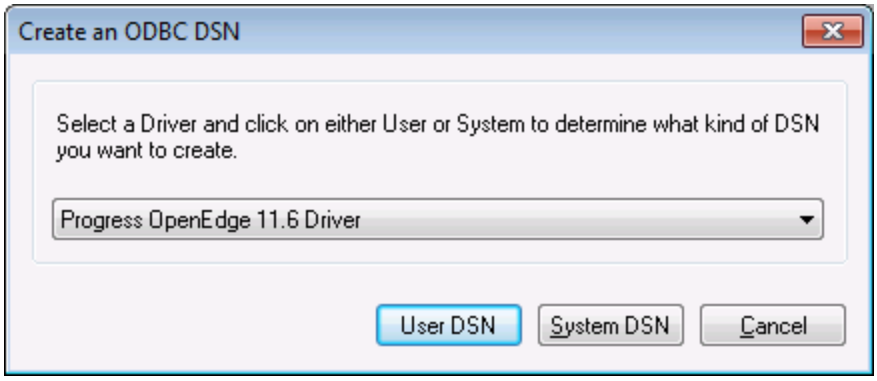
- The *ODBC Connector for Progress OpenEdge* driver must be installed on your system (see also Database Drivers Overview[510]). Download the version that corresponds to your version of UModel (32-bit or 64-bit). After installation, check if the ODBC driver is available on your machine[530].



- You have the following database connection details: host name, port number, database name, user ID, and password.

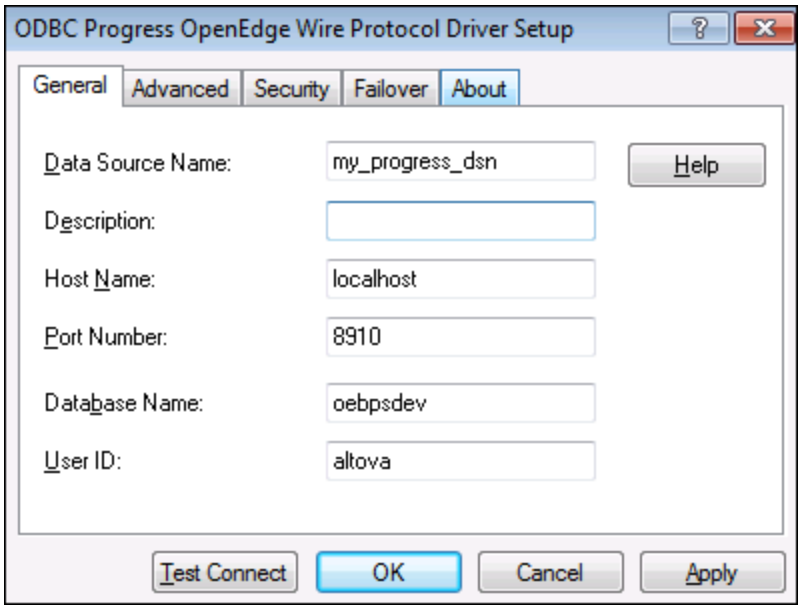
Connection

1. Start the database connection wizard[508] and click **ODBC Connections**.
2. Select *User DSN* (or *System DSN* if you have administrative privileges), and then click **Create a New DSN** .
3. Select *Progress OpenEdge Driver* from the drop-down list (*screenshot below*) and click **User DSN** (or **System DSN** if applicable).



4. Fill in the database connection details (Database, Server, Port, User Name, Password), and click **OK**.

© 2019-2025 Altova GmbH



5. To verify connectivity before saving the entered data, click **Test Connect**. To save, click **OK**.
6. The new data source now appears in the list of ODBC data sources. Click **Connect** to start the connection.

9.2.9.19 Sybase (JDBC)

This example illustrates how to connect to a Sybase database server via JDBC.

Prerequisites

- JRE (Java Runtime Environment) or Java Development Kit (JDK) must be installed. This may be either Oracle JDK or an open source build such as Oracle OpenJDK. UModel will determine the path to the Java Virtual Machine (JVM) from the following locations, in this order: (i) the custom JVM path you may have set in application **Options**; see Java Settings 694; (ii) the JVM path found in the Windows registry; (iii) the `JAVA_HOME` environment variable.
- Make sure that the platform of UModel (32-bit, 64-bit) matches that of the JRE/JDK.
- Sybase *jConnect* component must be installed on your operating system (in this example, *jConnect 7.0* is used, installed as part of the *Sybase Adaptive Server Enterprise PC Client* installation).
- You have the following database connection details: host, port, database path, name, or alias, username, and password.

Connection

1. Start the database connection wizard 508 and click **JDBC Connections** (*see JDBC Connection* 525 *for a screenshot of the dialog*).
2. In the *Classpaths* field, enter the path to the `.jar` file that provides connectivity to the database. If necessary, you can also enter a semicolon-separated list of `.jar` file paths. If you have added the filepath to the CLASSPATH 527 of the system, you can leave this field empty.
3. In the *Driver* field, select the appropriate driver. Relevant drivers will be available only if a valid `.jar` file path is in the *Classpaths* field or in the `CLASSPATH` environment variable.

4. Enter the username and password for the database in the corresponding fields.
5. Enter the JDBC connection string in the *Database URL* field according to the pattern in the table below, replacing the highlighted values with the ones applicable to your database server.
6. Click **Connect**.

Connection details of the Sybase example

Field	Value
Classpaths	`C:\sybase\jConnect-7_0\classes\jconn4.jar`
Driver	*com.sybase.jdbc4.jdbc.SybDriver*
Database URL	`jdbc:sybase:Tds:hostName:port/databaseName`

9.2.9.20 Teradata (JDBC)

This example illustrates how to connect to a Teradata database server via JDBC.

Prerequisites

- JRE (Java Runtime Environment) or Java Development Kit (JDK) must be installed. This may be either Oracle JDK or an open source build such as Oracle OpenJDK. UModel will determine the path to the Java Virtual Machine (JVM) from the following locations, in this order: (i) the custom JVM path you may have set in application **Options**; see Java Settings 694; (ii) the JVM path found in the Windows registry; (iii) the `JAVA_HOME` environment variable.
- Make sure that the platform of UModel (32-bit, 64-bit) matches that of the JRE/JDK.
- The JDBC driver (one or more `.jar` files that provide connectivity to the database) must be available on your operating system. In this example, Teradata JDBC Driver 16.20.00.02 is used. For more information, see https://downloads.teradata.com/download/connectivity/jdbc-driver.
- You have the following database connection details: host, port, database path, name, or alias, username, and password.

Connection

1. Start the database connection wizard 508 and click **JDBC Connections** (*see JDBC Connection* 525 *for a screenshot of the dialog*).
2. In the *Classpaths* field, enter the path to the `.jar` file that provides connectivity to the database. If necessary, you can also enter a semicolon-separated list of `.jar` file paths. If you have added the filepath to the CLASSPATH 527 of the system, you can leave this field empty.
3. In the *Driver* field, select the appropriate driver. Relevant drivers will be available only if a valid `.jar` file path is in the *Classpaths* field or in the `CLASSPATH` environment variable.
4. Enter the username and password for the database in the corresponding fields.
5. Enter the JDBC connection string in the *Database URL* field according to the pattern in the table below, replacing the highlighted values with the ones applicable to your database server.
6. Click **Connect**.

Connection details of the Teradata example

© 2019-2025 Altova GmbH

Field	Value
Classpaths	`C:\jdbc\teradata\`
Driver	*com.teradata.jdbc.TeraDriver*
Database URL	`jdbc:teradata://databaseServerName`

9.2.9.21 Teradata (ODBC)

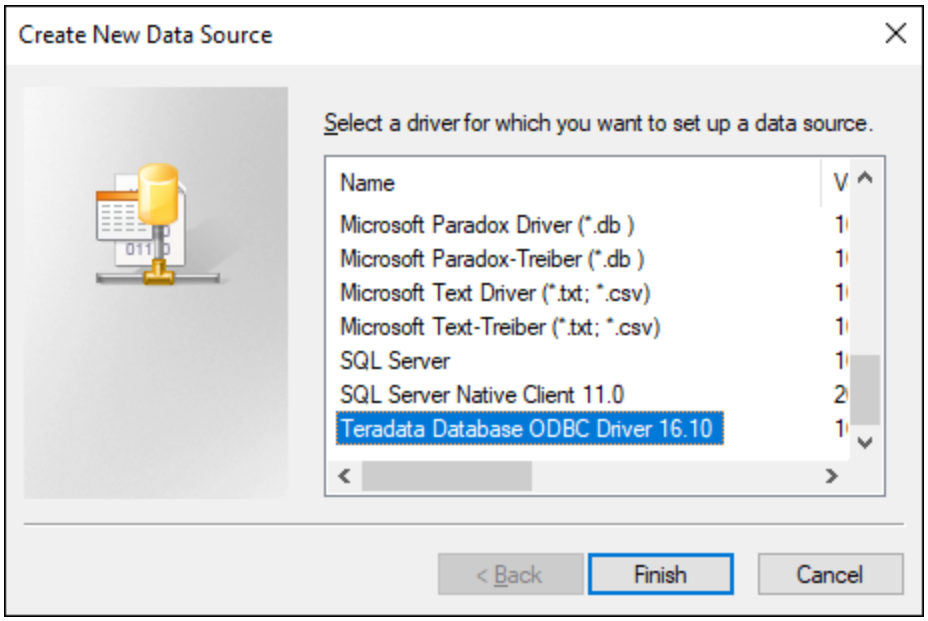
This example illustrates how to connect to a Teradata database server via ODBC.

Prerequisites

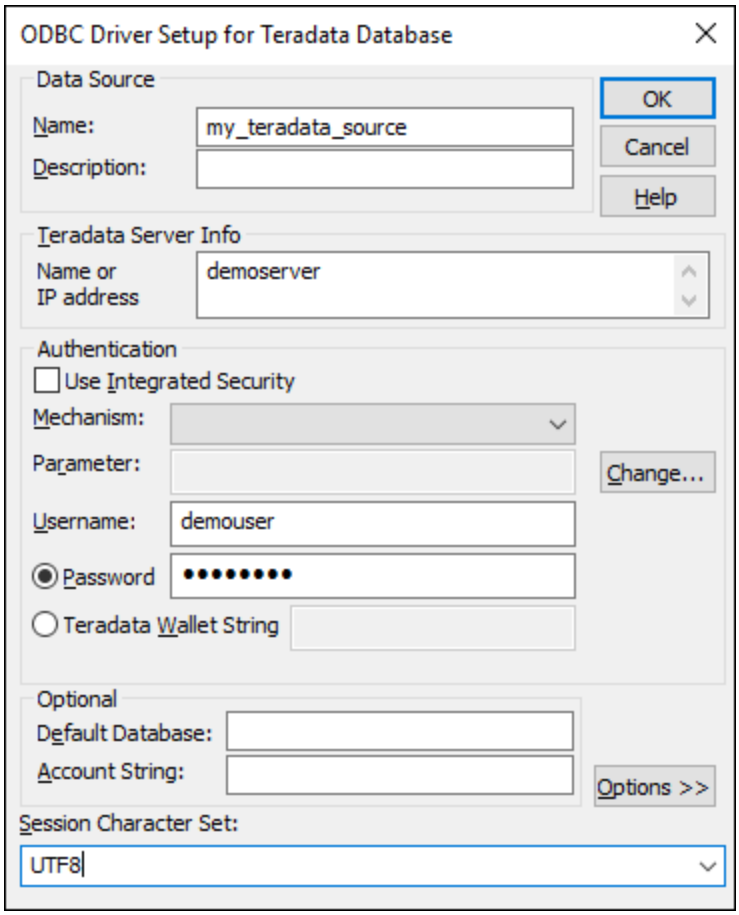
- The Teradata ODBC driver must be installed (downloadable from the Teradata website).
- You have the following database connection details: host, username, and password.

Connection

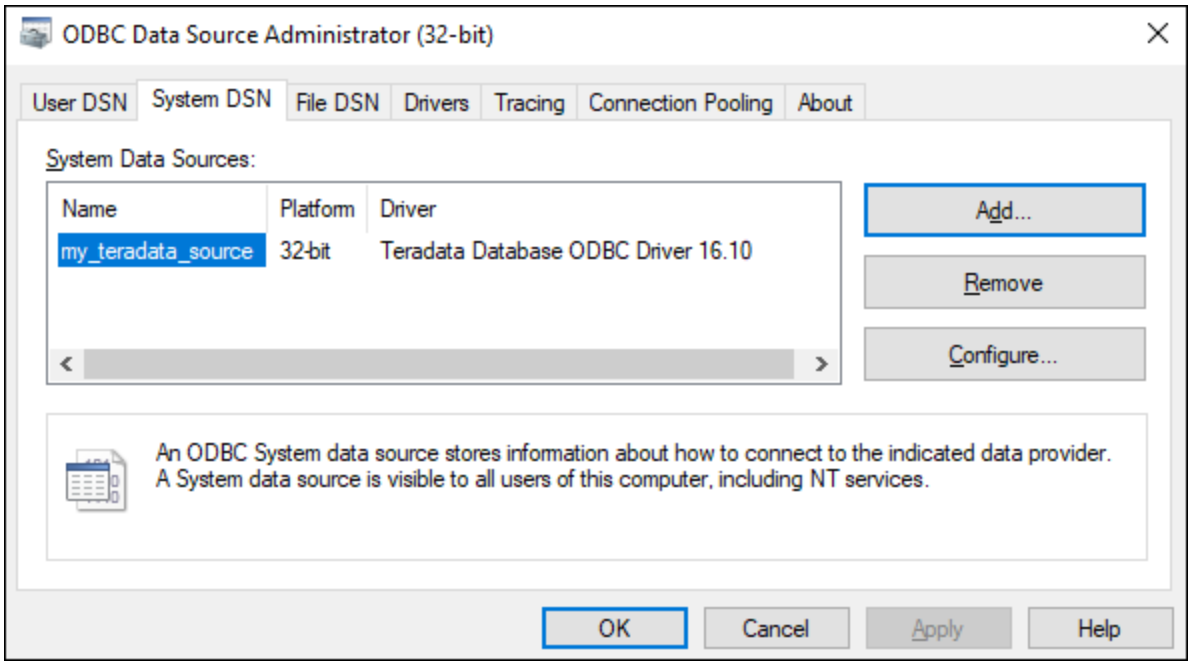
1. Press the **Windows** key, start typing `ODBC`, and select *Set up ODBC data sources (32-bit)* from the list of suggestions. (If appropriate, select the 64-bit option instead.)
2. In the ODBC Data Source Administrator dialog that appears, click the *System DSN* tab, and then click **Add**.
3. In the Create New Data Source dialog (screenshot below), select *Teradata Database ODBC Driver* and click **Finish**.



4. Enter the database and authentication details, and click **OK** when done.

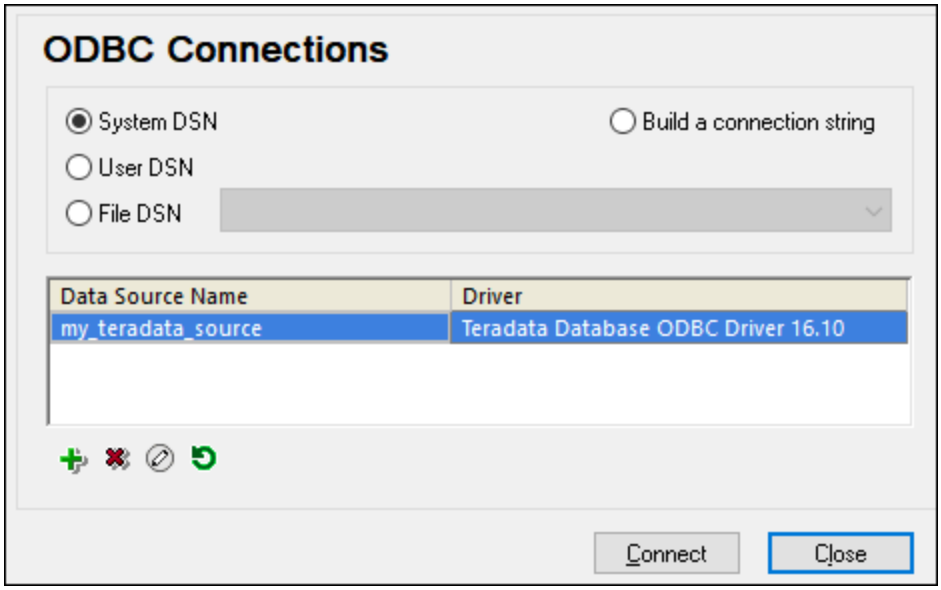


5. Click **OK**. The data source now appears in the Data Sources list.



© 2019-2025 Altova GmbH

6. Run UModel and start the database connection wizard[508].
7. Click **ODBC Connections** and select *System DSN* (*see screenshot below*).



8. The data source you just created will be listed. Click **Connect** to start teh connection.

Note: If you get the following error: *"The driver returned invalid (or failed to return) SQL_DRIVER_ODBC_VER: 03.80"*, then make sure that the path to the ODBC client (for example, `C:\Program Files\Teradata\Client\16.10\bin`) exists in your system's **PATH** environment variable. If this path is missing, then add it manually.

10 XMI - XML Metadata Interchange

Altova website: Exchanging UModel projects using XMI

You can export UModel projects to XML Metadata Interchange (XMI) files, and import XMI files as UModel projects. This provides interoperability with other UML tools that support XMI. The supported XMI versions are as follows:

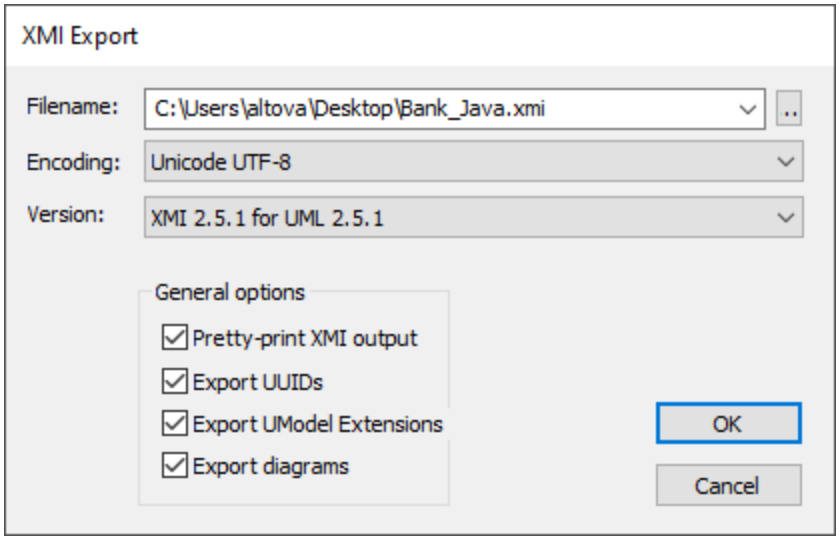
- XMI 2.1 for UML 2.0
- XMI 2.1 for UML 2.1.2
- XMI 2.1 for UML 2.2
- XMI 2.1 for UML 2.3
- XMI 2.4.1 for UML 2.4.1
- XMI 2.4.1 for UML 2.5
- XMI 2.5.1 for UML 2.5.1

To import an XMI file into UModel:

- On the **File** menu, click **Import from XMI File**.

To export a UModel project to an XMI file:

- On the **File** menu, click **Export to XMI File**.



Notes:

- During the export process, all included files, even those defined as include by reference 166, are exported.
- If you intend to re-import generated XMI code into UModel, make sure that you select the **Export UModel Extensions** check box.

© 2019-2025 Altova GmbH

The sections below describe options available when exporting projects to XMI.

Pretty-print XMI output

If you select this option, the XMI file will be generated with XML tag indentation and carriage returns.

Export UUIDs

XMI defines three versions of element identification: IDs, UUIDs and labels.

- IDs are unique within the XMI document, and are supported by most UML tools. UModel exports these type of IDs by default, i.e. none of the check boxes need activated.
- UUID are Universally Unique Identifiers, and provide a mechanism to assign each element a global unique identification, GUID. These IDs are globally unique, i.e. they are not restricted to the specific XMI document. UUIDs are generated by selecting the "Export UUIDs" check box.
- UUIDs are stored in the standard canonical UUID/GUID format (e.g "6B29FC40-CA47-1067-B31D-00DD010662DA", "550e8400-e29b-41d4-a716-446655440000",...)
- Labels are not supported by UModel.

Note: The XMI import process automatically supports both types of IDs.

Export UModel Extensions

XMI defines an "extension mechanism" which allows each application to export its tool-specific extensions to the UML specification. Other UML tools will, however, only be able to import the standard UML data (ignoring the UModel extensions). This UModel extension data will be available when importing into UModel.

Data such as the file names of classes, or element colors, are not part of the UML specification and thus have to be deleted in XMI, or be saved in "Extensions". If they have been exported as extensions and re-imported, all file names and colors will be imported as defined. If extensions are not used for the export process, then these UModel-specific data will be lost.

When importing an XMI document, the format is automatically detected and the model generated.

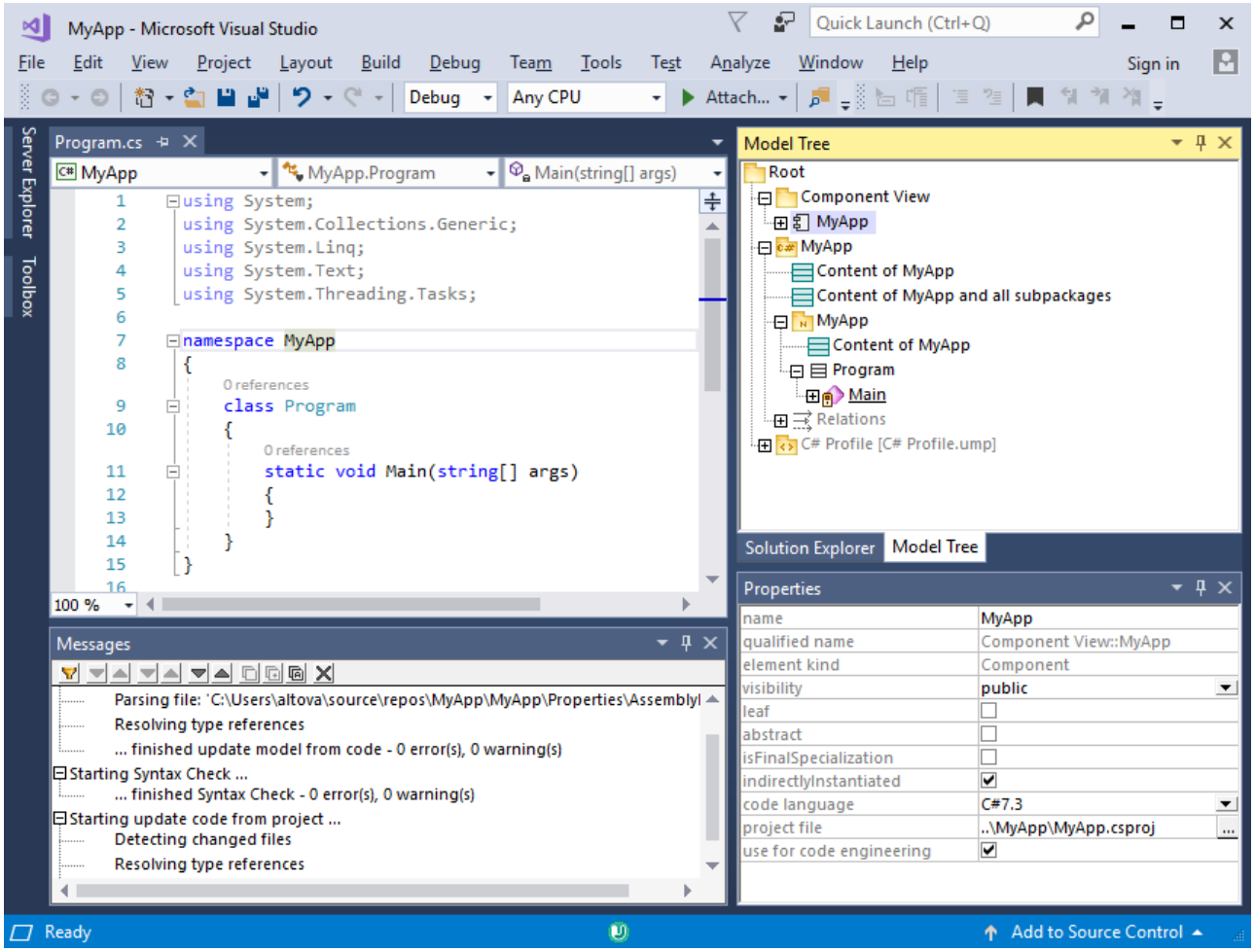
Export diagrams

Exports UModel diagrams as "Extensions" in the XMI file. The option **Export UModel Extensions** must be selected before you can save the diagrams as extensions.

11 UModel Plug-in for Visual Studio

You can integrate UModel 2025 into the Microsoft Visual Studio versions 2012/2013/2015/2017/2019/2022. This unifies the best of both worlds, combining the modeling capabilities of UModel with the development environment of Visual Studio.

One of the main benefits to using UModel as a Visual Studio plug-in is automatic synchronization between the C# or VB.NET code and the UML model. This means that, if you make changes to your code in Visual Studio, these are automatically propagated to the model. Likewise, if you make changes to the model (for example, by editing class diagrams), these would be propagated to the code. If necessary, you can disable automatic synchronization, and synchronize the code and the model manually (in either direction).



Sample Visual Studio 2017 project with UModel Plug-in support

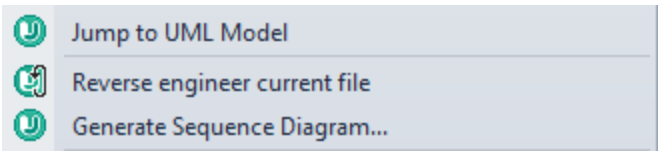
Compared to the standalone edition of UModel, the UModel plug-in for Visual Studio has the following behavior:

- Automatic synchronization between the UModel model and the project code is available, in either direction (see Synchronizing the Model and Code 578).
- In Visual Studio 2019, the functionality of UModel is available in the Extensions menu. In older versions of Visual Studio, the UModel functionality is accessible from the following menus:

© 2019-2025 Altova GmbH

File	Contains menu entries from both UModel and Visual Studio.
Edit	Contains menu entries from both UModel and Visual Studio.
View	The UModel-specific commands are grouped under **View | UModel**.
Project	The UModel-specific commands are grouped under **Project | UModel**.
Layout	Same as in the standalone edition of UModel.
Tools	Contains menu entries from both UModel and Visual Studio. The UModel options are available under **Tools | UModel options**.
Help	The UModel help is available under **Help | UModel Help**.

- When the cursor is in the Visual Studio code editor, the following new context menu items are available (in contexts where these commands are meaningful):
  - Jump to UML Model
  - Reverse engineer current file
  - Generate Sequence Diagram...



  On the other hand, when the cursor is inside an element in the Model Tree window, the **Jump to Code** context menu item is available (in contexts where this command is meaningful).
- When UModel runs as a Visual Studio plug-in, you can use the version control functionality available in Visual Studio. The source control commands from the standalone edition of UModel available through the Microsoft Source Control Plug-in API are not supported.
- The dialogs triggered by the commands **UModel | Import Source Directory** and **UModel | Import Source Project** do not have the option to select "C#" and "Visual Basic" in the Language combo box. Import of existing projects is done through Visual Studio commands (for example, in versions older than 2019, **File | Add | Existing Project**).
- The Scripting Editor (**Tools | Scripting Editor**) and the menu option **Tools | Restore Toolbars and Windows** are not supported.

11.1 Installing the UModel Plug-in for Visual Studio

To install the UModel Plug-in for Visual Studio, take the steps below:

1. Install Microsoft Visual Studio 2012/2013/2015/2017/2019/2022. Note that from Visual Studio 2022 onwards, Visual Studio is being made available only as a 64-bit application.
2. Install UModel (Enterprise or Professional Edition). If you have installed Visual Studio 2022+, then you must install the 64-bit version of UModel.
3. Download and run the UModel integration package for Microsoft Visual Studio. This package is available on the UModel (Enterprise and Professional Editions) download page at www.altova.com.

Once the integration package has been installed, you will be able to use UModel in the Visual Studio environment.

Important

You must use the integration package corresponding to your UModel version (current version is 2025). The integration package is not edition-specific and can therefore be used for both Enterprise and Professional editions.

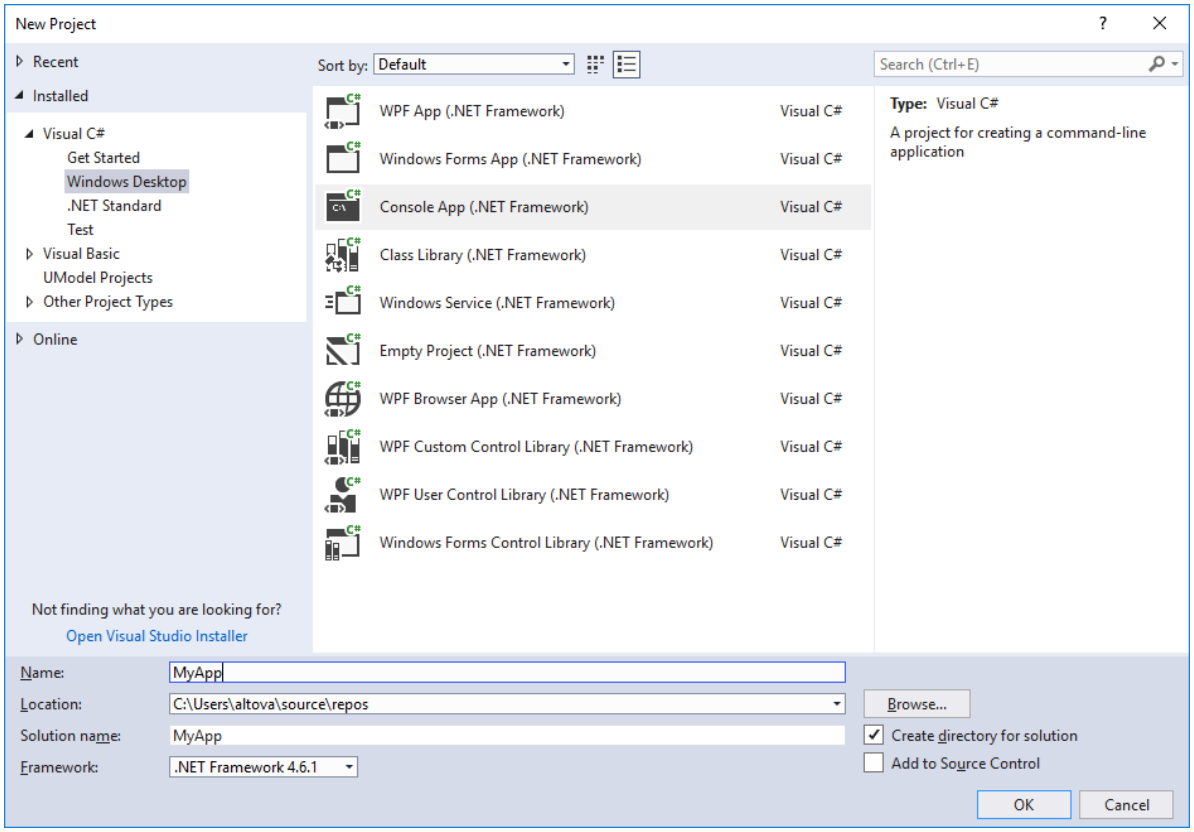
© 2019-2025 Altova GmbH

11.2 Adding UModel Support to Visual Studio Projects

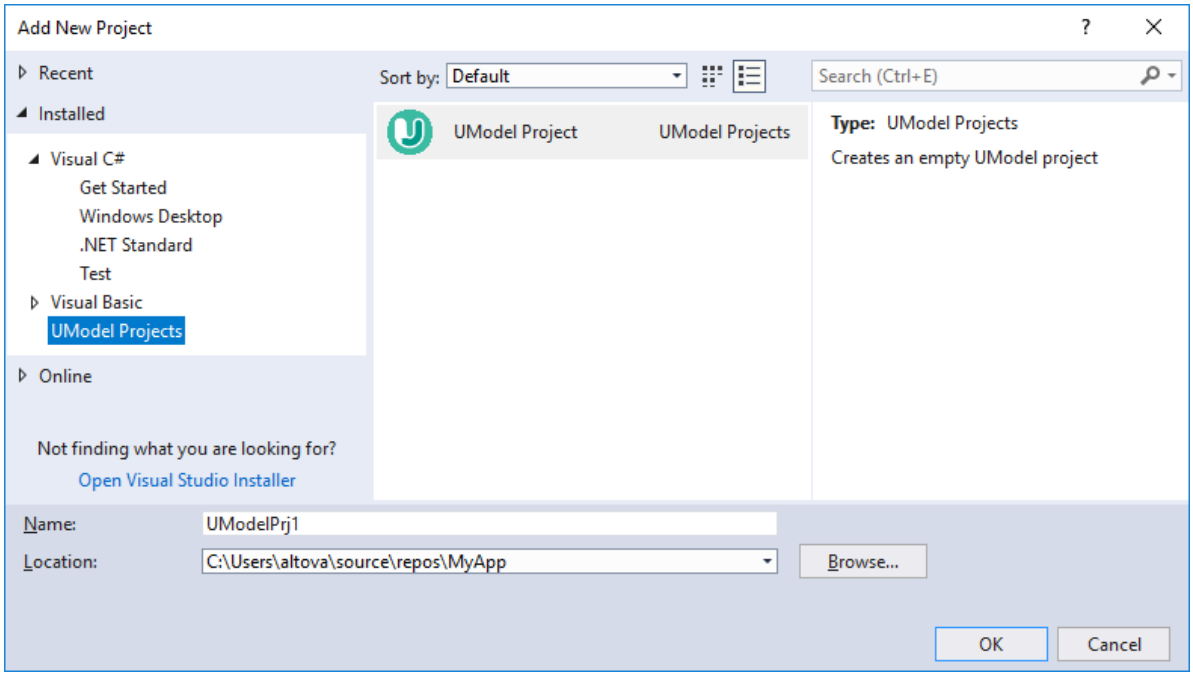
Adding UModel support to new or existing Visual Studio projects enables you to set up automatic synchronization between your Visual Studio project and the UModel model. A Visual Studio solution can contain one UModel project (not more).

To add UModel support to a Visual Studio project:

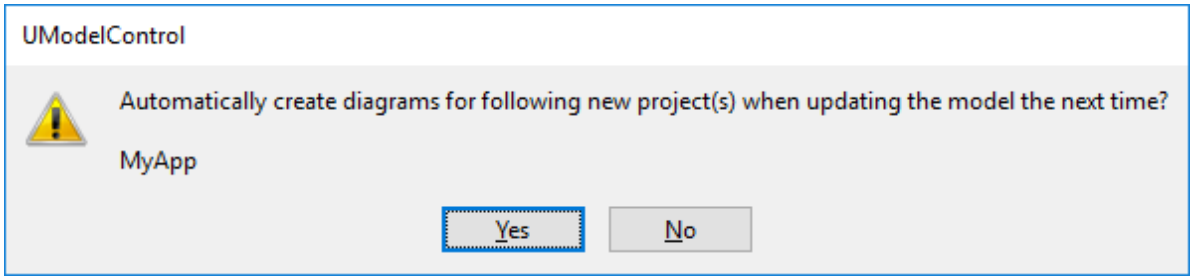
1. Create a new Visual Studio project, or open an existing one. (In this example, a new C# project called "MyApp" is being created with Visual Studio 2017).



2. On the **File** menu, click **Add**, and then click **New Project**.
3. Select **UModel Projects**, and click OK.

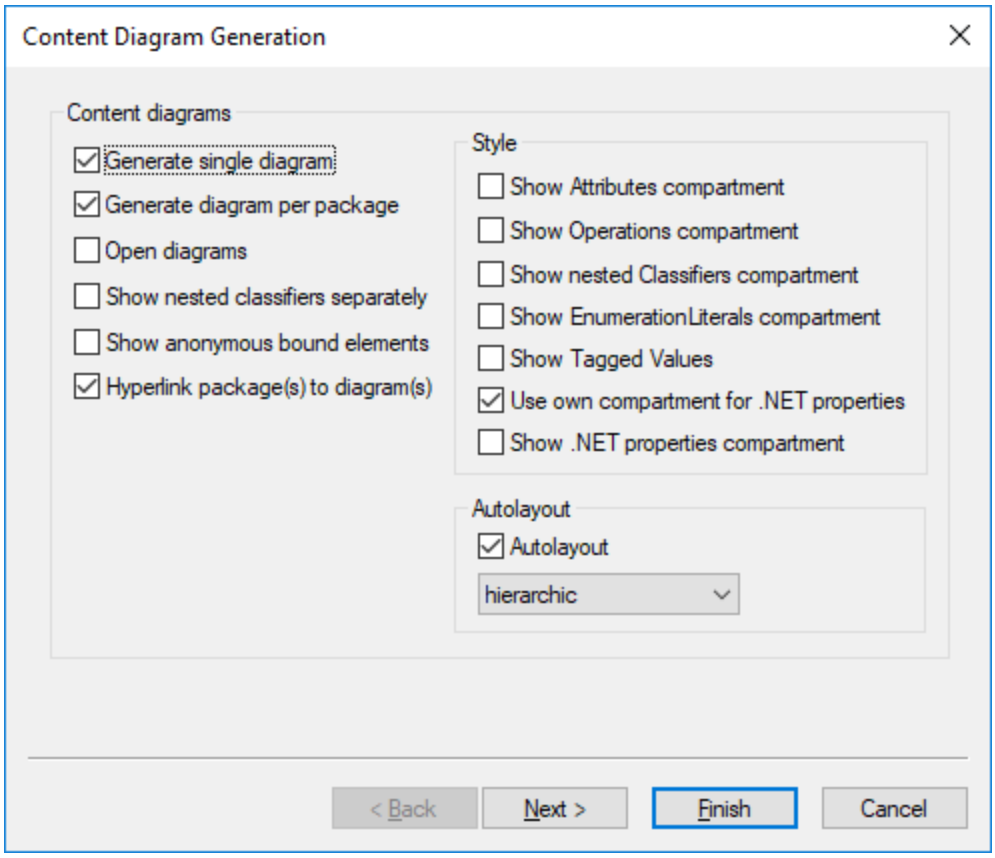


4. If you want diagrams to be created automatically in the model based on the code, click **Yes** when prompted (this is the recommended option).

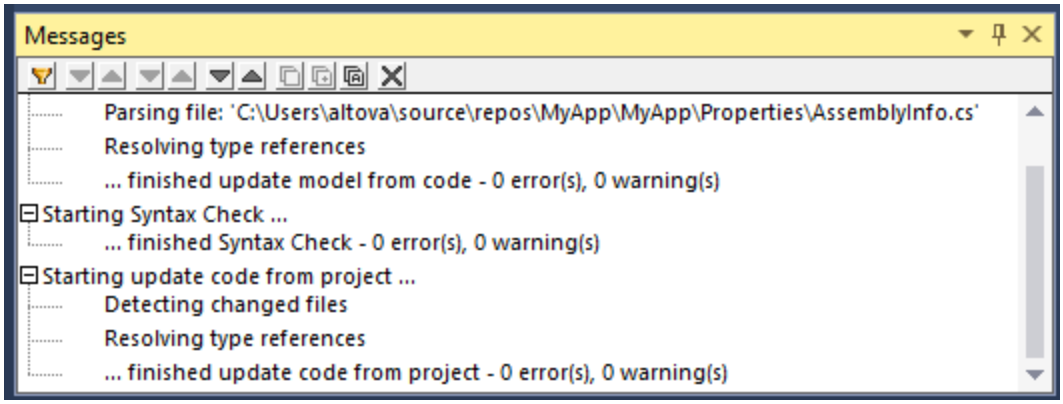


5. When prompted to select the diagrams generation options, choose your preferences as you go through the wizard steps, and click **Finish**. These steps are the same as in the standalone edition of UModel.

© 2019-2025 Altova GmbH



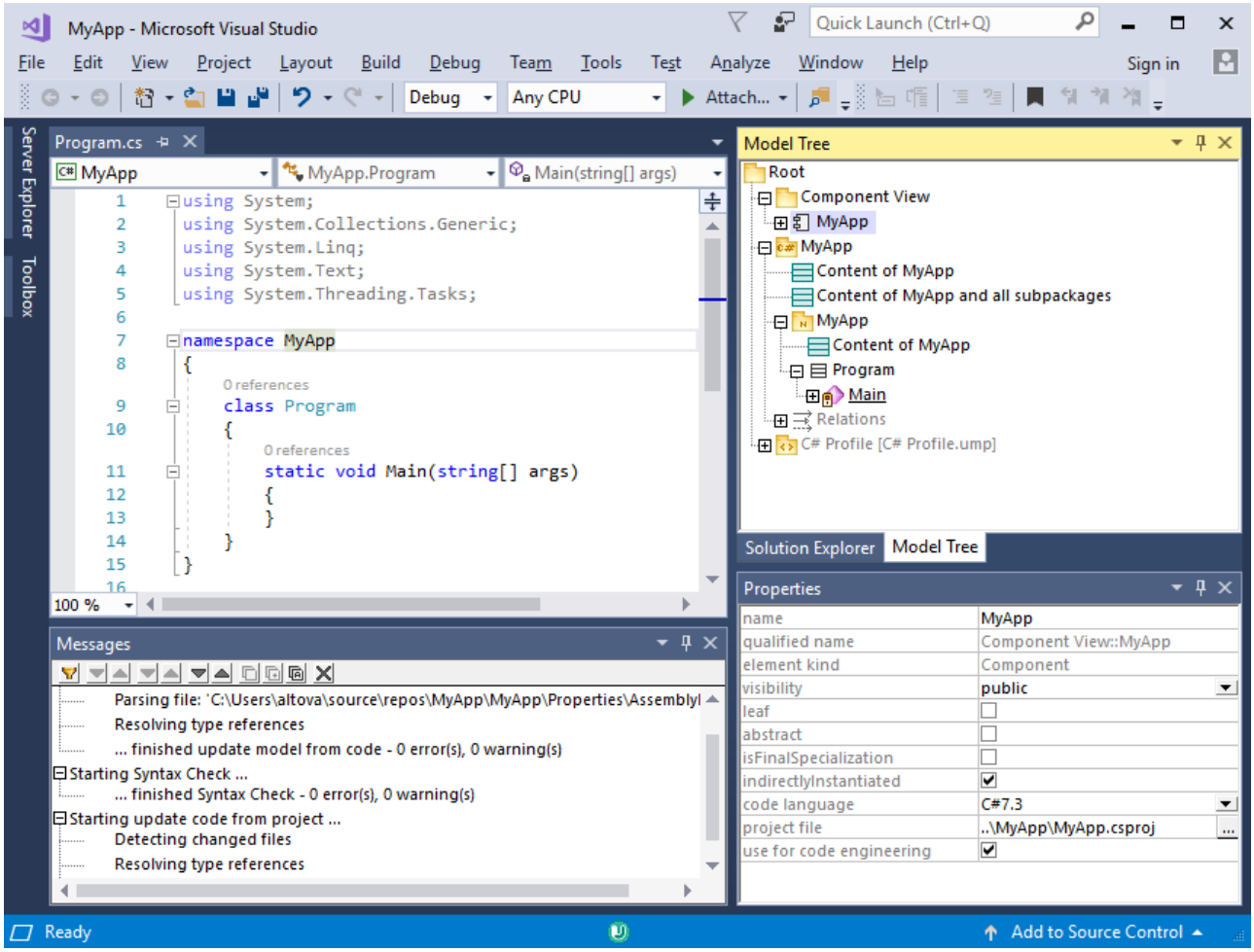
When you click **Finish**, UModel starts the synchronization process and displays a dialog box. Click **OK** to close the dialog box. The synchronization details are displayed in the Messages window.



Note that the Messages window might not be visible by default in Visual Studio. You can display this window (and all other UModel-specific windows) by selecting the menu command **View | UModel | [Name of the window]**.

When you add a new UModel project to a Visual Studio solution, the settings required for code engineering (such as the component realization, and the C# or VB.NET profile) are defined automatically. To view these settings, open the **Model Tree** and the **Properties** windows (on the **View** menu, click **UModel | Model Tree**

and **UModel | Properties**, respectively). Make sure to click the code engineering component in the Model Tree window (in this case, "MyApp") in order to populate the Properties window.



© 2019-2025 Altova GmbH

11.3 Loading/Unloading UModel Projects

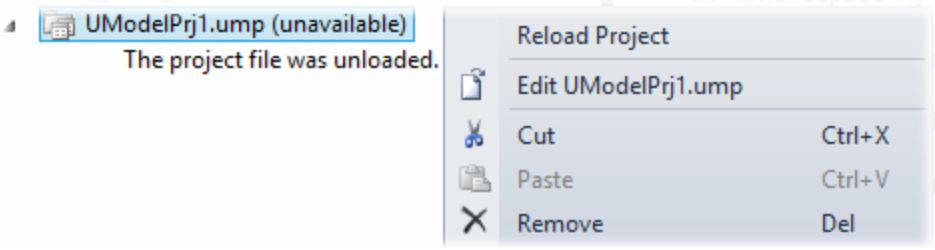
After you add an UModel project to a Visual Studio solution, it appears in the **Solution Explorer** of Visual Studio along with any other projects that are part of the solution. If necessary, you can temporarily unload the UModel project from the solution. When an UModel project is unloaded from the solution, its files remain on the disk, and in the **Solution Explorer**. This way, you can reload the project back into the solution at a later time.

To unload an UModel project from a Visual Studio solution:

1. Click the UModel project in Solution Explorer of Visual Studio.
2. On the **Project** menu, click **Unload project**.

To reload the UModel project back into the solution:

- Right-click the project in Solution Explorer, and click **Reload Project**.



To remove the UModel project from the Visual Studio solution:

- Unload the project, as shown above.
- Right-click the project in Solution Explorer, and click **Remove**.

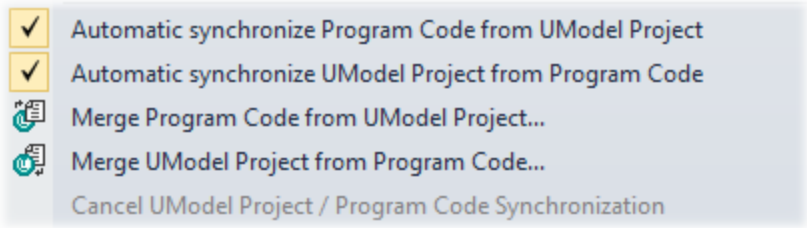
11.4 Synchronizing the Model and Code

The synchronization process between the UModel .ump file (the model) and the C# or VB.NET code can be manual or automatic.

Automatic synchronization takes place once you add UModel support to your Visual Studio project (see Adding UModel Support to Visual Studio Projects 573). Automatic synchronization means that, whenever you edit the code, the UModel Plug-in for Visual Studio parses the code and updates the model. Likewise, if you make changes to the model (for example, by editing a diagram), the code is updated accordingly. Manual synchronization, on the other hand, is initiated on demand, as shown below.

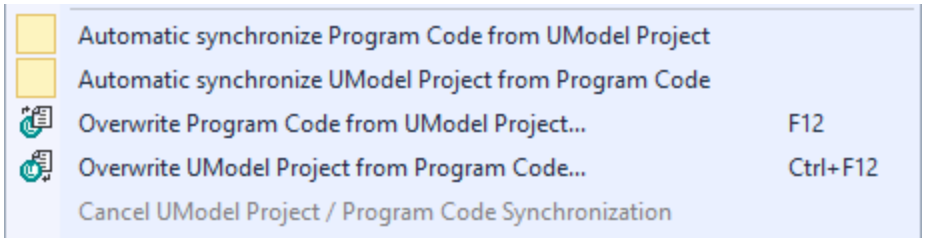
Both the automatic and the manual synchronization update changes in bulk, for the entire project. When UModel runs as a Visual Studio plug-in, the option to merge or update a single class is not available in the Model Tree.

The commands which control automatic or manual synchronization are available in the **Project | UModel Project** menu:



Code synchronization menu commands (Visual Studio 2010)

In newer versions of Visual Studio, selected menu items have a slightly different appearance:



Code synchronization menu commands (Visual Studio 2017)

The meaning of each command is as follows.

Automatic synchronize Program Code from UModel Project	This menu option is switched on by default, meaning that synchronization from model to code is set to take place automatically. To enable or disable automatic synchronization, click the menu item.
Automatic synchronize UModel Project from Program Code	Same as above, in the opposite direction (from code to model).

 © 2019-2025 Altova GmbH

Merge Program Code from UModel Project	Updates the program code with changes made in the UModel project (same functionality as in the standalone version). The name of this command changes to **Overwrite Program Code from UModel Project**, if you have set this option from **Project | UModel Project | Synchronization Settings**.
Merge UModel Project from Program Code	Updates the UModel project with changes made in the program code (same functionality as in the standalone version). The name of this command changes to **Overwrite UModel Project from Program Code**, if you have set this option from **Project | UModel Project | Synchronization Settings**.
Cancel UModel Project / Program Code Synchronization	Enables you to cancel a synchronization operation which is in progress. When no synchronization operation is in progress, this option is disabled.

During synchronization, the progress of the operation appears in the Visual Studio status bar, for example:

Code synchronization between code and model cannot take place in the following cases:

- Code is not parseable
- The last reverse engineering or forward engineering process encountered an error.
- The syntax check throws an error in UModel.

In such cases, the error details are displayed in the **Messages** window. To open the source file which contains the error, click the corresponding line in the **Messages** window. The cursor will be positioned on the line containing the error.

Automatic synchronization limitations

Some C# and VB.NET code modifications in Visual Studio do not trigger an internal Visual Studio event and are thus not automatically updated in UModel. In such cases, you can either perform a forced synchronization manually, or make a different modification which triggers a source file update. Manual synchronization is necessary when adding or changing the following entities:

- Default values for attributes
- Default values for operation parameters
- TemplateParameters
- TemplateBindings
- Summary section for all elements
- Remark section for all elements
- All changes in method bodies

Note that if you change any of the above-mentioned modeling elements in the model, automatic code synchronization will take place normally. There are no limitations when automatic synchronization is from model to code.

To perform a forced manual synchronization from code to model, right-click the source code file in the code editor and select **Reverse engineer current file** from the context menu.

If your UModel project contains the language profile for Java, then automatic synchronization is automatically disabled for that project in Visual Studio, and a message box informs you of this. Such projects must be synchronized manually (using the menu commands **UModel | Merge Program Code from UModel Project**, and **UModel | Merge UModel Project from Program Code**). Alternatively, consider using the UModel Plug-in for Eclipse (see UModel Plug-in for Eclipse 581).

© 2019-2025 Altova GmbH

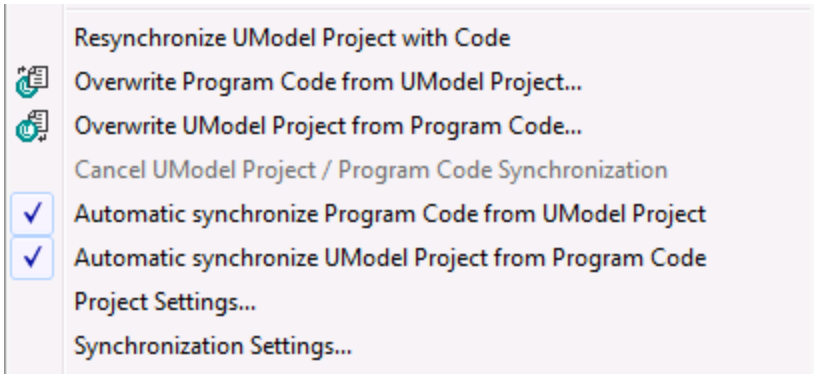
12 UModel Plug-in for Eclipse

Eclipse is an open source framework that integrates different types of applications delivered in form of plug-ins. The UModel plug-in for the Eclipse Platform allows you to access UModel functionality directly from Eclipse (versions 2024-12 (4.34), 2024-09 (4.33), 2024-06 (4.32), 2024-03 (4.31)), while also exposing some Eclipse-specific behavior discussed in this chapter.

One of the main benefits to using UModel as an Eclipse plug-in is automatic synchronization between the Java code and the UModel model. This means that, if you make changes to your Java code in Eclipse, these are automatically propagated to the model. Conversely, if you make changes to the model (for example, by editing class diagrams), these would be propagated to the code. If necessary, you can disable automatic synchronization, and synchronize the code and the model manually (in either direction).

Compared to the standalone version of UModel, the UModel plug-in for Eclipse has the following behavior:

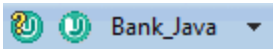
- In Eclipse, several graphical user interface elements conform to the specifics of the Eclipse development environment (see The UModel Perspective 586). As in the standalone version, some user interface elements may be disabled or not available if the context is not relevant. For example, the UModel toolbar buttons are shown based on the kind of diagram active in the main editor.
- In Eclipse, a **UModel** menu is available—it corresponds to the **Project** menu in the standalone version of UModel. While most of the commands in this menu are not different to the standalone version, there are several new commands that enable you to control automatic synchronization:



Resynchronize UModel Project with Code	Enables you to explicitly initiate the synchronization between the UModel project and the program code (this may be the case when last automatic synchronization has failed due to any reason).
Merge Program Code from UModel Project	Updates the program code with changes made in the UModel project (same functionality as in the standalone version).
Merge UModel Project from Program Code	Updates the UModel project with changes made in the program code (same functionality as in the standalone version).
Cancel UModel Project / Program Code Synchronization	Enables you to cancel a synchronization operation which is in progress. When no synchronization operation is in progress, this option is disabled.

Automatic synchronize Program Code from UModel Project	This menu option is switched on by default, meaning that synchronization from model to code is set to take place automatically. To disable automatic synchronization, switch it off.
Automatic synchronize UModel Project from Program Code	Same as above, in the opposite direction (from code to model).

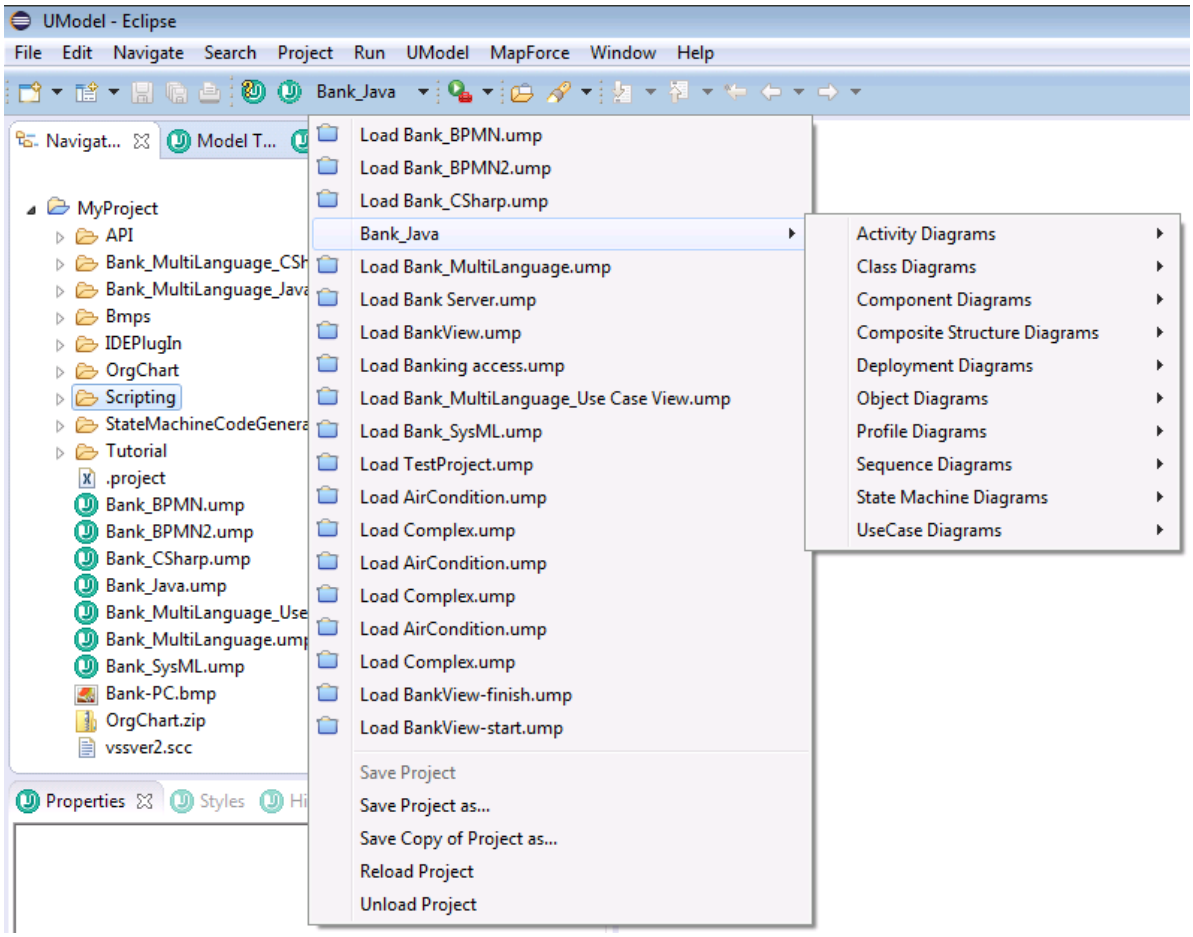
- The version control commands available in the standalone version of UModel through the Microsoft Source Control Plug-in API are not supported in Eclipse. Instead, you have the flexibility to use third-party version control systems that can integrate with Eclipse.
- The dialogs triggered by the commands **UModel | Import Source Directory** and **UModel | Import Source Project** do not have the option to select "Java" in the Language combo box. To import Java source code into an Eclipse project, use the standard Eclipse commands (for example, **File | Import**).
- In Eclipse, a new toolbar is available—the UModel toolbar, which contains some general as well as project-related commands.



The toolbar button opens the help file. The toolbar button displays the current status of the code engineering process (when it turns red this indicates an error, and you can view the details in the Messages view). Finally, the drop-down list in the toolbar has several functions:

  - It enables you to quickly load or unload in Eclipse a particular UModel project (.ump) file. Your Eclipse project must include at least one UModel project (.ump) file; otherwise, the drop-down list is disabled.
  - When a UModel project is loaded, it provides several contextual commands, including quick access to any of the diagrams of the loaded project:

© 2019-2025 Altova GmbH



- The Scripting Editor (**Tools | Scripting Editor**) and the menu option **Tools | Restore Toolbars and Windows** are not supported.
- The UModel **Help**, **Support Center**, **Check for Updates** and **About** menus are available in the **Help | UModel Help** menu of Eclipse. The version information of the UModel Plug-in for Eclipse is also available from the Eclipse menu (select **Help | About Eclipse**, and then click the UModel icon).

12.1 Installing the UModel Plug-in for Eclipse

Prerequisites

- Eclipse 2024-12 (4.34), 2024-09 (4.33), 2024-06 (4.32), 2024-03 (4.31) (http://www.eclipse.org), 64-bit.
- A Java Runtime Environment (JRE) or Java Development Kit (JDK) for the 64-bit platform.
- UModel Enterprise or Professional Edition 64-bit.

Note: All the prerequisites listed above must have the 64-bit platform. Integration with older Eclipse 32-bit platforms is no longer supported, although it may still work.

After the prerequisites listed above are in place, you can install the UModel Integration Package (64-bit) to integrate UModel in Eclipse. The integration can be carried out either during the installation of the Integration Package or manually from Eclipse after the Integration Package has been installed. The UModel Integration Package is available for download at https://www.altova.com/components/download.

Note: Eclipse must be closed while you install or uninstall the UModel Integration Package.

Integrate UModel during installation of the Integration Package

You can integrate UModel in Eclipse during the installation of the UModel Integration Package. Do this as follows:

1. Run the UModel Integration Package to start the installation wizard.
2. Go through the initial steps of the installation with eth wizard.
3. In the Integration step, select *Let this wizard integrate Altova UModel plug-in into Eclipse*, and browse for the directory where the Eclipse executable (`eclipse.exe`) is located.
4. Click **Next** and complete the installation.

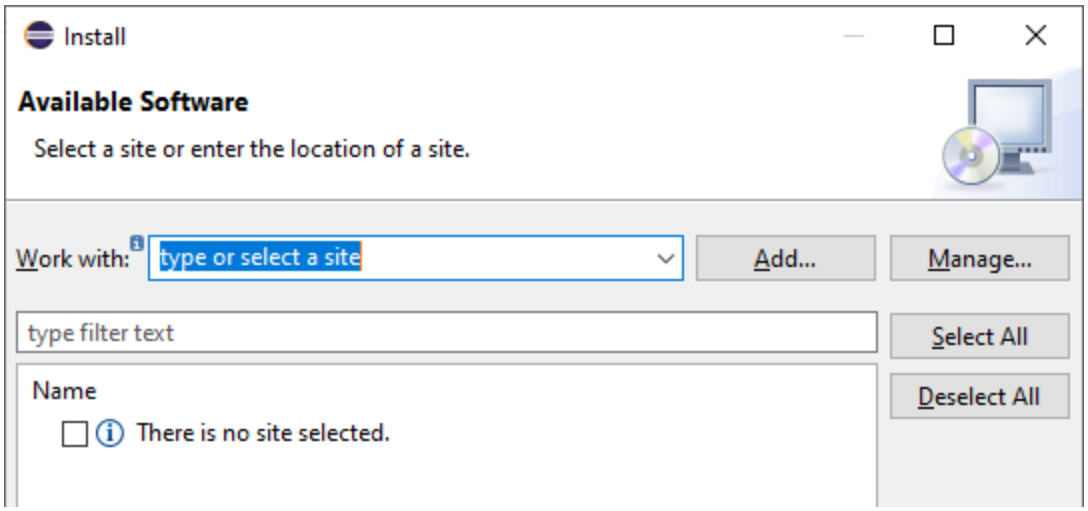
The UModel perspective and menus will be available in Eclipse the next time you start it.

Integrate UModel in Eclipse manually

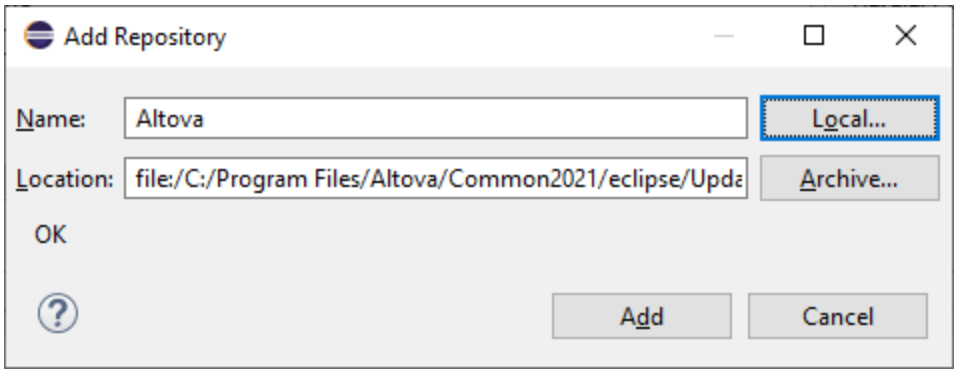
After you have installed the UModel Integration Package, you can manually integrate UModel in Eclipse as follows:

1. In Eclipse, select the menu command **Help | Install New Software**.
2. In the Install dialog box, click **Add**.

© 2019-2025 Altova GmbH



3. In the Add Repository dialog box, click **Local**. Browse for the folder `C:\Program Files\Altova\Common2025\eclipse\UpdateSite`, and select it. Provide a name for the site (such as "Altova").

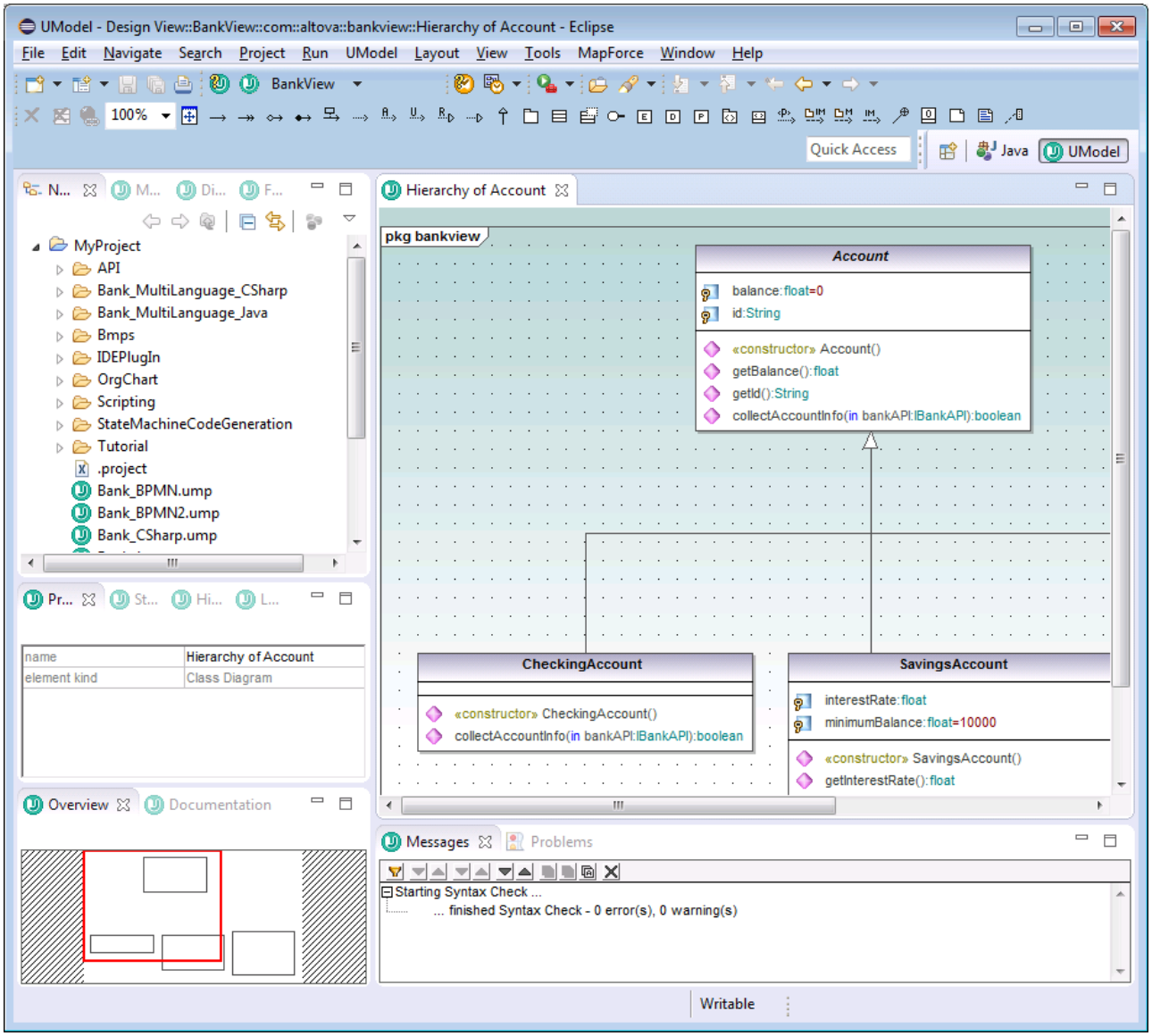


4. Repeat the steps 2-3 above, this time selecting the folder `C:\Program Files\Altova\<%APPNAMESHORT%>\eclipse\UpdateSite` and providing a name such as "Altova UModel".
5. On the Install dialog box, select *Only Local Sites*. Next, select the "Altova category" folder and click **Next**.
6. Review the items to be installed and click **Next** to proceed.
7. To accept the license agreement, select the respective check box.
8. Click **Finish** to complete the installation.

Note: If there are problems with the plug-in (missing icons, for example), start Eclipse from the command line with the `-clean` flag.

12.2 The UModel Perspective

After you install the UModel plug-in for Eclipse, a new perspective ("UModel") becomes available in Eclipse. By default, the UModel perspective resembles to some extent the graphical user interface of the standalone version of UModel. To switch to the UModel perspective, click **Window | Perspective | Open Perspective | Other**, and choose UModel from the list. The image below illustrates a sample UModel project (BankView.ump) loaded into Eclipse, with the UModel perspective switched on.



The UModel perspective in Eclipse is organized as follows:

- The Diagram window is available as an Eclipse editor. Like in the standalone version, when there are multiple diagrams open, they are shown in individual editors.
- All of the following UModel windows are available as Eclipse views (by default, to the left of the main editor):

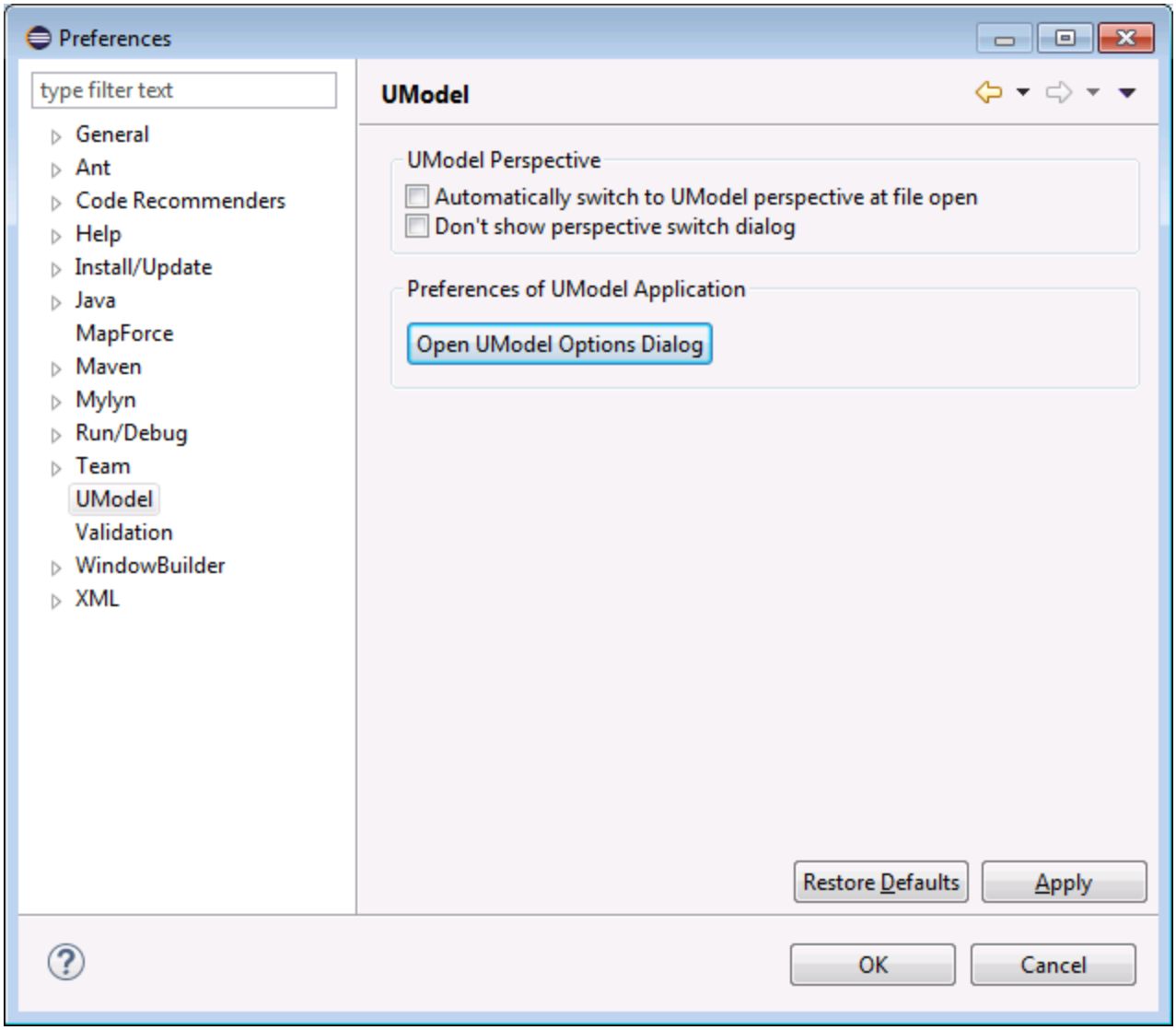
© 2019-2025 Altova GmbH

  - Diagram Tree
  - Favorites
  - Properties
  - Styles
  - Hierarchy
  - Overview
  - Documentation
  - Layer
- Finally, the Messages window is also available as an Eclipse view (by default, under the main editor).

The UModel perspective behaves just like any other Eclipse perspective—you can switch to it whenever required using the menu command **Window | Navigation | Next Perspective**.

To configure the settings applicable to the UModel perspective:

1. On the **Window** menu, click **Preferences**.
2. On the Preferences dialog box, select **UModel**.



To customize the appearance of the UModel perspective (toolbar visibility, menu visibility, and so on), switch to the UModel perspective, and then select the menu command **Window | Perspective | Customize Perspective**. To revert to the default settings, select **Window | Perspective| Reset Perspective**.

To display a particular view in the UModel perspective, switch to the UModel perspective, and then select the required view from the **Window | Show View** menu.

For general information about Eclipse perspectives, refer to the Eclipse documentation.

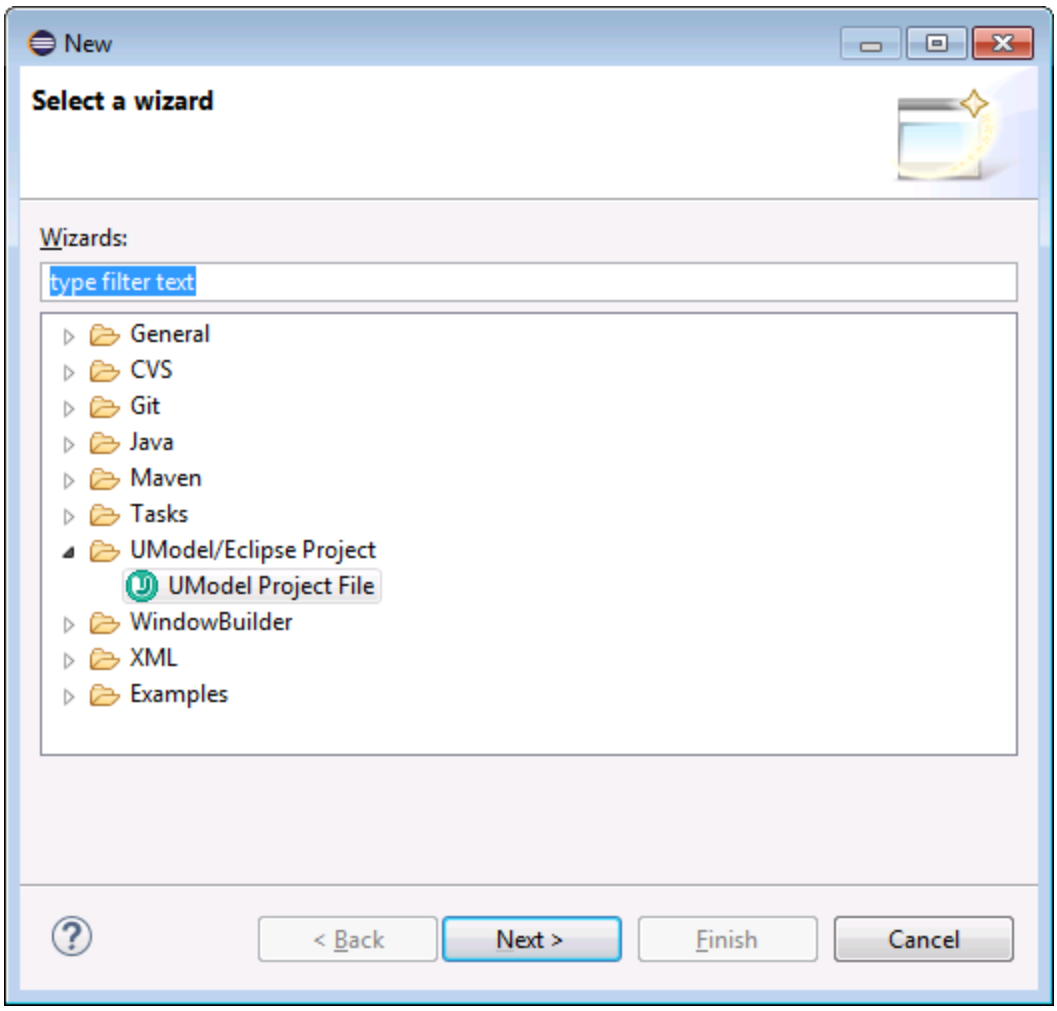
© 2019-2025 Altova GmbH

12.3 Adding UModel Support to Eclipse Projects

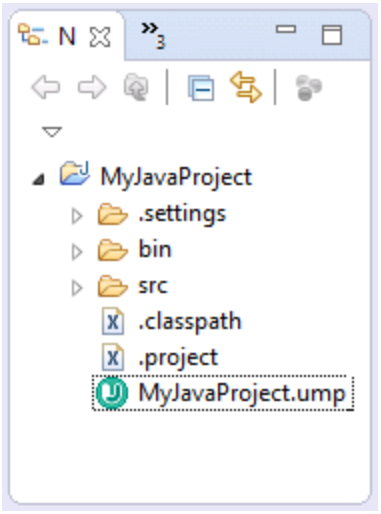
Before you can work with UModel projects (.ump file) in the Eclipse environment, make sure to create or open an Eclipse project first (this can be, for example, a new or existing Java project to which you would like to add UML support). This topic shows you how to create a new UModel project within an Eclipse project. For instructions on how to import an existing UModel project into an Eclipse project, see Importing Existing UModel Projects 591.

To add a UModel project to an Eclipse project:

1. Create a new (or open an existing) Eclipse project, by using the standard Eclipse commands (**File | New | Project**, or **File | Open File**).
2. On the **File** menu, click **New | Other**, and then select the **UModel Project File** type from the dialog box.



3. Click **Next**.
4. When prompted, select a parent folder for the new UModel project, and click **Finish**. The new UModel project becomes available in the Navigator view, under the parent folder you specified.



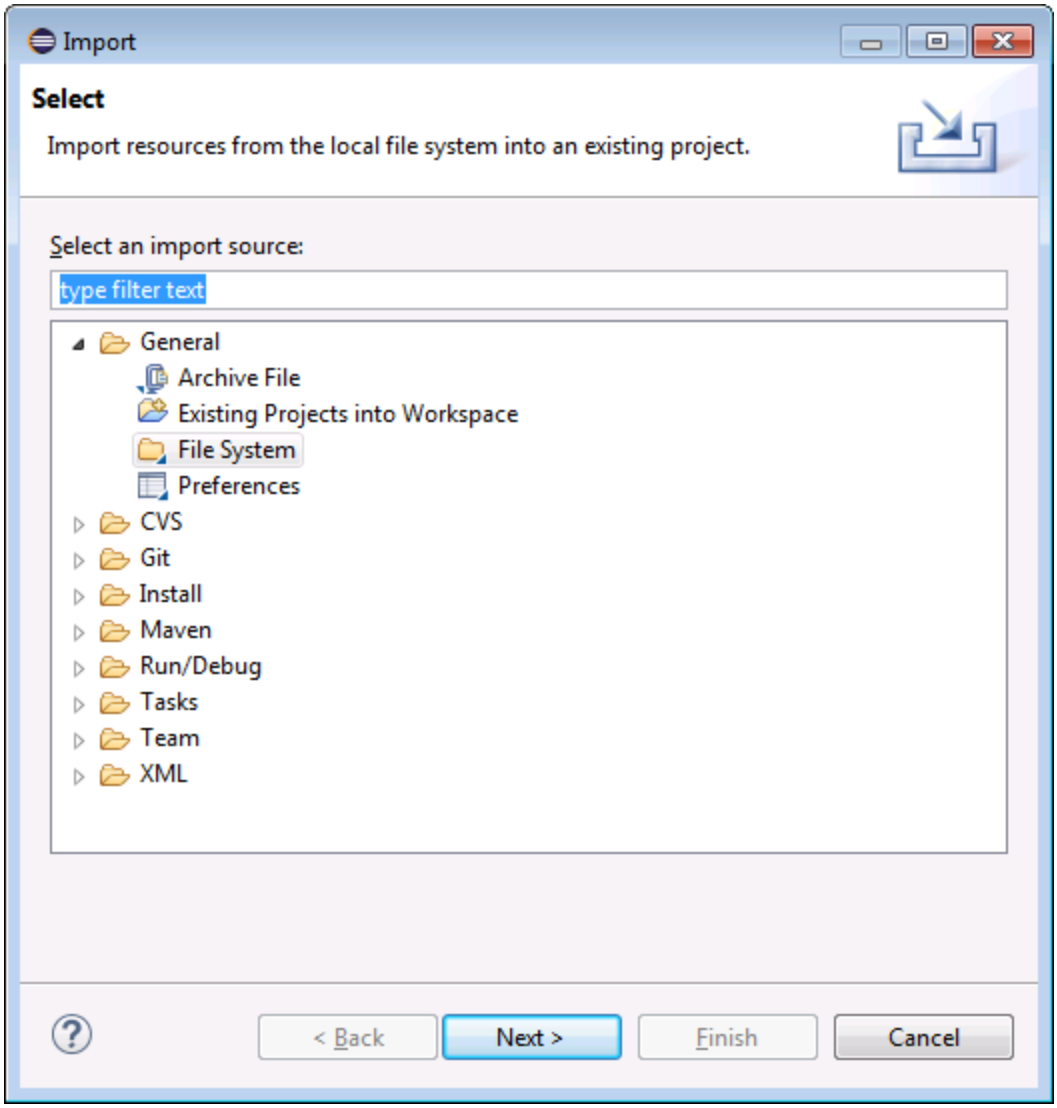
UModel projects cannot be opened in an editor. To take actions against the project (such as saving or loading its contents into Eclipse), right-click the .ump file, and select the required command.

© 2019-2025 Altova GmbH

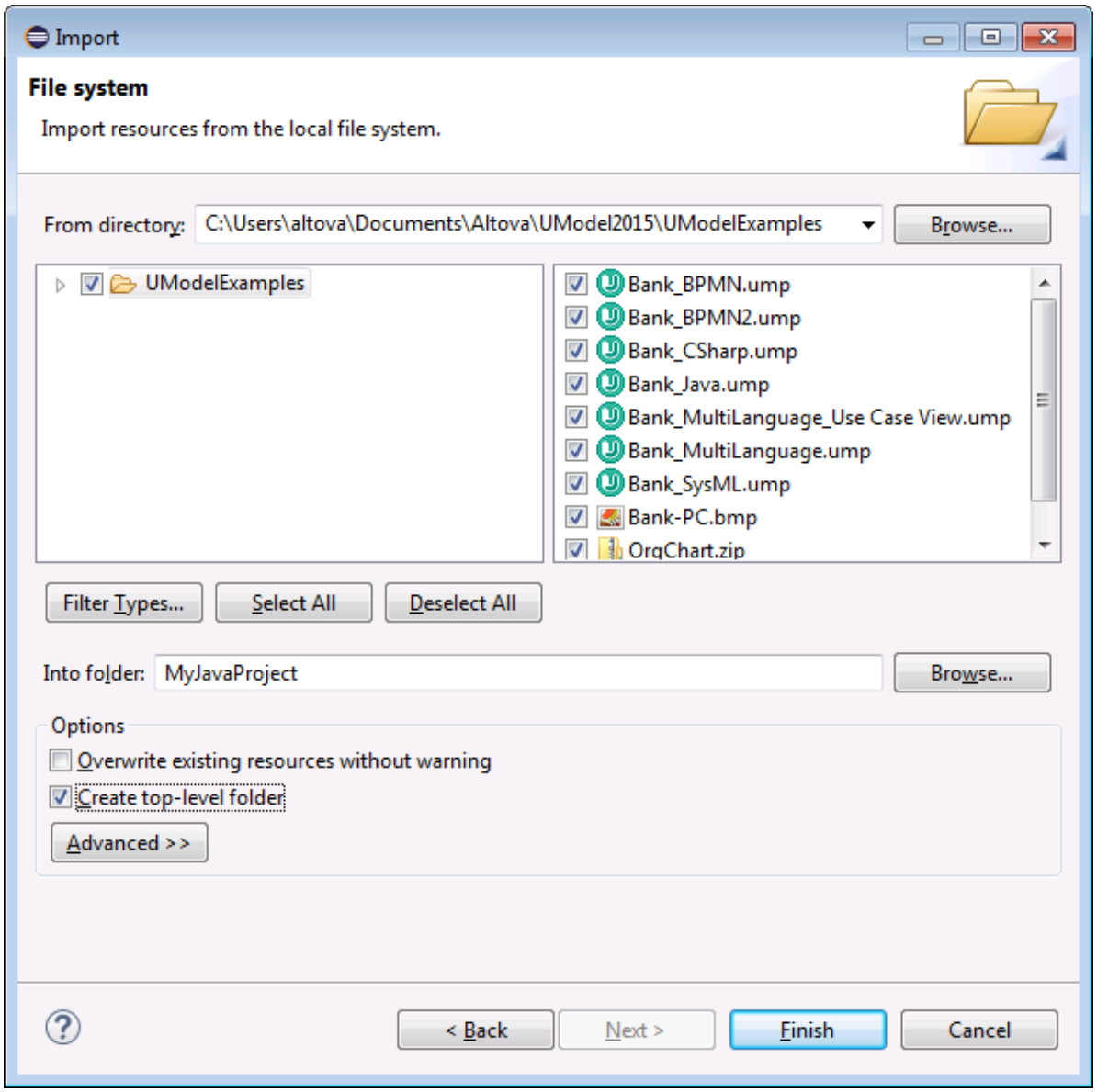
12.4 Importing Existing UModel Projects

To import existing UModel projects into Eclipse:

1. Create a new (or open an existing) Eclipse project.
2. On the **File** menu, click **Import**.
3. Select **General | File System**.



4. Click **Next**.
5. Click **Browse** and select the UModel project folders you want to import (for example, the UModel Examples folder).



6. Click **Finish**.

© 2019-2025 Altova GmbH

12.5 Loading/Unloading UModel Projects

After you have created or imported one or more UModel project files, they appear in the Navigator view of Eclipse. Even though an Eclipse project can contain multiple UModel project files, only one UModel project can be active (loaded) at a time in Eclipse. You can load a specific project as follows:

- Right-click the file in the Navigator view, select **UModel | Load**.
- In the UModel toolbar, select **Load YourProjectName.ump**.

To unload a project:

- Right-click the file in the Navigator view, select **UModel | Unload**.
- In the UModel toolbar, select **Unload project**.

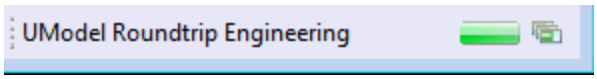
12.6 How Automatic Synchronization Works

Automatic synchronization takes place after you add UModel support to a Java project (see Adding UModel Support to Eclipse Projects 589). Automatic synchronization means that, whenever you edit the code in the Eclipse environment, the UModel Plug-in for Eclipse parses the code and updates the model. Likewise, if you make changes to a diagram in the model, the code is updated accordingly.

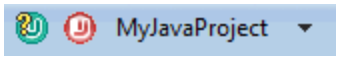
If your UModel project contains the language profile for C# or Visual Basic, then automatic synchronization is automatically disabled for that project, and a message box informs you of this. Such projects must be synchronized manually (using the menu commands **UModel | Merge Program Code from UModel Project**, and **UModel | Merge UModel Project from Program Code**).

Automatic or manual synchronization updates changes in bulk, for the entire project. The option to merge or update a single class is not available in the Model Tree.

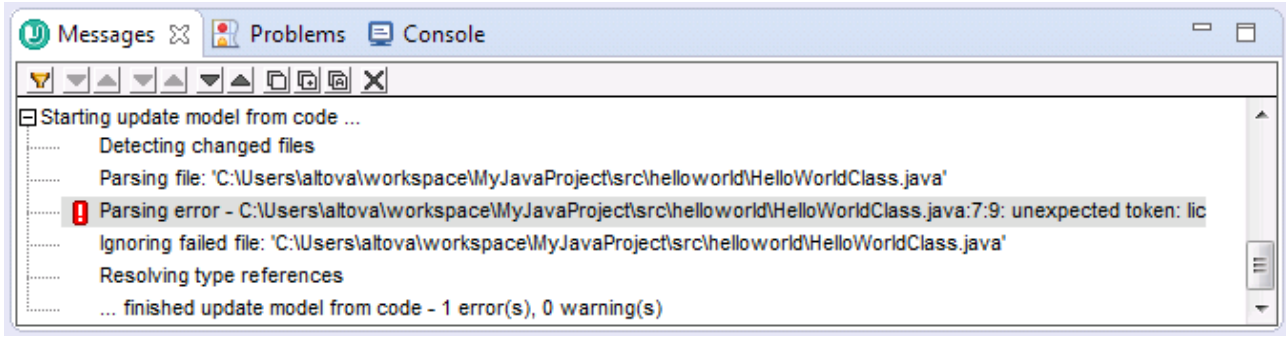
During synchronization, the progress of the operation appears in the Eclipse status bar.



If code is not parseable then the Code Engineering Status tool bar button turns red. This also happens if the last reverse engineering or forward engineering process encountered an error. The same is true if the syntax check throws an error in UModel.



The Messages view displays the error details.



To open the source file which contains the error, click the corresponding line in the Messages view. The cursor will be positioned on the line containing the error

 © 2019-2025 Altova GmbH

12.7 Example: Setting up Automatic Synchronization

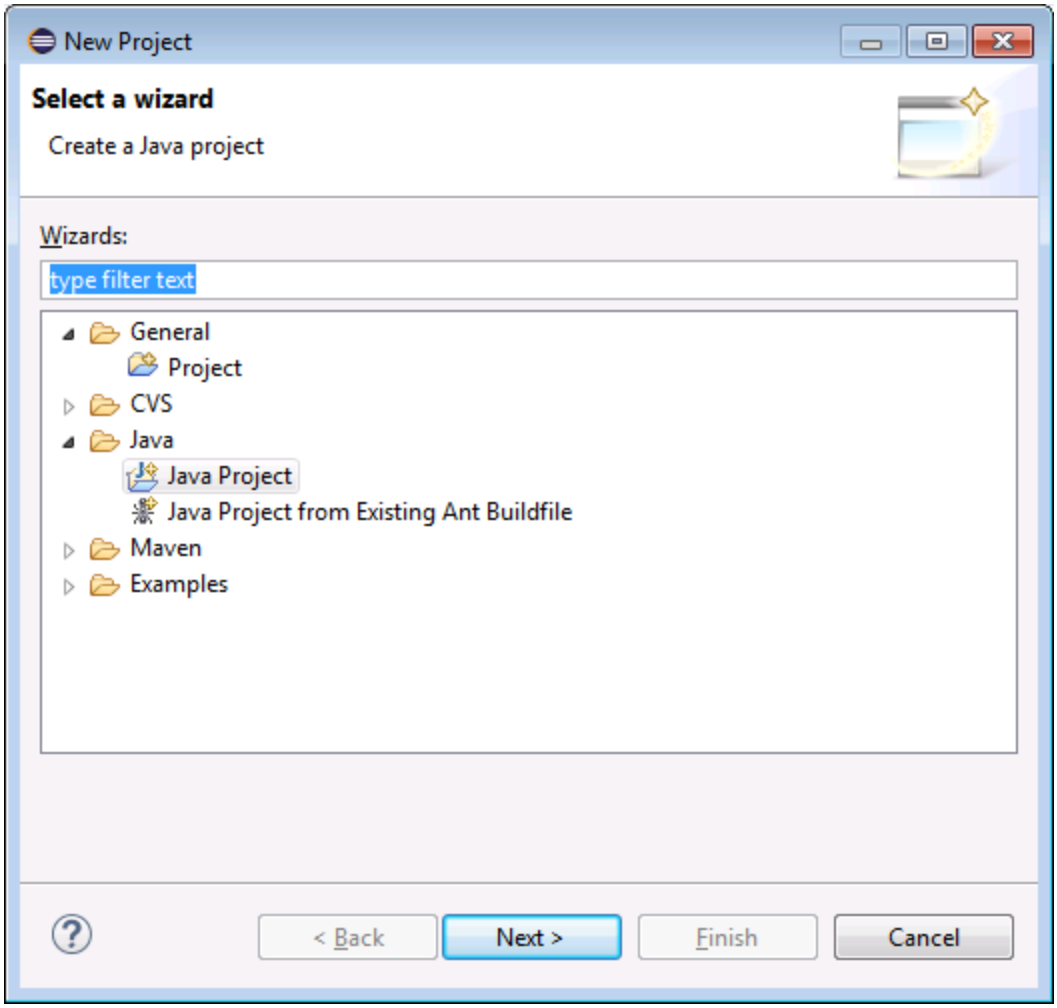
This tutorial illustrates how to set up automatic synchronization between a Java project and its corresponding UML model. Before you proceed, make sure that you have already installed the UModel plug-in for Eclipse, and the Java Development Kit (not just the Java Runtime Environment) required by Eclipse.

Step 1: Create a new Java project

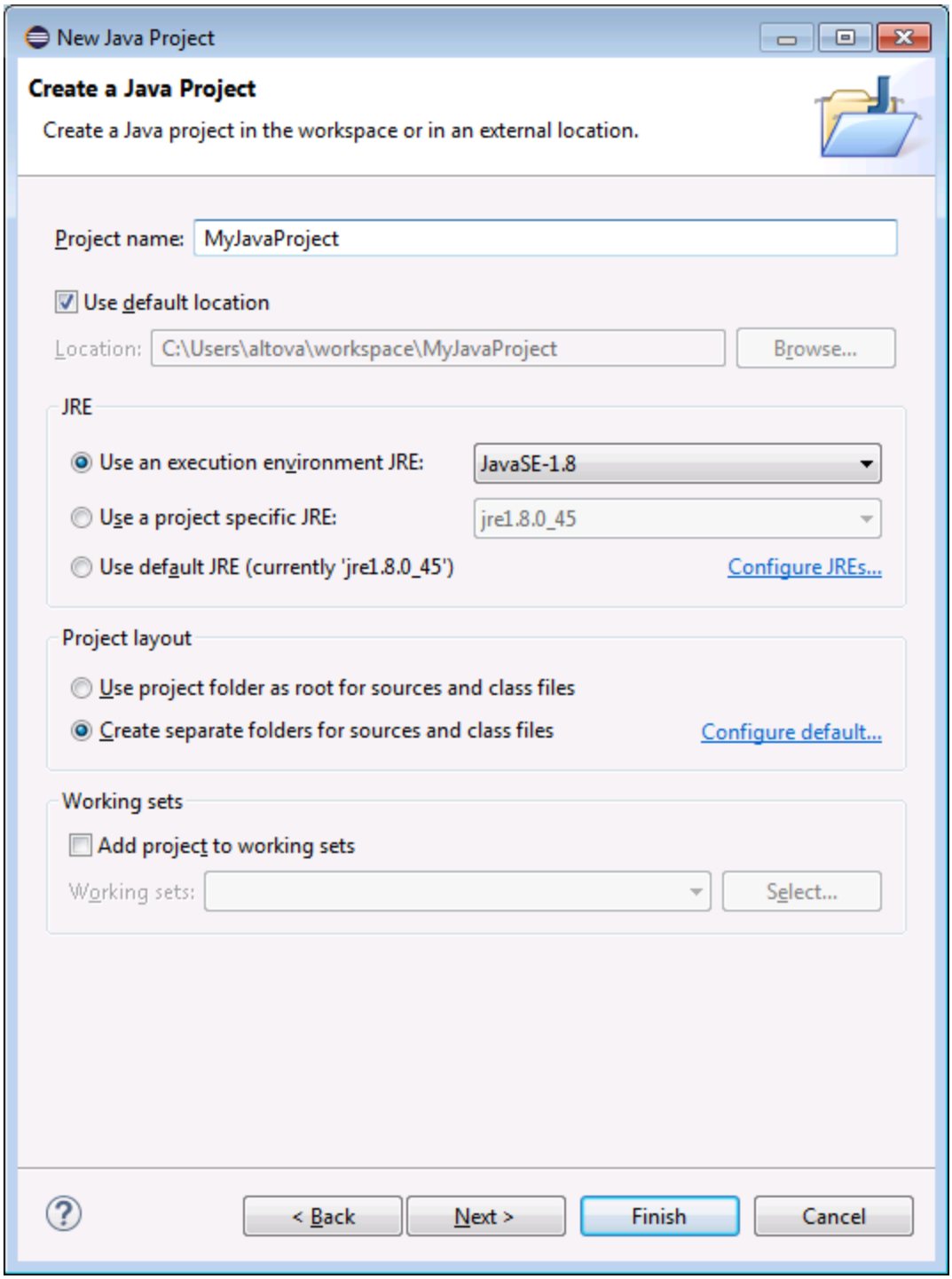
We will begin by creating a new Java project in Eclipse. For the scope of this example, this will be a simple application that displays the text "Hello, World" when it is run.

To create the "Hello, World" application:

1. Start Eclipse and switch to the Java perspective.
2. On the **File** menu, click **New | Project**.

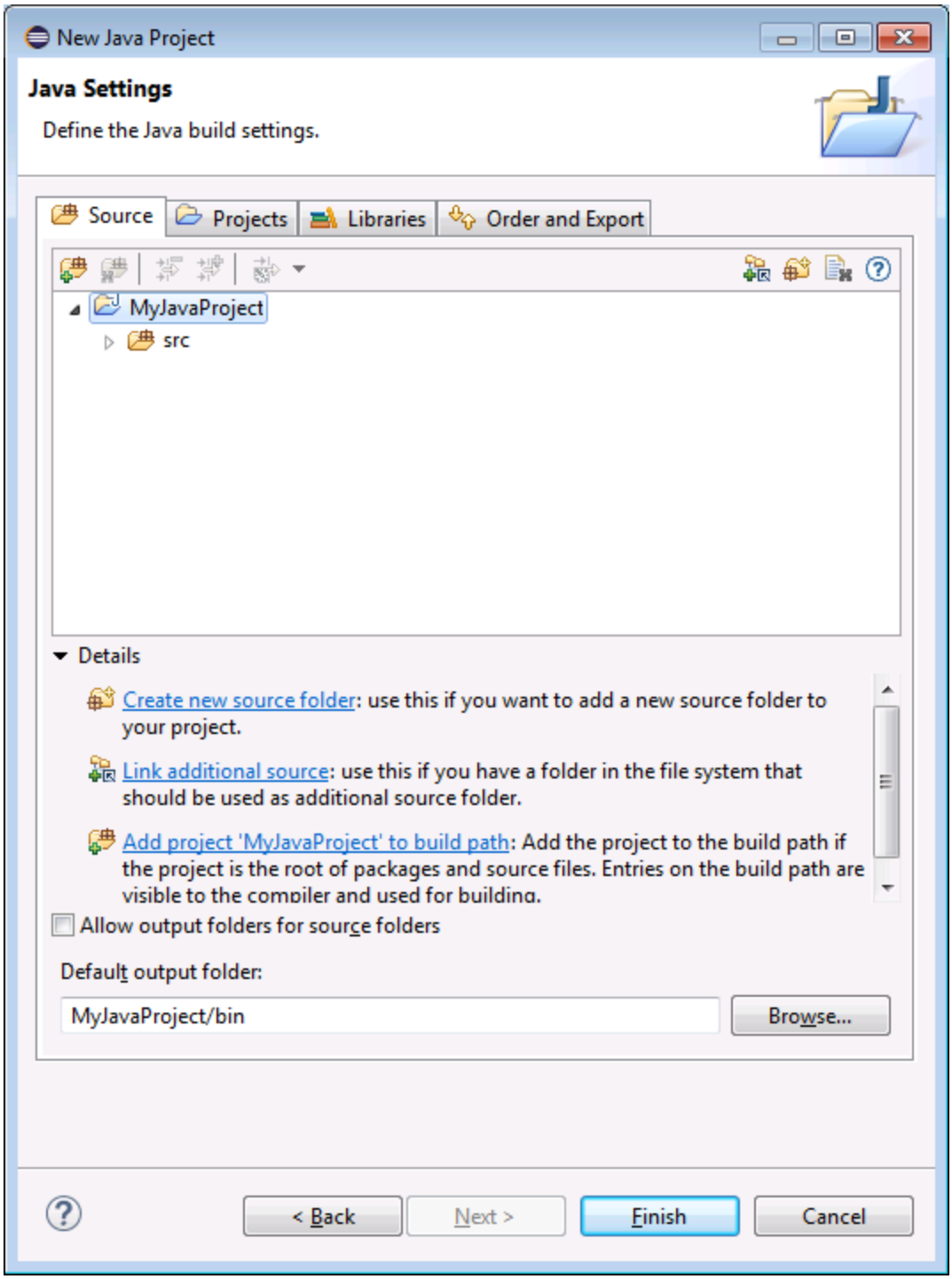


3. Select **Java | Java Project**, and then click **Next**.

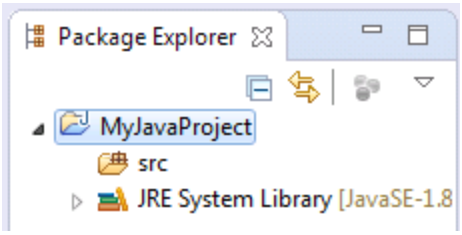


4. Enter "MyJavaProject" as project name, and then click **Next**.

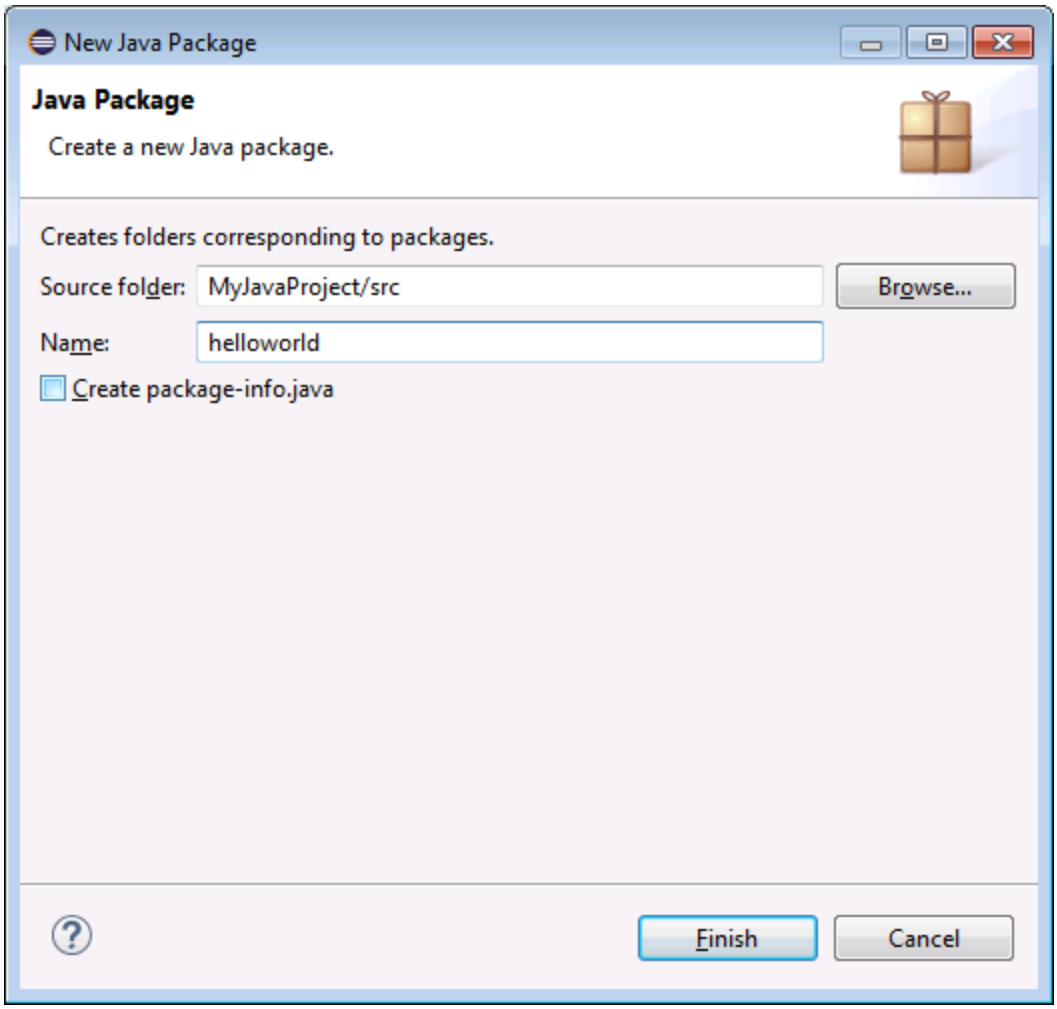
© 2019-2025 Altova GmbH



5. Leave the default settings as is, and click **Finish**. Your project now appears in the Package Explorer.

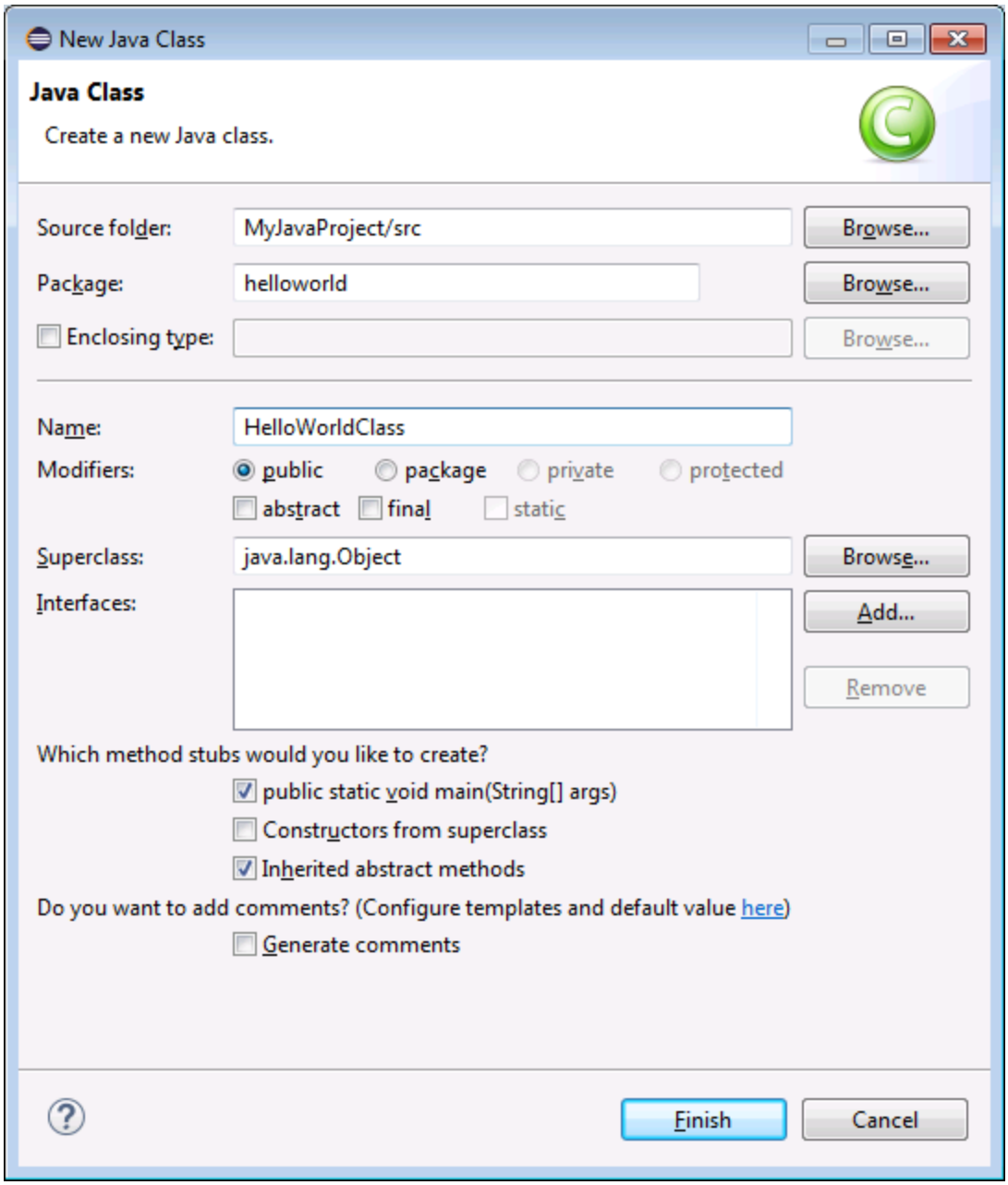


6. On the **File** menu, click **New | Package**.



7. Enter "helloworld" as package name, and click **Finish**.
8. On the **File** menu, click **New | Class**. Enter "HelloWorldClass" as class name, and make sure to select the **public static_void main(String[] args)** option.

© 2019-2025 Altova GmbH

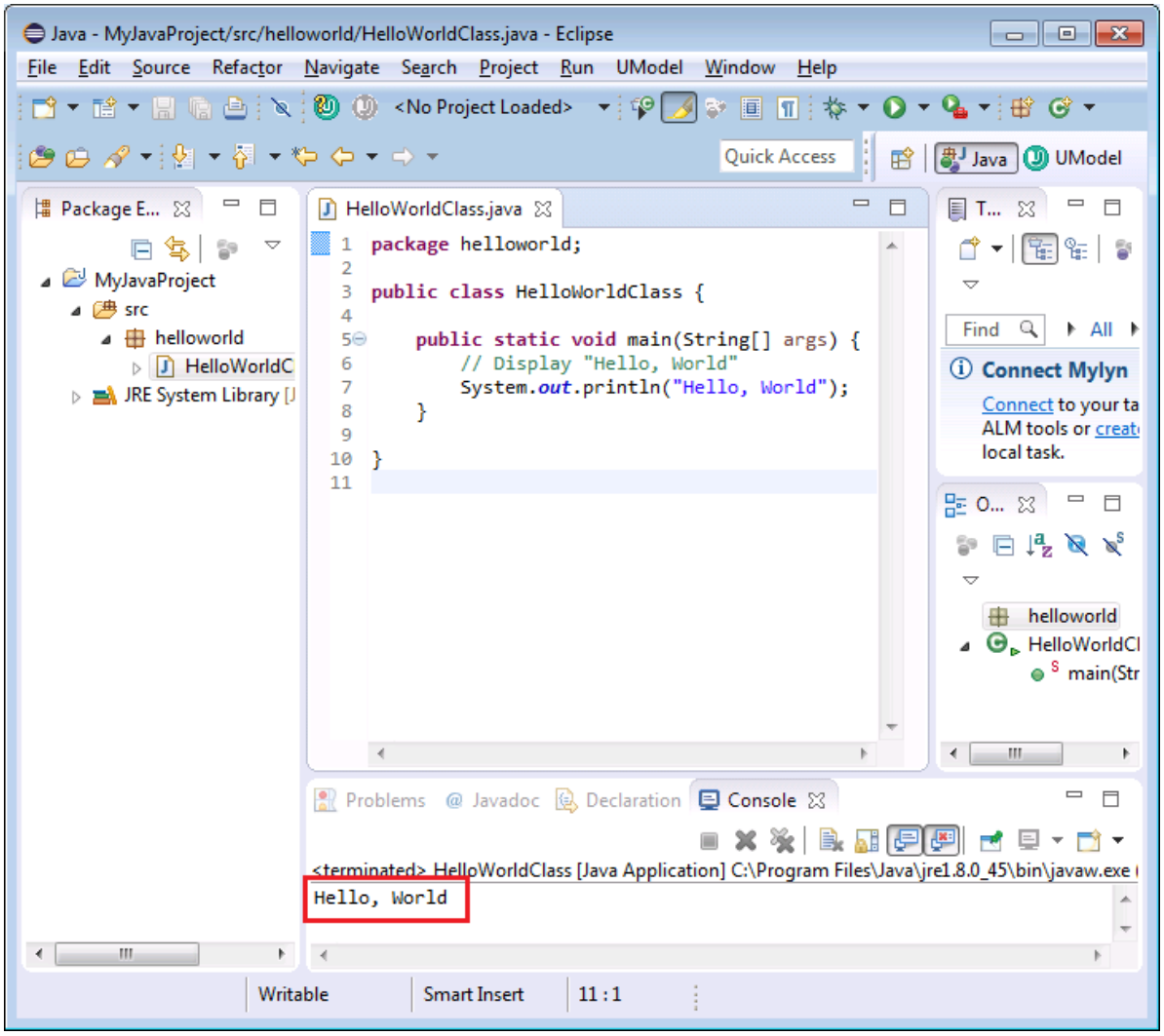


9. Open the class file, and add the following text to the body of the class:

```
package helloworld;

public class HelloWorldClass {

   public static void main(String[] args) {
      // Display "Hello, World"
      System.out.println("Hello, World");
   }
```

```
}
```

10. Run the application. The Console view displays the text "Hello, World", as shown below.

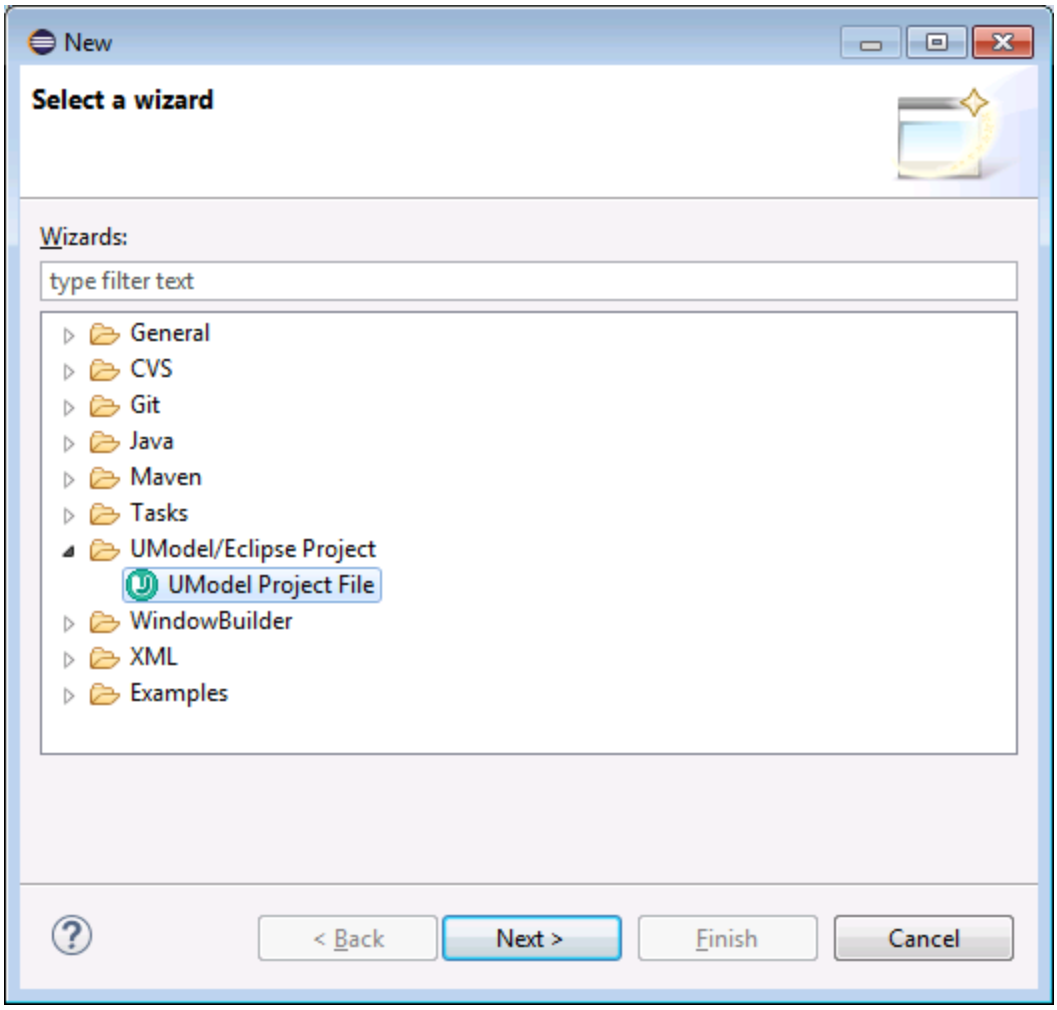


Step 2: Add the UModel project to the Java project

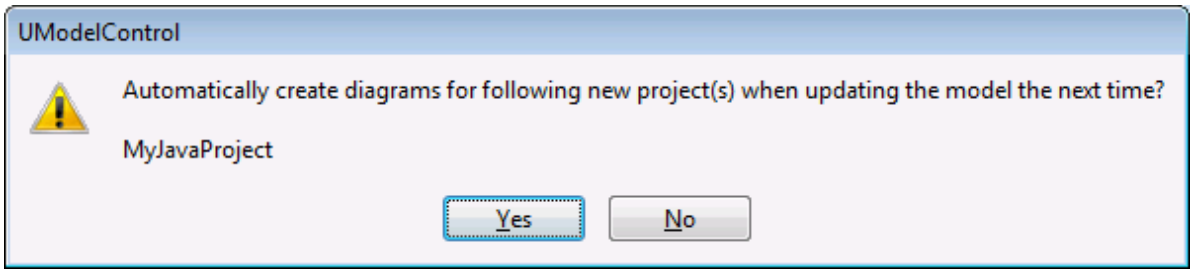
It is now time to add the UModel project file to the Eclipse project. This will create a synchronization relationship between the model and the code.

1. On the **File** menu, click **New | Other**, and select **UModel Project File**.

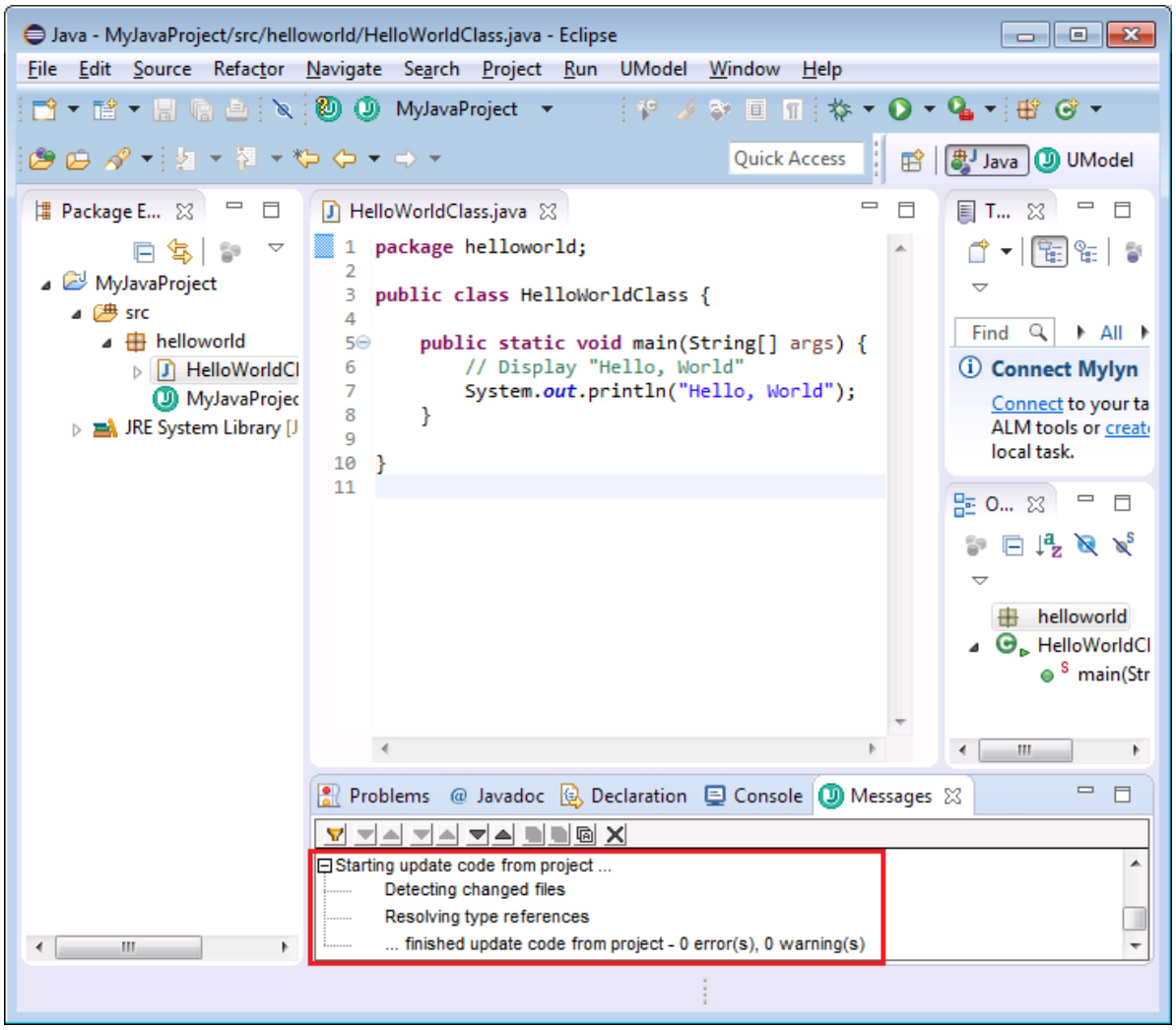
© 2019-2025 Altova GmbH



2. Click **Next**. When prompted to specify a location for the new UModel project, leave the default settings as is, and then click **Finish**.
3. When prompted by UModel to create diagrams for the project, click **Yes**.



4. Go through the wizard steps, leaving the default settings as is. When you click **Finish**, the new UModel project is added to the Eclipse project, and synchronization of the code with the model takes place automatically. Notice the messages displayed in the Messages view of UModel.



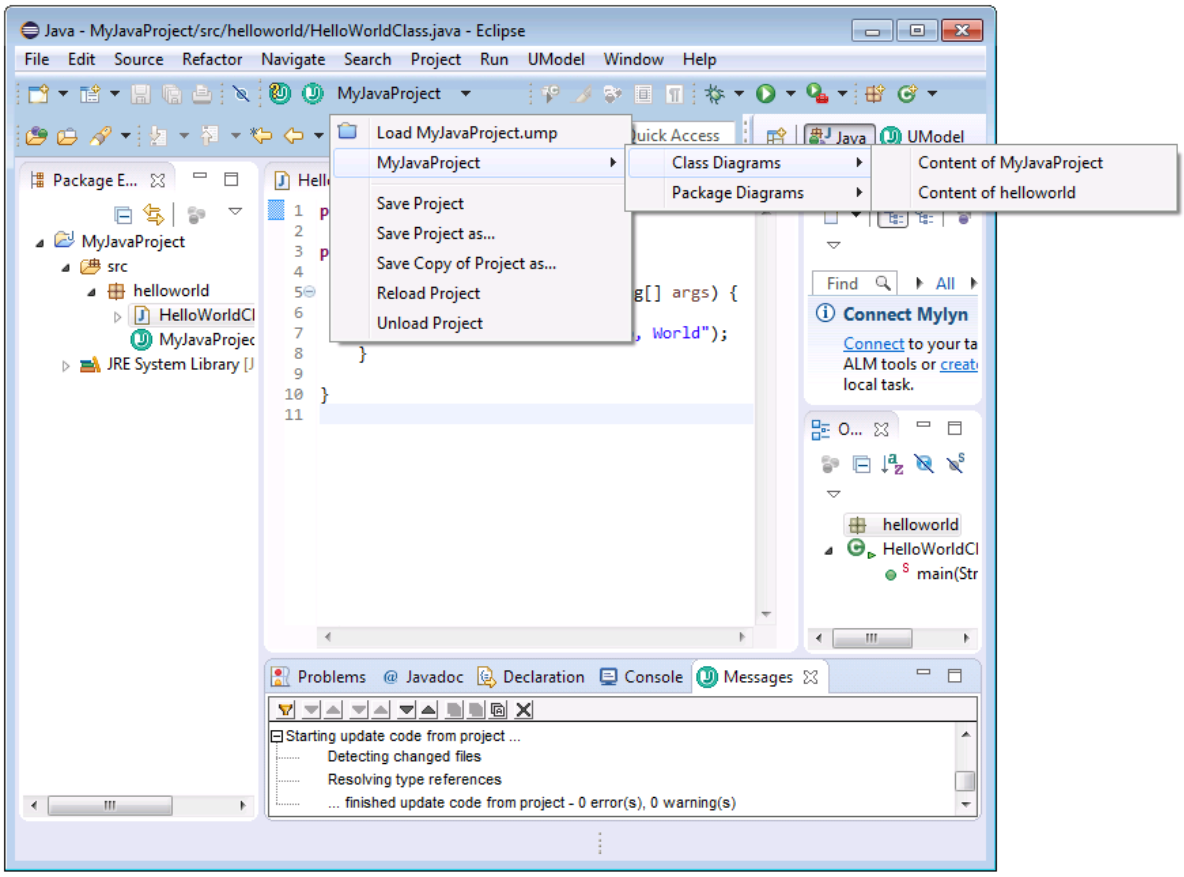
Step 3: Trigger automatic synchronization from model to code

To trigger automatic synchronization from model to code, we will make some changes to the class diagram in the model. Namely, we will add to the class a new property called "Property1" of type "Boolean".

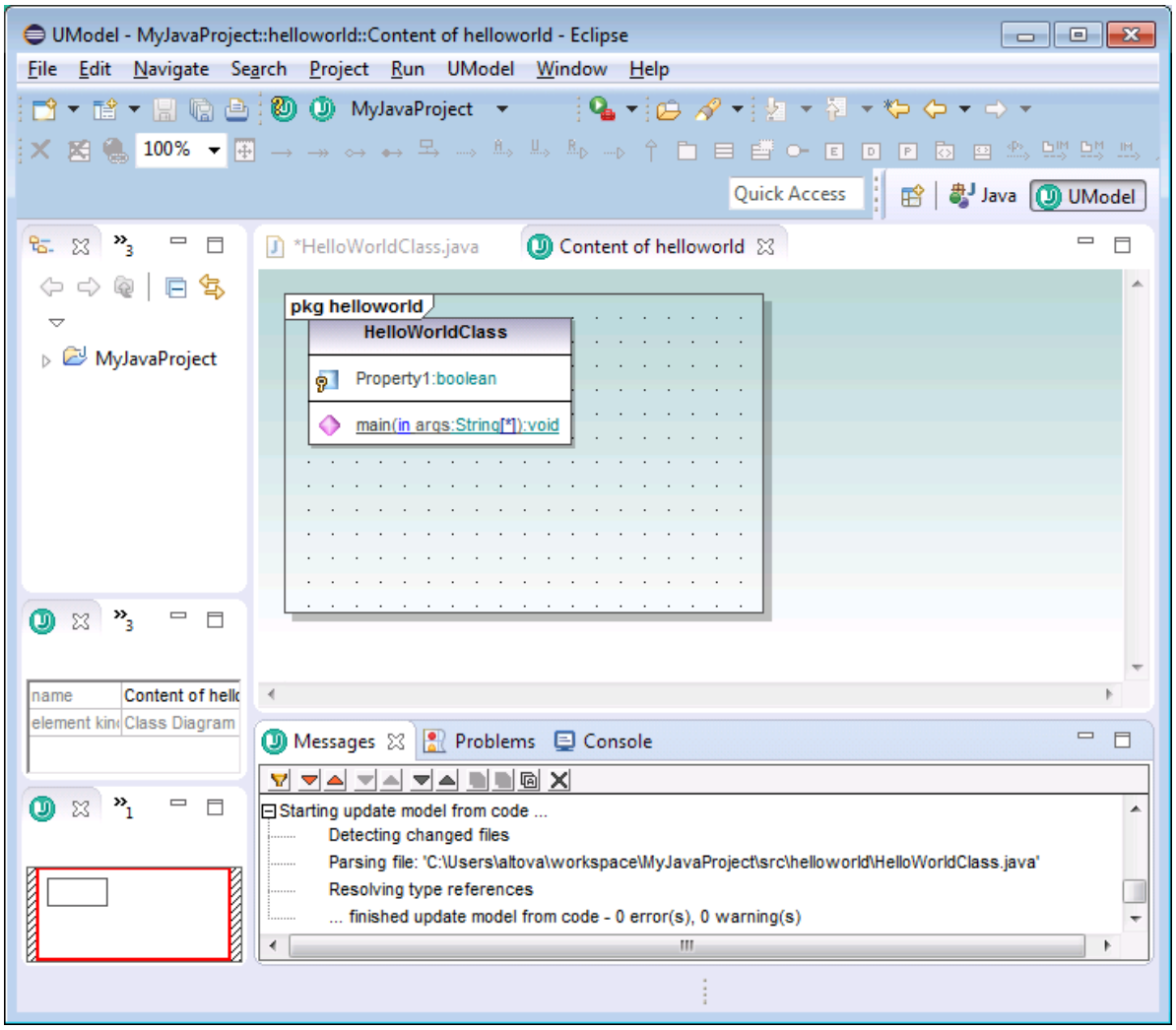
To add the property to the class:

1. In the UModel toolbar, expand the project drop-down list, and open the generated "Content of helloworld" class diagram.

© 2019-2025 Altova GmbH

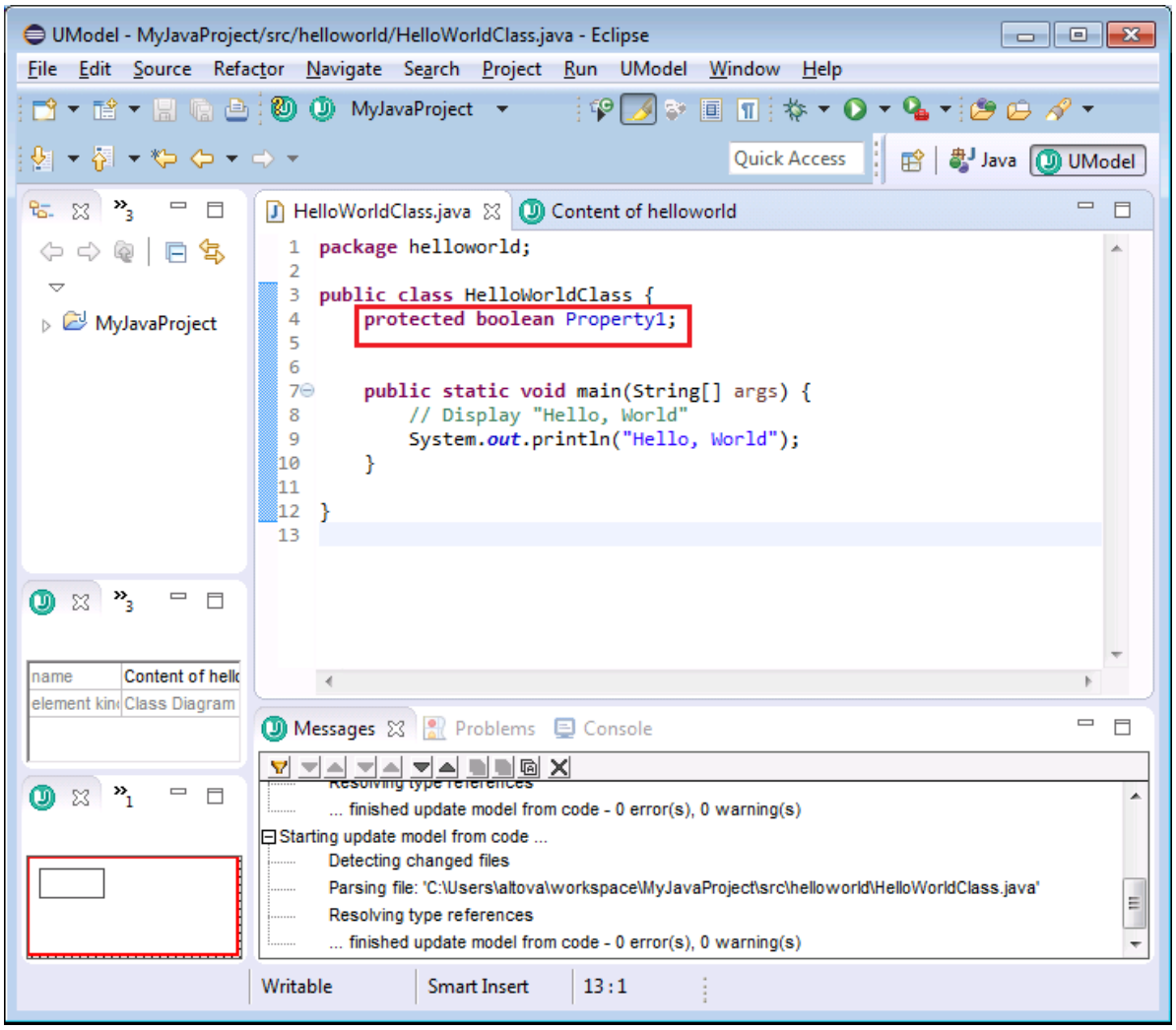


2. Right-click the class, and select **New | Property** from the context menu.
3. Type the property name ("Property1"), followed by the colon character (:), followed by the type ("boolean").



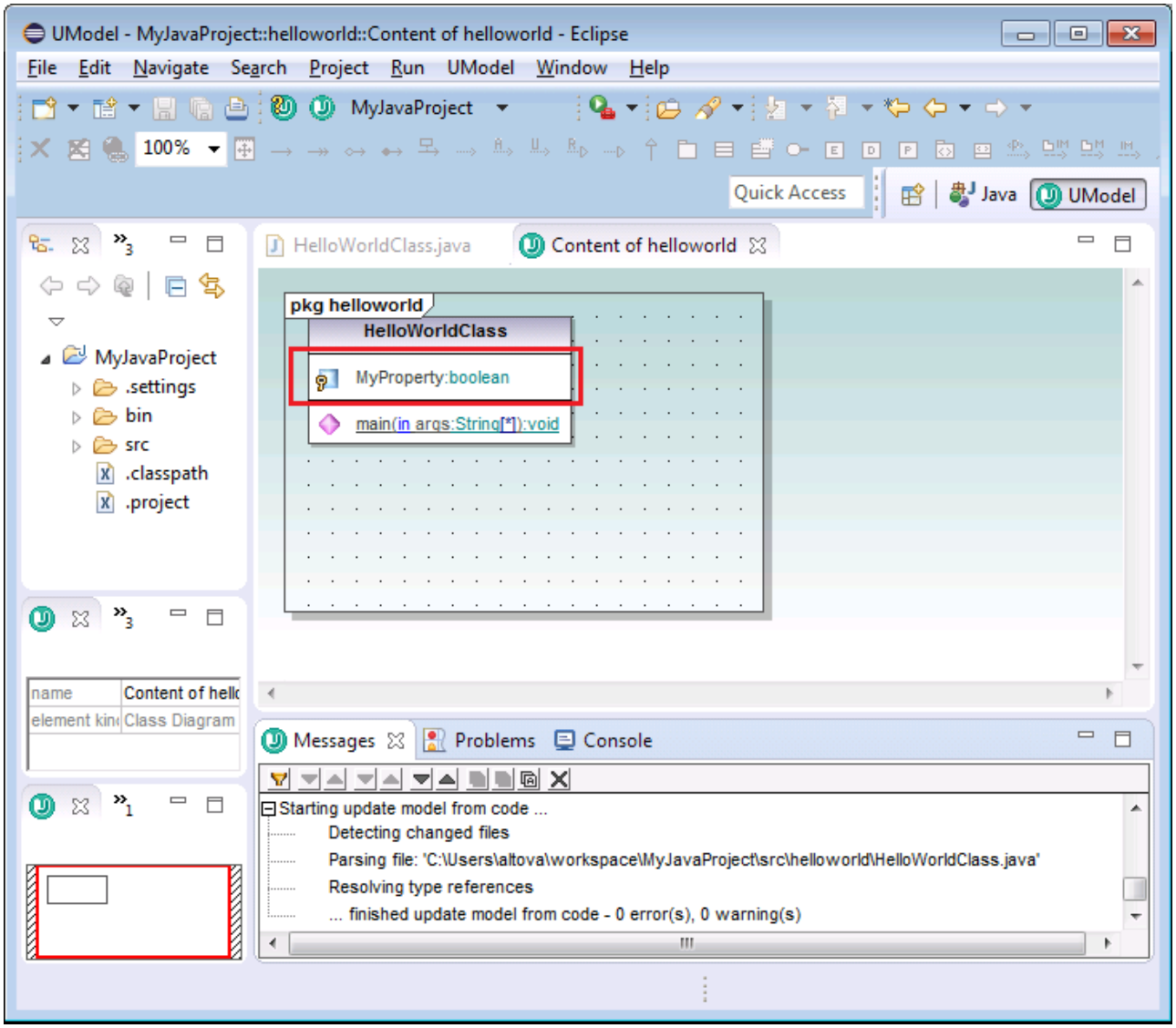
4. Switch back to the code editor. Notice that the newly added property is now reflected in the code.

© 2019-2025 Altova GmbH



Step 4. Trigger automatic synchronization from code to model

Let's now trigger automatic synchronization of changes in the opposite direction (from code to model). To do this, change in the code the name of the "Property1" property to "MyProperty", and then save the project. Notice that the changes are now reflected in the diagram.



© 2019-2025 Altova GmbH

13 Source Control

The source control support in UModel is available through the Microsoft Source Control Plug-in API (formerly known as the MSSCCI API), versions 1.1, 1.2 and 1.3. This enables you to run source control commands such as "Check in" or "Check out" directly from UModel to virtually any source control system that lets native or third-party clients connect to it through the Microsoft Source Control Plug-in API.

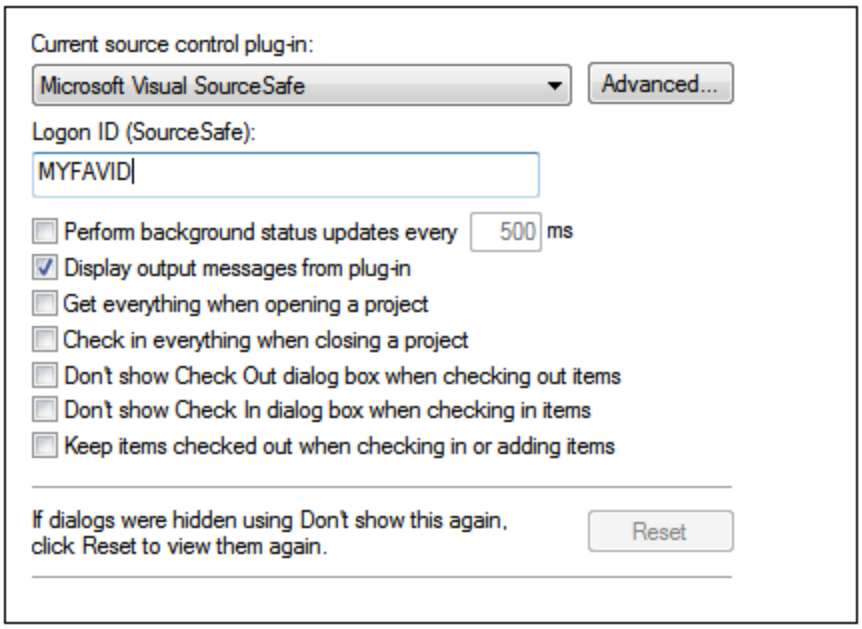
You can use as your source control provider any commercial or non-commercial plug-in that supports the Microsoft Source Control Plug-in API, and can connect to a compatible version control system. For the list of source control systems and plug-ins tested by Altova, see Supported Source Control Systems 610.

Installing and configuring the source control provider

To view the source control providers available on your system, do the following:

1. On the **Tools** menu, click **Options**.
2. Click the **Source Control** tab.

Any source control plug-ins compatible with the Microsoft Source Code Control Plug-in API are displayed in the **Current source control plug-in** drop-down list.



If a compatible plug-in cannot be found on your system, the following message is displayed:

```
"Registration of installed source control providers could not be found or is incomplete."
```

Some source control systems might not install the source control plug-in automatically, in which case you will need to install it separately. For further instructions, refer to the documentation of the respective source control system. A plug-in (provider) compatible with the Microsoft Source Code Control Plug-in API is expected to be registered under the following registry entry on your operating system:

```
HKEY_LOCAL_MACHINE\SOFTWARE\SourceCodeControlProvider\InstalledSCCProviders
```

Upon correct installation, the plug-in becomes available automatically in the list of plug-ins available to UModel.

Accessing the source control commands

The commands related to source control are available in the **Project | Source Control** menu.

Resource / Speed issues

Very large source control databases might be introducing a speed/resource penalty when automatically performing background status updates.

You might be able to speed up your system by disabling (or increasing the interval of) the **Perform background status updates every ... seconds** option in the **Source Control** tab accessed through **Tools | Options**.

Note: The **64-bit** version of your Altova application automatically supports any of the supported 32-bit source control programs listed in this documentation. When using a 64-bit Altova application with a 32-bit source control program, the **Perform background status updates every ... seconds** option is automatically grayed-out and cannot be selected.

Differencing with Altova DiffDog

You can configure many source control systems (including Git and TortoiseSVN) so that they use Altova DiffDog as their differencing tool. For more information about DiffDog, see https://www.altova.com/diffdog. For DiffDog documentation, see https://www.altova.com/documentation.html.

© 2019-2025 Altova GmbH

13.1 Setting Up Source Control

The mechanism for setting up source control and placing files in a UModel project under source control is as follows:

1. If this hasn't been done already, install the source control system (see Supported Source Control Systems 610) and set up the source control database (repository) to which you wish to save your work.
2. Create a local workspace folder that will contain the working files that you wish to place under source control. The folder that contains all your workspace folders and files is called the local folder, and the path to the local folder is referred to as the local path. This local folder will be bound to a particular folder in the repository.
3. In your Altova application, create an application project folder to which you must add the files you wish to place under source control. This organization of files in an application project is abstract. The files in a project reference physical files saved locally, preferably in one folder (with sub-folders if required) for each project.
4. In the source control system's database (also referred to as source control or repository), a folder is created that is bound to the local folder. This folder (called the bound folder) will replicate the structure of the local folder so that all files to be placed under source control are correctly located hierarchically within the bound folder. The bound folder is usually created when you add a file or an application project to source control for the first time.

13.2 Supported Source Control Systems

The list below shows the Source Control Servers (SCSs) supported by UModel, together with their respective Source Control Clients (SCCs). The list is organized alphabetically by SCS. Note the following:

- Altova has implemented the Microsoft Source Control Plug-in API (versions 1.1, 1.2, and 1.3) in UModel, and has tested support for the listed drivers and revision control systems. It is expected that UModel will continue to support these products if, and when, they are updated.
- Source Code Control clients not listed below, but which implement the Microsoft Source Control Plug-in API, should also work with UModel.

Source Control System	Source Code Control Clients
AccuRev 4.7.0 Windows	AccuBridge for Microsoft SCC 2008.2
Bazaar 1.9 Windows	Aigenta Unified SCC 1.0.6
Borland StarTeam 2008	Borland StarTeam Cross-Platform Client 2008 R2
Codice Software Plastic SCM Professional 2.7.127.10 (Server)	Codice Software Plastic SCM Professional 2.7.127.10 (SCC Plugin)
Collabnet Subversion 1.5.4	• Aigenta Unified SCC 1.0.6 • PushOK SVN SCC 1.5.1.1 • PushOK SVN SCC x64 version 1.6.3.1 • TamTam SVN SCC 1.2.24
ComponentSoftware CS-RCS (PRO) 5.1	ComponentSoftware CS-RCS (PRO) 5.1
Dynamsoft SourceAnywhere for VSS 5.3.2 Standard/Professional Server	Dynamsoft SourceAnywhere for VSS 5.3.2 Client
Dynamsoft SourceAnywhere Hosted	Dynamsoft SourceAnywhere Hosted Client (22252)
Dynamsoft SourceAnywhere Standalone 2.2 Server	Dynamsoft SourceAnywhere Standalone 2.2 Client
Git	PushOK GIT SCC plug-in (see Source Control with Git 632)
IBM Rational ClearCase 7.0.1 (LT)	IBM Rational ClearCase 7.0.1 (LT)
March-Hare CVSNT 2.5 (2.5.03.2382)	Aigenta Unified SCC 1.0.6
March-Hare CVS Suite 2008	• Jalindi Igloo 1.0.3 • March-Hare CVS Suite Client 2008 (3321) • PushOK CVS SCC NT 2.1.2.5 • PushOK CVS SCC x64 version 2.2.0.4 • TamTam CVS SCC 1.2.40
Mercurial 1.0.2 for Windows	Sergey Antonov HgSCC 1.0.1
Microsoft SourceSafe 2005 with CTP	Microsoft SourceSafe 2005 with CTP

© 2019-2025 Altova GmbH

Source Control System	Source Code Control Clients
Microsoft Visual Studio Team System 2008/2010 Team Foundation Server	Microsoft Team Foundation Server 2008/2010 MSSCCI Provider
Perforce 2008 P4S 2008.1	Perforce P4V 2008.1
PureCM Server 2008/3a	PureCM Client 2008/3a
QSC Team Coherence Server 7.2.1.35	QSC Team Coherence Client 7.2.1.35
Reliable Software Code Co-Op 5.1a	Reliable Software Code Co-Op 5.1a
Seapine Surround SCM Client/Server for Windows 2009.0.0	Seapine Surround SCM Client 2009.0.0
Serena Dimensions Express/CM 10.1.3 for Win32 Server	Serena Dimensions 10.1.3 for Win32 Client
Softimage Alienbrain Server 8.1.0.7300	Softimage Alienbrain Essentials/Advanced Client 8.1.0.7300
SourceGear Fortress 1.1.4 Server	SourceGear Fortress 1.1.4 Client
SourceGear SourceOffsite Server 4.2.0	SourceGear SourceOffsite Client 4.2.0 (Windows)
SourceGear Vault 4.1.4 Server	SourceGear Vault 4.1.4 Client
VisualSVN Server 1.6	• Aigenta Unified SCC 1.0.6 • PushOK SVN SCC 1.5.1.1 • PushOK SVN SCC x64 version 1.6.3.1 • TamTam SVN SCC 1.2.24

13.3 Source Control Commands

The following sections use Visual SourceSafe to show the source control features of UModel. The examples in this section use the **Bank_CSharp.ump** UModel project (and associated code files) available in the C:\Users\<username>\Documents\Altova\UModel2025\UModelExamples folder. Note that a Source Control project is not the same as a UModel project. Source Control projects are directory dependent, whereas UModel projects are logical constructions without direct directory dependence.

To access the Source Control commands, do one of the following:

- Use the menu command **Project | Source Control**
- Use the **context** menu in the Model Tree
- Click the source control toolbar buttons in the Source Control toolbar. Use **Tools | Customize | Toolbars** to activate the toolbar.

The description of the version control commands that follow apply to the standalone version of UModel. The Visual Studio and Eclipse versions of UModel use the version control functionality and menu items available in those IDEs.

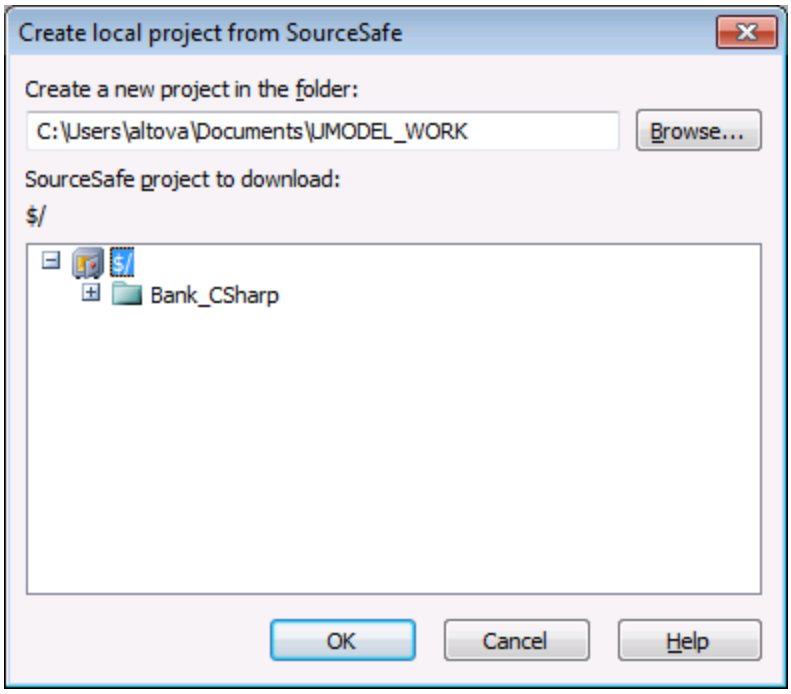
Open from Source Control 612
Enable Source Control 615
Get Latest Version 616
Get 616
Get Folder(s) 617
Check Out 618
Check In 620
Undo Check Out... 620
Add to Source Control 622
Remove from Source Control 624
Share from Source Control 625
Show History 626
Show Differences 628
Show Properties 629
Refresh Status 630
Source Control Manager 630
Change Source Control 630

13.3.1 Open from Source Control

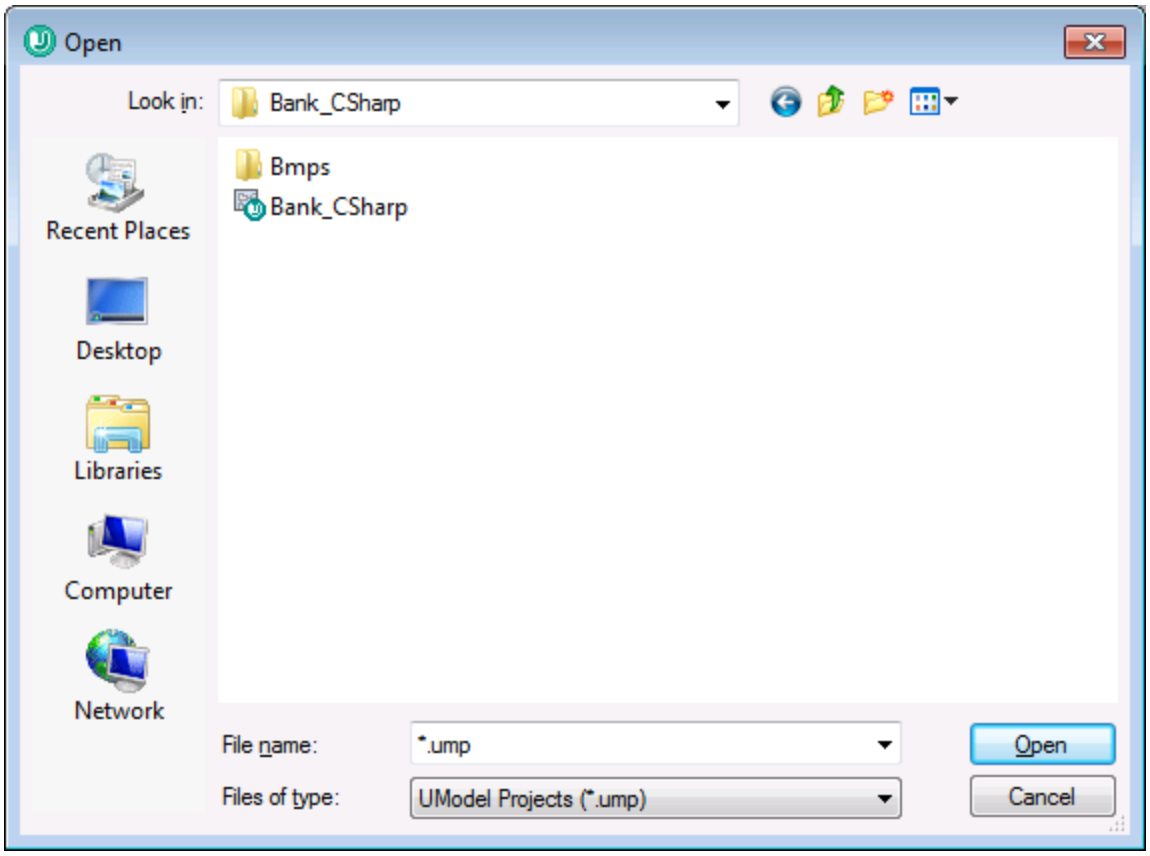
The Open from Source Control command creates a local project from an existing source control database, and places it under source control, SourceSafe in this case.

1. Select **Project | Source Control | Open from Source Control.**
   The Login dialog box is opened, enter your login details to continue.
   The "Create local project from SourceSafe" dialog box appears.
2. Define the directory to contain the new local project e.g. c:\temp\ssc. This becomes the **Working directory**, or the Check Out Folder.

© 2019-2025 Altova GmbH



3. Select the SourceSafe project you want to download e.g. Bank_CSharp.
   If the folder you define here does not exist at the location, a dialog box opens prompting you to create it.
4. Click **Yes** to create the new directory.
   The Open dialog box is now visible.



5. Select the **Bank_CSharp.ump** UModel project file and click Open.

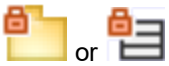
   Bank_CSharp.ump now opens in UModel, and the file is placed under source control. This is indicated by the lock symbol visible on the Root folder in the Model Tree window. The Root folder represents both the project file and the working directory for source control operations.

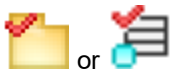
   The BankCSharp directory has been created locally, you can now work with these files as you normally would.

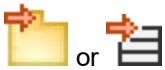
Note:
To place under source control the code files generated when synchronizing code, see: Add to Source Control 622

© 2019-2025 Altova GmbH

Source control symbols


The lock symbol denotes that the file, or folder is under source control, but is currently not checked out.


The red check mark denotes checked out, i.e. the UModel project file (or code file) has been checked out for editing. The asterisk in the Application title bar denotes that changes have been made to the file, and you will be prompted to save it when you exit.


The arrow symbol shows that the file(s) have been checked out by someone else in the network, or by you into a different working directory

13.3.2 Enable Source Control

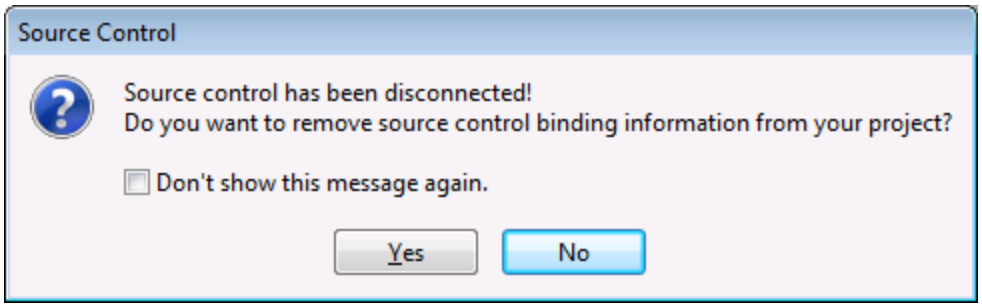
This command allows you to enable or disable source control for a UModel project and is available through the Project menu item, i.e. **Project | Source Control | Enable Source Control**. Selecting this option on any file or folder, enables/disables source control for the whole UModel project.

To enable Source Control for a project:

1. Select the menu option **Project | Source Control** and activate/check the **Enable source control** check box of the fly-out menu. The previous check in/out status of the various files are retrieved and displayed in the Model Tree window.

To disable Source Control for a project:

1. Select the menu option **Project | Source Control** and uncheck the **Enable source control** check box.



You are now prompted if you want to remove the binding information from the project.

To **provisionally** disable source control for the project, select **No**.

To **permanently** disable source control for the project, select **Yes**.

13.3.3 Get Latest Version

Retrieves and places the latest source control version of the selected file(s) in the working directory. The files are retrieved as read-only and are not checked out.

If the affected files are currently checked out, different things occur depending on the specific version control plugin: nothing happens, new data are merged into your local file, or your changes are overwritten.

This command works in a similar fashion to the Get command, but does not display the "Source control - Get" dialog box. It is therefore not possible to specify Advanced get options.

Note that this command automatically performs a recursive get latest version operation when performed on a folder, i.e. it affects all other files below the current one in the package hierarchy.

To get the latest version of a file:

1. Select the file(s) you want to get the latest version of in the Model Tree.
2. Select **Project** | **Source Control** | **Get Latest Version**.

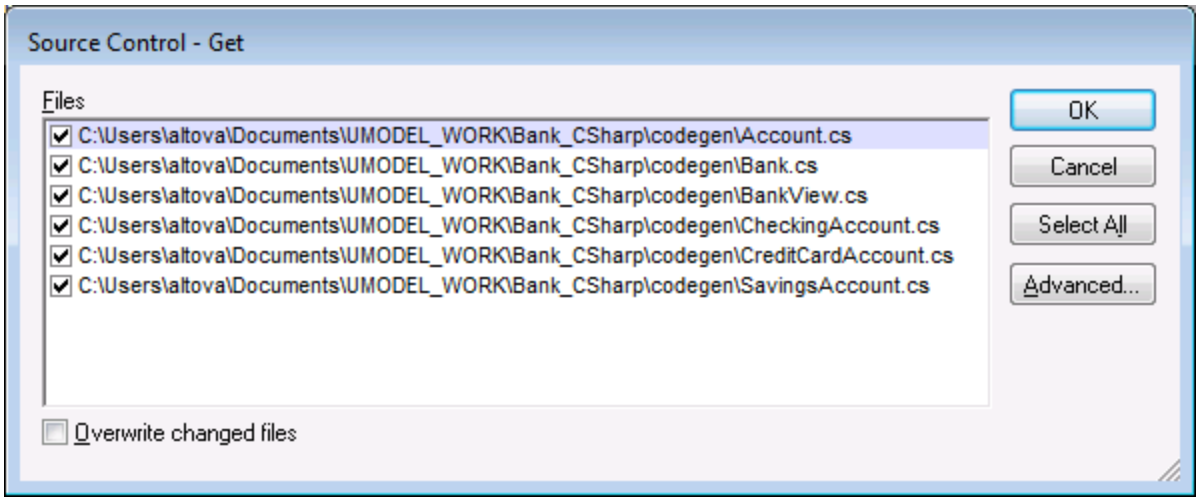
13.3.4 Get

Retrieves a read-only copy of the selected files and places them in the working folder. The files are not checked-out for editing per default.

Using Get:

- Select the files you want to get in the Model Tree.
- Select **Project** | **Source Control** | **Get**.

© 2019-2025 Altova GmbH

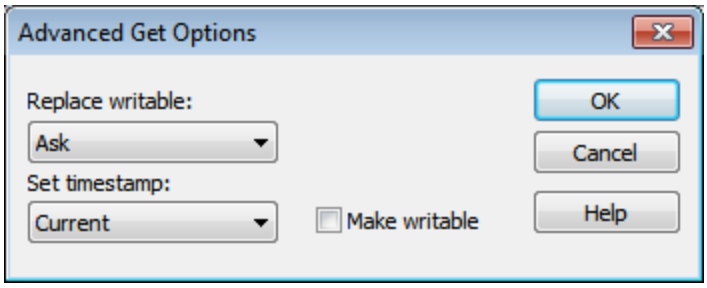


Overwrite changed files
Overwrites those files that have been changed locally with those from the source control database.

Select All
Selects all the files in the list box.

Advanced

Allows you to define the **Replace writable** and **Set timestamp** options in the respective combo boxes.



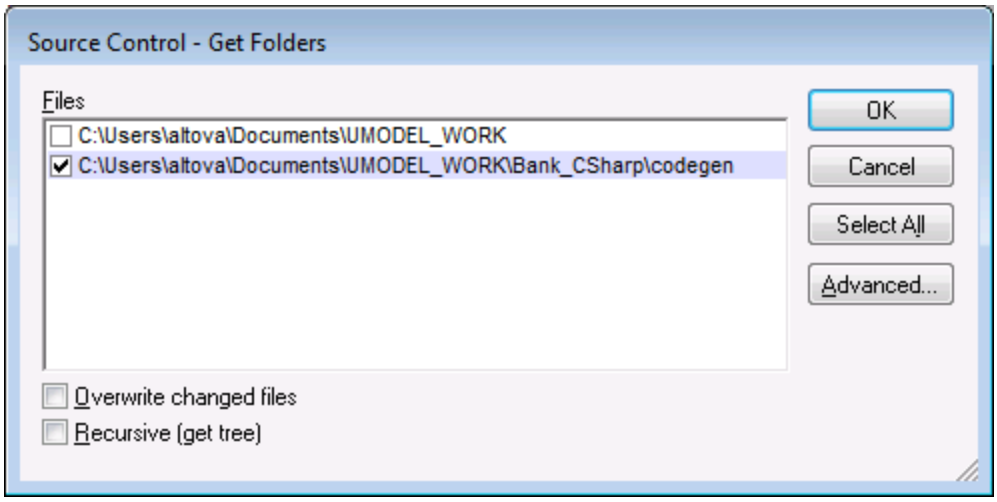
The "Make writable" check box removes the read-only attribute of the retrieved files.

13.3.5 Get Folder(s)

Retrieves read-only copies of files in the selected folders and places them in the working folder. The files are not checked-out for editing per default.

Using Get Folders:

- Select the folder you want to get in the Model Tree.
- Select **Project** | **Source Control** | **Get Folders**.

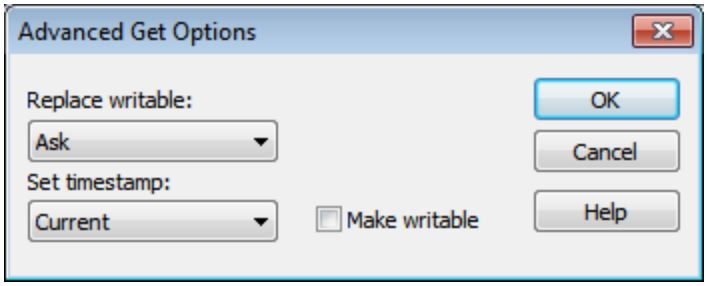


Overwrite changed files
Overwrites those files that have been changed locally with those from the source control database.

Recursive (get tree)
Retrieves all files of the folder tree below the selected folder.

Advanced

Allows you to define the **Replace writable** and **Set timestamp** options in the respective combo boxes.



The "Make writable" check box removes the read-only attribute of the retrieved files.

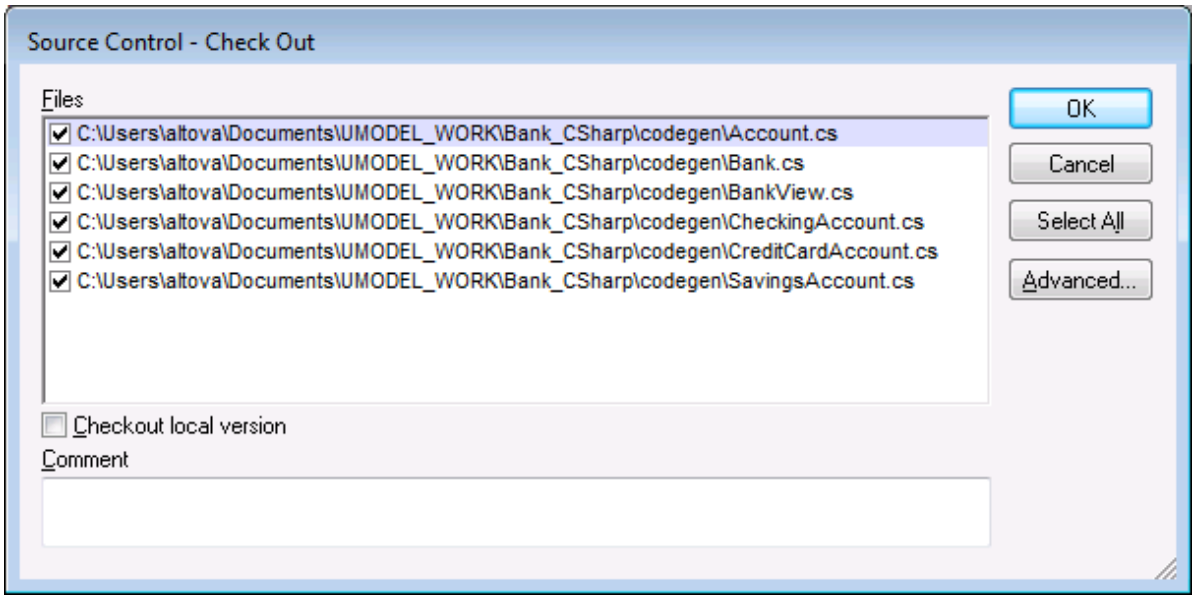
13.3.6 Check Out

This command **checks out** the latest version of the selected files and places writable copies in the working directory. The files are flagged as "checked out" for all other users.

To Check Out files:

- Select the file or folder you want to check out in the Model Tree.
- Select **Project | Source Control | Check Out**.

© 2019-2025 Altova GmbH

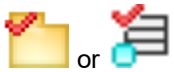


Note: You can change the number of files to check out, by activating the individual check boxes in the Files list box.

Select the option **Checkout local version** to check out only the local versions of the files, not those from the source control database.

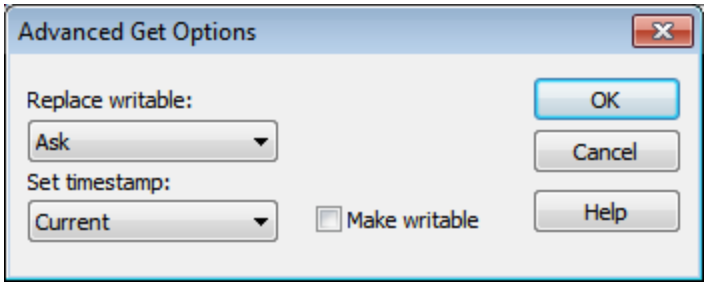
The following items can be checked out:

- Single files, click on the respective files (CTRL + click, in the Model Tree)
- Folders, click on the folders (CTRL + click, in the Model Tree)

 or
The red check mark denotes that the file/folder has been checked out.

Advanced

Allows you to define the **Replace writable** and **Set timestamp** options in the respective combo boxes.



The "Make writable" check box removes the read-only attribute of the retrieved files.

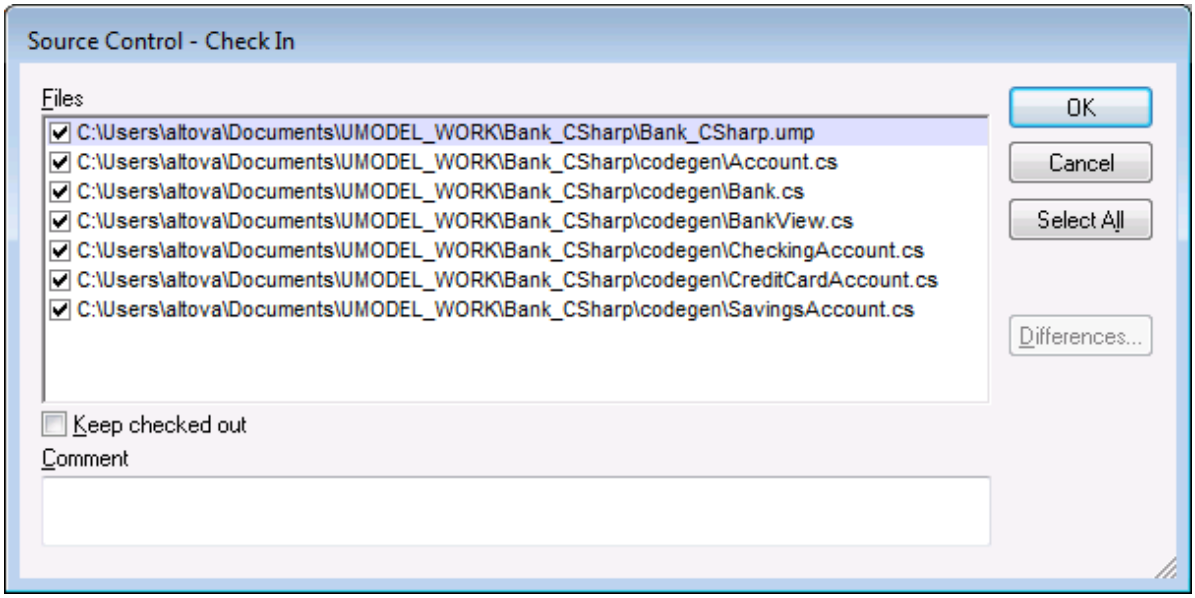
13.3.7 Check In

This command **checks in** the previously checked out files, i.e. your locally updated files, and places them in the source control database.

To Check In files:

- Select the files in the Model Tree
- Select **Project** | **Source Control** | **Check In**.

Shortcut: Right-click a checked out item in the project window, and select "Check in" from the Context menu.



Note:
You can change the number of files to check in, by activating the individual check boxes in the Files list box.

The following items can be checked in:

- Single files, click on the respective files (CTRL + click, in Model Tree)
- Folders, click on the folders (CTRL + click, in Model Tree)

, or
The lock symbol denotes that the file/folder is **under source control**, but is currently not checked out.

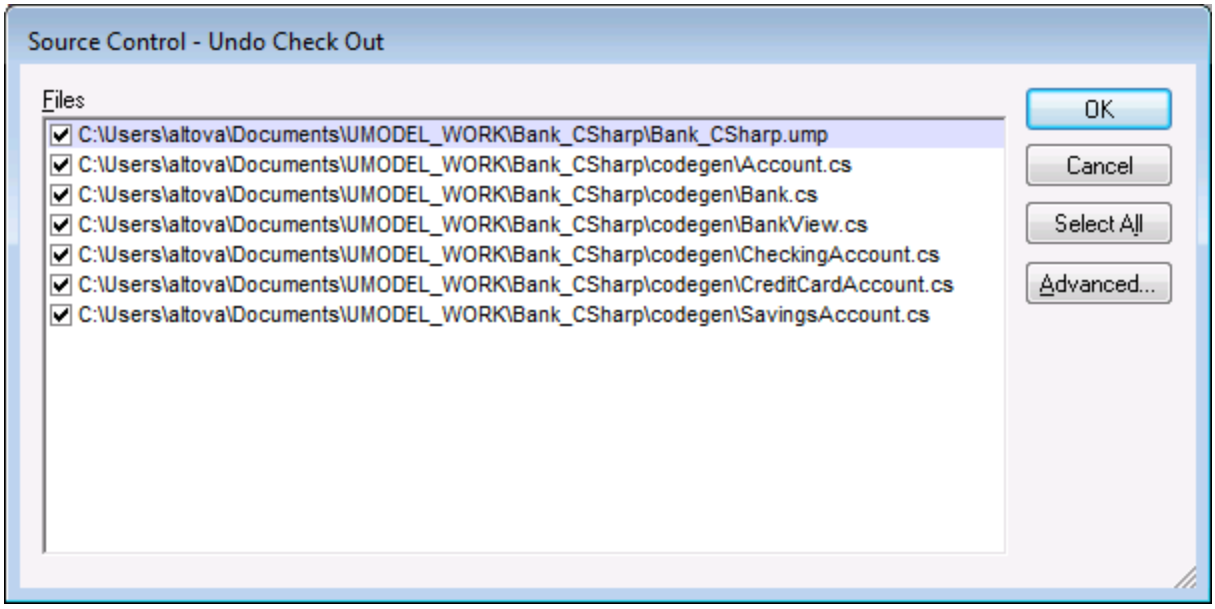
13.3.8 Undo Check Out...

This command **discards changes** made to previously checked out files, i.e. your locally updated files, and retains the old files from the source control database.

© 2019-2025 Altova GmbH

To Undo Check Out..

- Select the files in the Model Tree
- Select **Project** | **Source Control** | **Undo Check Out**.

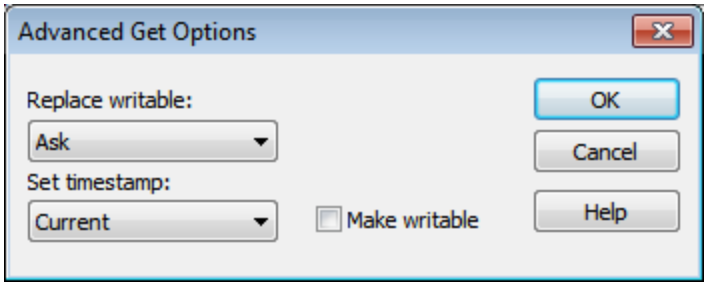


Note:
You can change the number of files by activating the individual check boxes in the Files list box.

The Undo check out option can apply to the following items:
- Single files, click on the respective files (CTRL + click, in Model Tree)
- Folders, click on the folders (CTRL + click, in Model Tree)

Advanced

Allows you to define the **Replace writable** and **Set timestamp** options in the respective combo boxes.



The "Make writable" check box removes the read-only attribute of the retrieved files.

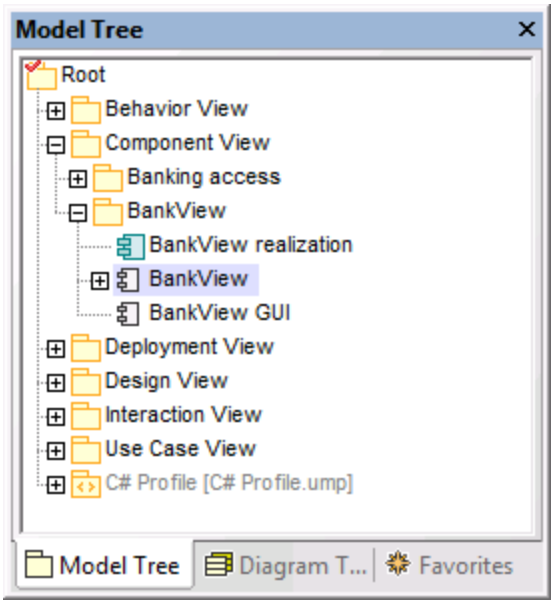
13.3.9 Add to Source Control

Adds the selected files or folders to the source control database and places them under source control. If you are adding a new UModel project you will be prompted for the workspace folder and the location at which your project should be stored.

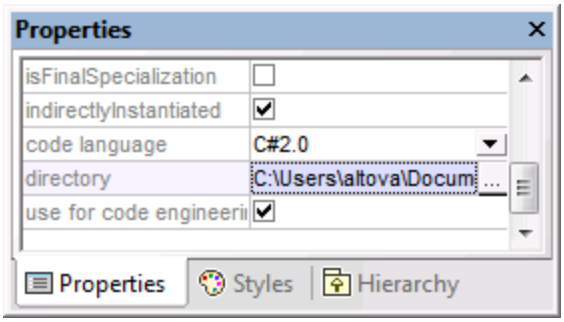
Having placed the UModel project file (*.ump) under source control, you can then add the code files produced by the code-engineering process, to source control as well. For this to work, the generated code files and the UModel project have to be placed **in**, or **under**, the same SourceSafe **working** directory. The working directory used in this section is **C:\Users\Altova\Documents\UMODEL_WORK**.

To add UModel generated code files to source control:

1. Expand the Component View folder in the Model Tree and Navigate to the BankView component.

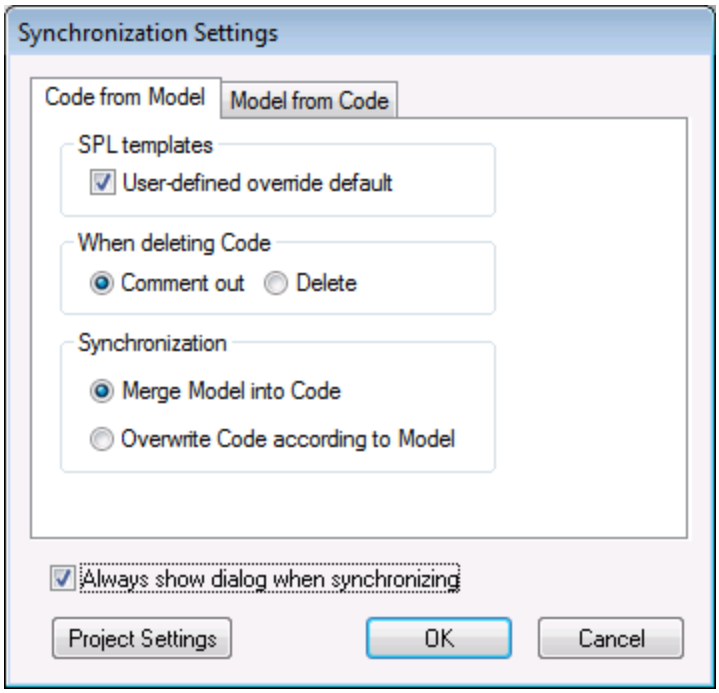


2. Click the BankView component and click the Browse icon next to the "**directory**" field in the Properties window.

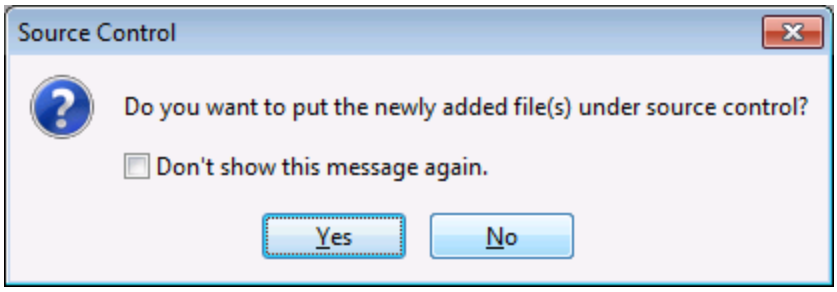


3. Change the code engineering directory to **C:\Users\Altova\Documents\UMODEL_WORK\codegen**.
4. Select the menu item **Project | Merge Program Code from UModel project**.
5. Change the Synchronization settings if necessary, and click OK to confirm.

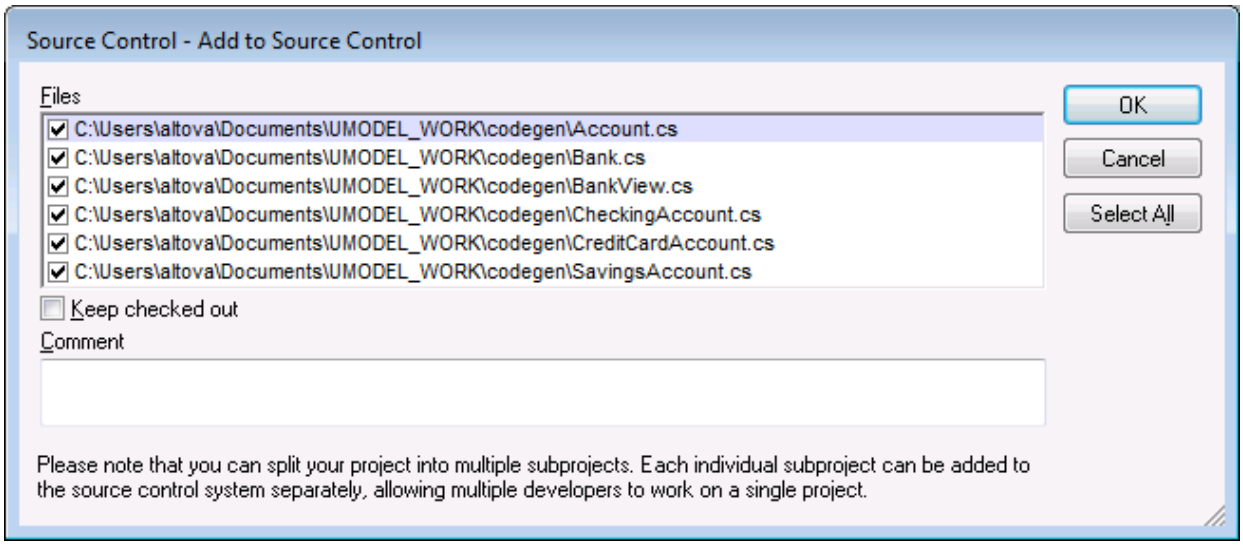
© 2019-2025 Altova GmbH



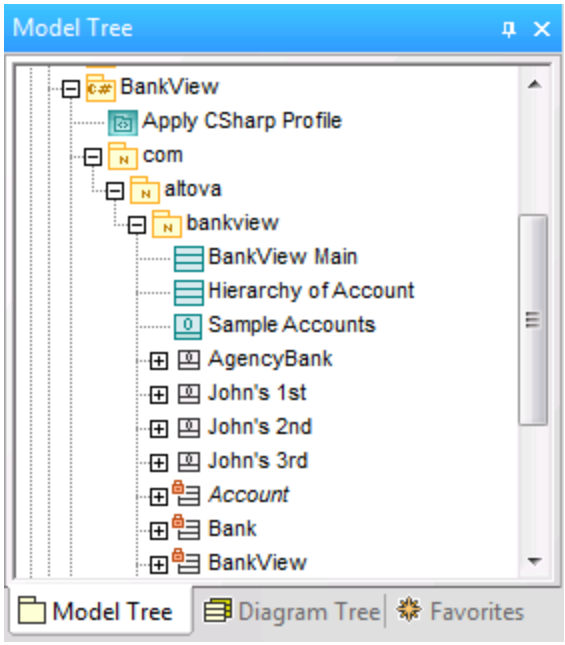
The Messages window displays the code from project process.
A message box opens asking if you want to place the newly created files under source control.



6. Click Yes to do so.
7. The "Add to Source Control" dialog box is opened, allowing you to select the files you want to place under source control.



8. Click OK once you have selected the files you want to place under source control.
The lock symbol now appears next to each of the classes/file sources placed under source control.



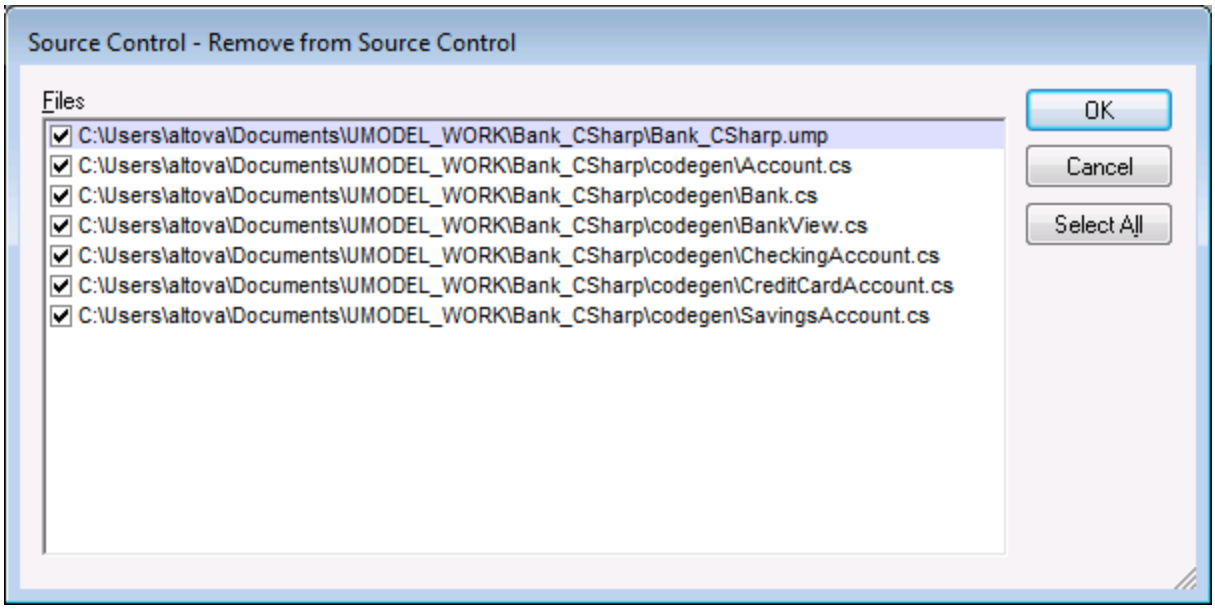
13.3.10 Remove from Source Control

This command **removes** previously added files, from the source control database. These type of files remain visible in the Model Tree but cannot be checked in or out. Use the "Add to Source Control" command to place them back under source control.

© 2019-2025 Altova GmbH

To remove files from the source control provider:

- Select the files you want to remove in the Model Tree.
- Select **Project** | **Source Control** | **Remove from Source Control**.



Note:
You can change the number of files to remove, by activating the individual check boxes in the Files list box.

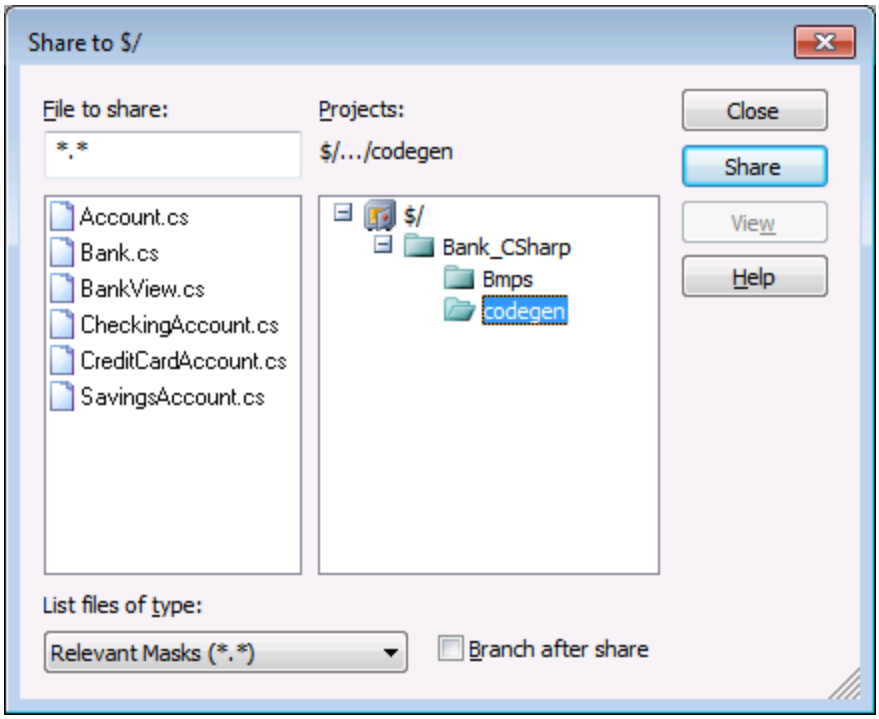
The following items can be removed from source control:
- Single files, click on the respective files (CTRL + click, for several)
- Folders, click on the folder icon.

13.3.11 Share from Source Control

This command shares/branches files from other projects/folders within the source control repository, into the selected folder. To use the Share command you must have the Check in/out rights to the project you are sharing from.

To share a file from source control:

1. Select the folder you want to share files to, in the Model Tree window, and select **Project | Source Control | Share from Source Control**. e.g. BankView Component in the Component View folder.
2. Select the project folder that contains the file you want to share in the "Projects" list box.



3. Select the file you want to share in the "Files to share" list box and click the Share button. The file is now removed from the "File to share" list.
4. Click the Close button to continue.

Branch after share
Shares the file and creates a new branch to create a separate version.

13.3.12 Show History

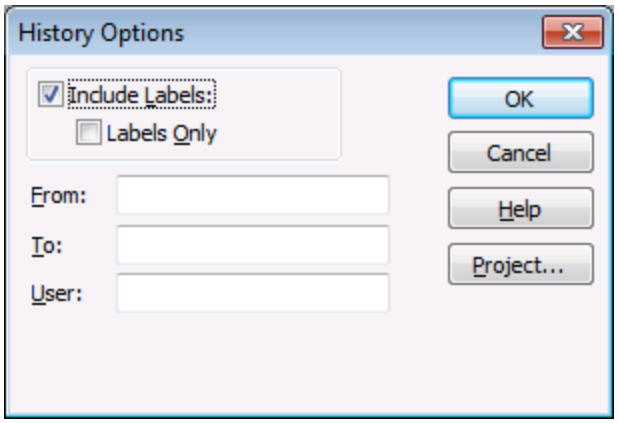
This command **displays** the history of a file under source control, and allows you to view, see detailed history info, difference, or retrieve previous versions of a file.

To show the history of a file:

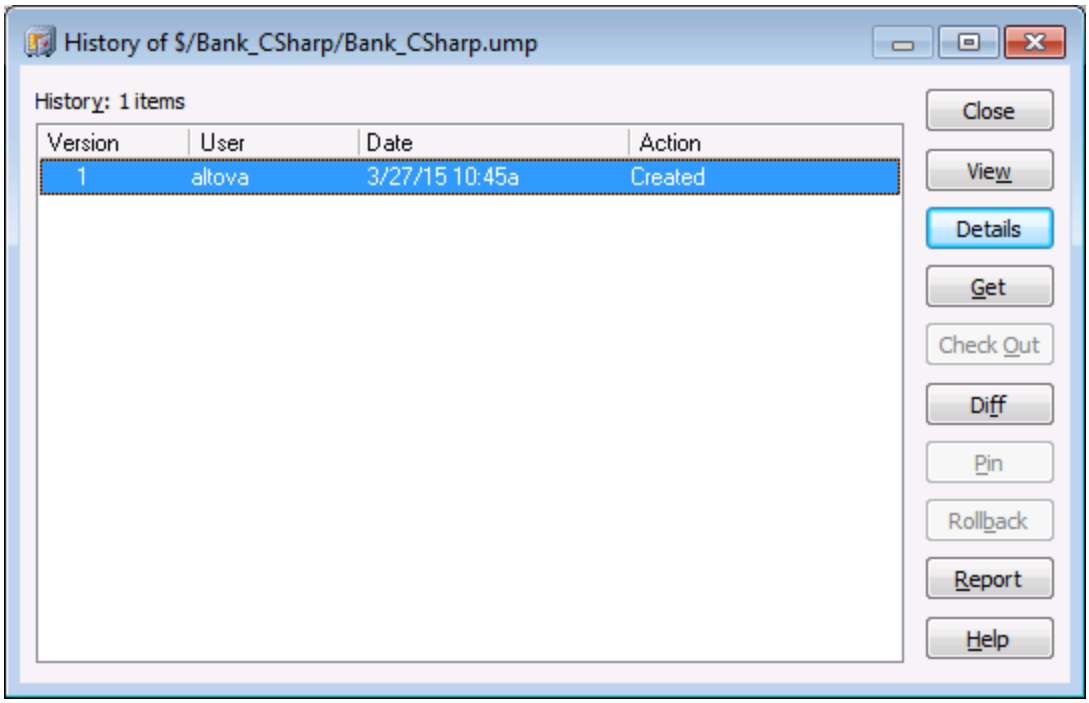
1. Click on the file in the Model Tree window.
2. Select the menu options **Project | Source control | Show history**.

   A dialog box prompting for more information opens.

© 2019-2025 Altova GmbH



3. Select the appropriate entries and confirm with **OK**.



This dialog box is provides various way of comparing and getting specific versions of the file in question. Double clicking an entry in the list opens the History Details dialog box for that file.

Close
Closes this dialog box.

View
Opens a further dialog box in which you can select the type of viewer you want to see the file with.

Details
Opens a dialog box in which you can see the properties 629 of the currently active file.

Get
Allows you to retrieve one of the previous versions of the file in the version list, and place it into the working directory.

Check Out
Allows you to check out the **latest** version of the file.

Diff
Opens the Difference options [628] dialog box, which allows you to define the difference options when viewing the differences between two file versions.

Use CTRL+Click to mark two file versions in this window, then click Diff to view the differences between them.

Pin
Pins or unpins a version of the file, allowing you to define the specific file version to use when differencing two files.

Rollback
Rolls back to the selected version of the file.

Report
Generates a history report which you can send to the printer, file, or clipboard.

Help
Opens the online help of the source control provider plugin.

13.3.13 Show Differences

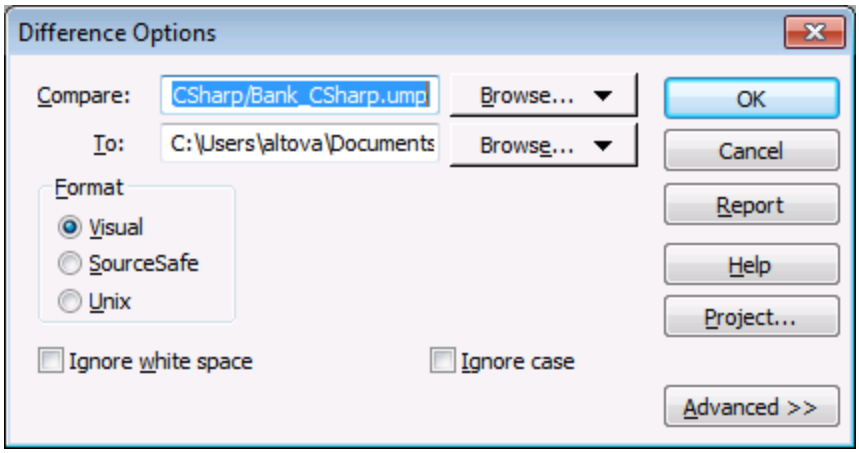
This command **displays** the differences between the file currently in the source control repository, and the **checked in/out** file of the same name in the working directory.

If you have "pinned" one of the files in the **history** dialog box, then the pinned file will be used in the "Compare" text box. Any two files can be selected using the Browse buttons.

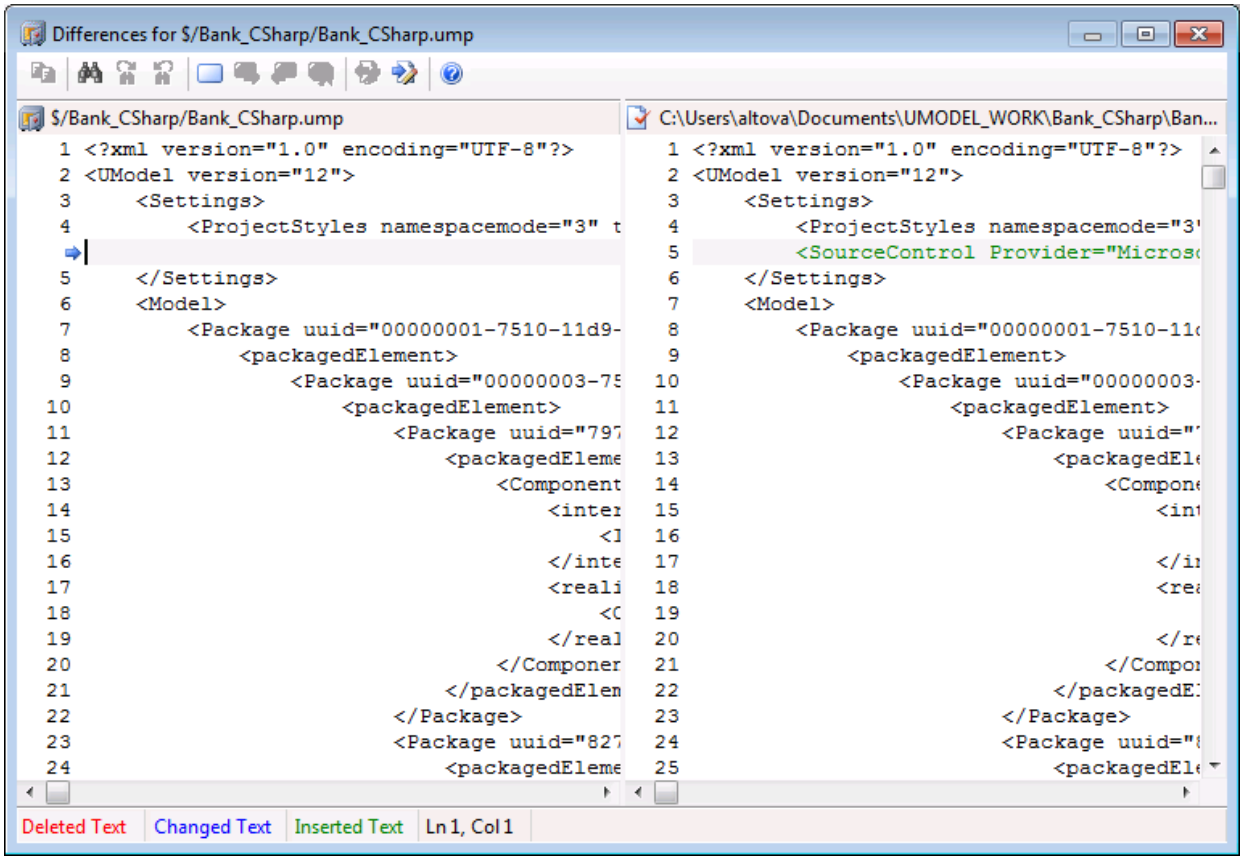
To show the differences between two files:

1. Click on a file in the Model Tree window.
2. Select the menu option **Project | Source control | Show Differences**.
   A dialog box prompting for more information appears.

© 2019-2025 Altova GmbH



3. Select the appropriate entries and confirm with **OK**.



The differences between the two files are highlighted in both windows (this example uses MS SourceSafe).

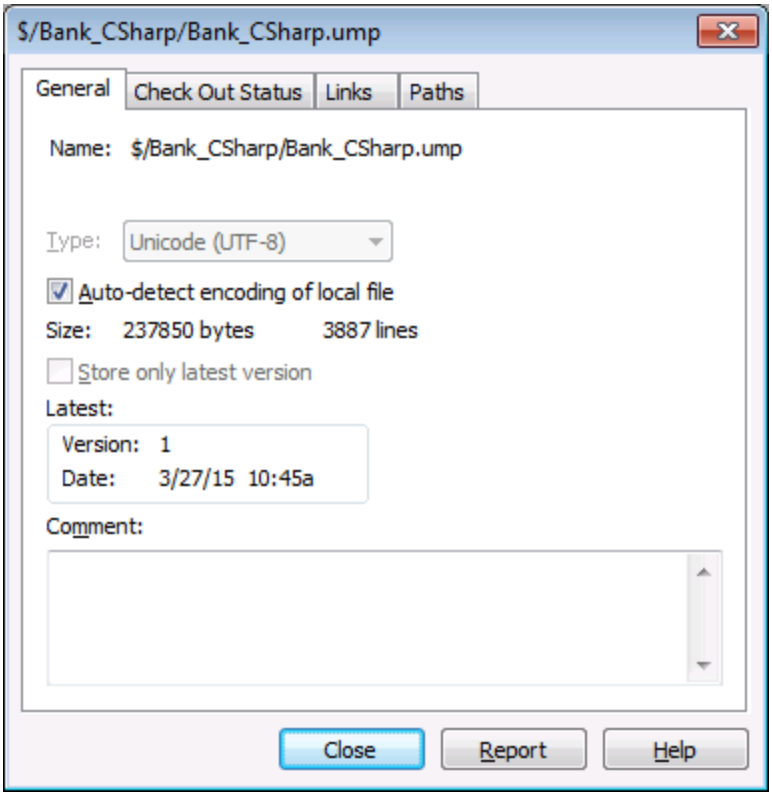
13.3.14 Show Properties

This command **displays** the properties of the currently selected file, and is dependent on the source control provider you use.

To display the properties of the currently selected file:

- Select **Project** | **Source Control** | **Properties**.

This command can only be used on single files.



13.3.15 Refresh Status

This command **refreshes** the status of all project files, independent of their current status.

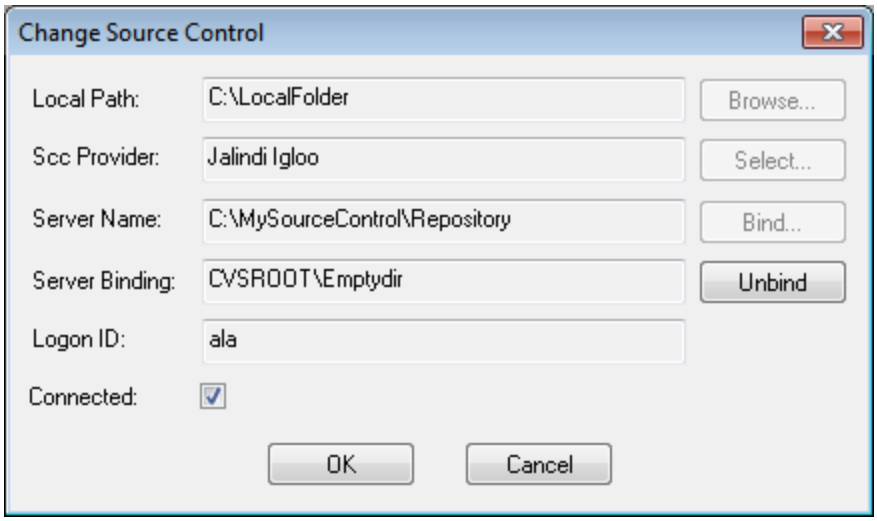
13.3.16 Source Control Manager

This command **starts** your source control software with its native user interface.

13.3.17 Change Source Control

This dialog box allows you to change the source control binding that you are using. Click the Unbind button first, then (optionally) click the Select button to select a new source control provider, and finally click the Bind button to bind to a new location in the repository.

© 2019-2025 Altova GmbH

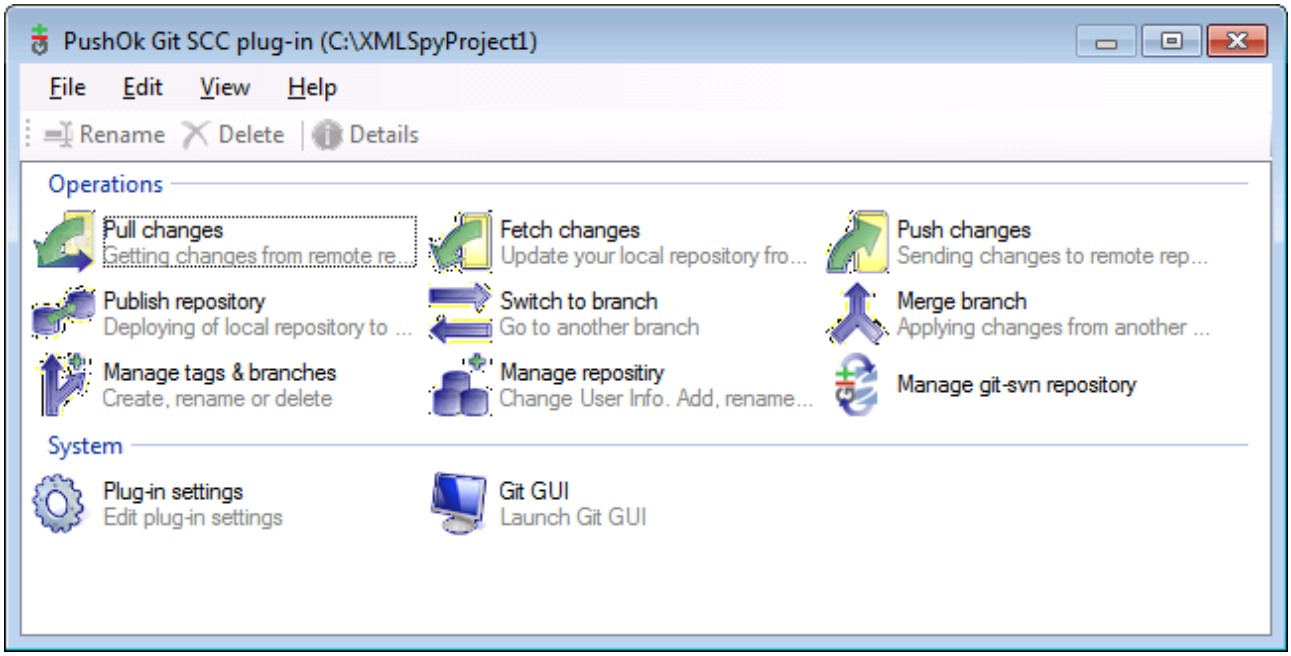

Change Source Control
Local Path:
C:\LocalFolder
Browse...
Scc Provider:
Jalindi Igloo
Select...
Server Name:
C:\MySourceControl\Repository
Bind...
Server Binding:
CVSROOT\Emptydir
Unbind
Logon ID:
ala
Connected:
OK
Cancel

13.4 Source Control with Git

Support for Git as a source control system in UModel is available through a third-party plug-in called **GIT SCC plug-in** (http://www.pushok.com/software/git.html).

> At the time when this documentation is written, the **GIT SCC plug-in** is available for experimental use. Registration with the plug-in publisher is required in order to use the plug-in.

The GIT SCC plug-in enables you to work with a Git repository using the commands available in the **Project | Source Control** menu of UModel. Note that the commands in the **Project | Source Control** menu of UModel are provided by the Microsoft Source Control Plug-in API (MSSCCI API), which uses a design philosophy different from Git. As a result, the plug-in essentially mediates between "Visual Source Safe"-like functionality and Git functionality. On one hand, this means that a command such as **Get latest version** may not be applicable with Git. On the other hand, there are new Git-specific actions, which are available in the "Source Control Manager" dialog box provided by the plug-in (under the **Project | Source Control | Source Control Manager** menu of UModel).



The Source Control Manager dialog box

Other commands that you will likely need to use frequently are available directly under the **Project | Source Control** menu.

The following sections describe the initial configuration of the plug-in, as well as the basic workflow:

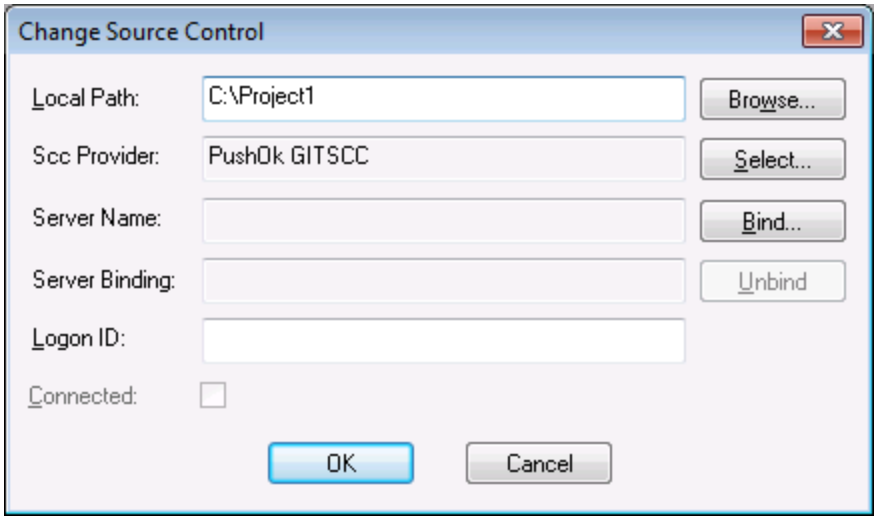
- Enabling Git Source Control with GIT SCC Plug-in 633
- Adding a Project to Git Source Control 633
- Cloning a Project from Git Source Control 635

© 2019-2025 Altova GmbH

13.4.1 Enabling Git Source Control with GIT SCC Plug-in

To enable Git source control with UModel, the third-party **PushOK GIT SCC plug-in** must be installed, registered, and selected as source control provider, as follows:

1. Download the plug-in installation file from the publisher's website (http://www.pushok.com), run it, and follow the installation steps.
2. On the **Project** menu of UModel, click **Change Source Control**, and make sure **PushOk GITSCC** is selected as source control provider. If you do not see **Push Ok GITSCC** in the list of providers, it is likely that the installation of the plug-in was not successful. In this case, check the publisher's documentation for a solution.



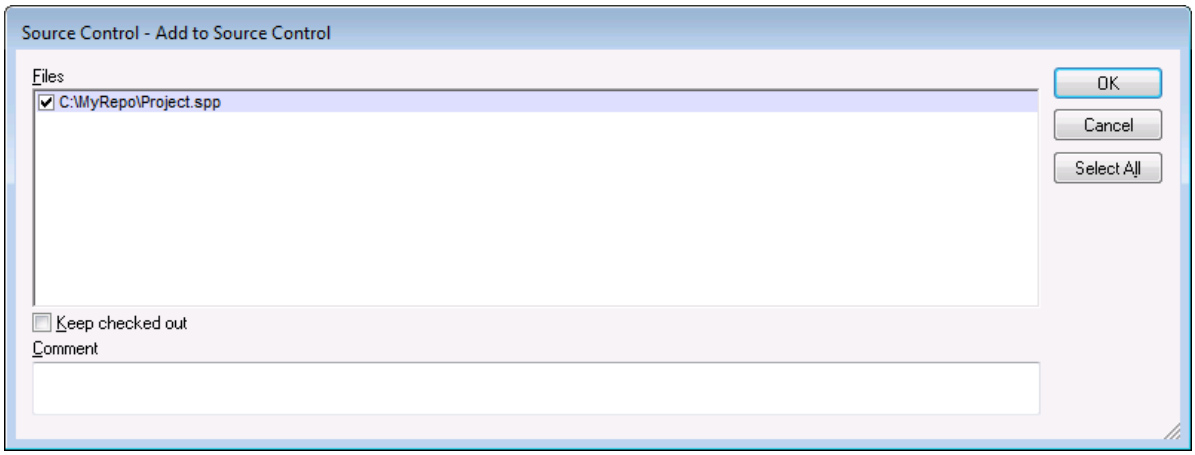
3. When a dialog box prompts you to register the plug-in, click **Registration** and follow the wizard steps to complete the registration process.

13.4.2 Adding a Project to Git Source Control

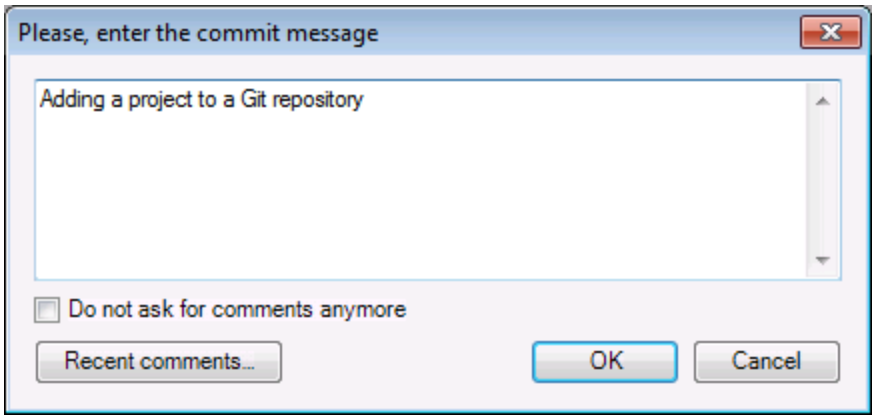
You can save UModel projects as Git repositories. The structure of files or folders that you add to the project would then correspond to the structure of the Git repository.

To add a project to Git source control:

1. Make sure that **PushOK GIT SCC Plug-in** is set as source control provider (see Enabling Git Source Control with GIT SCC Plug-in 633).
2. Create a new empty project and make sure that it has no validation errors (that is, the command **Project | Check Project Syntax** does not show any errors or warnings).
3. Save the project to a local folder, for example `C:\MyRepo\Project.ump`.
4. In the **Model Tree** pane, click the **Root** node.
5. On the **Project** menu, under **Source Control**, click **Add to Source Control**.

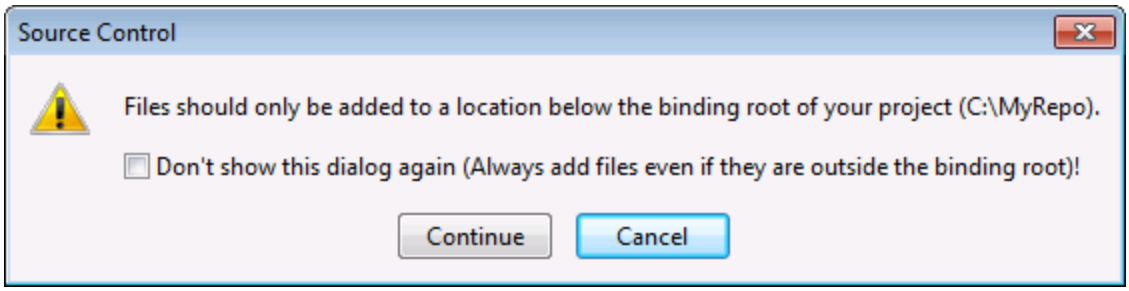


6. Click **OK**.



7. Enter the text of your commit message, and click **OK**.

You can now start adding modeling elements (diagrams, classes, packages, and so on) to your project. Note that all project files and folders must be under the root folder of the project. For example, if the project was created in the `C:\MyRepo` folder , then only files under `C:\MyRepo` should be added to the project. Otherwise, if you attempt to add to your project files that are outside the project root folder, a warning message is displayed:

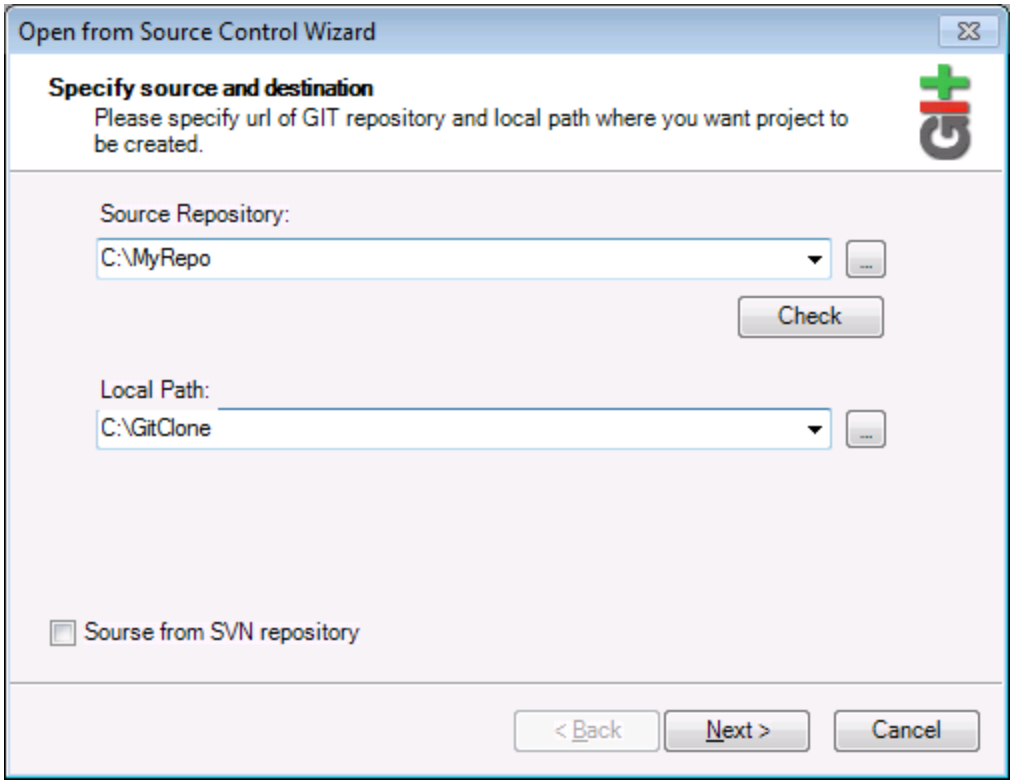


© 2019-2025 Altova GmbH

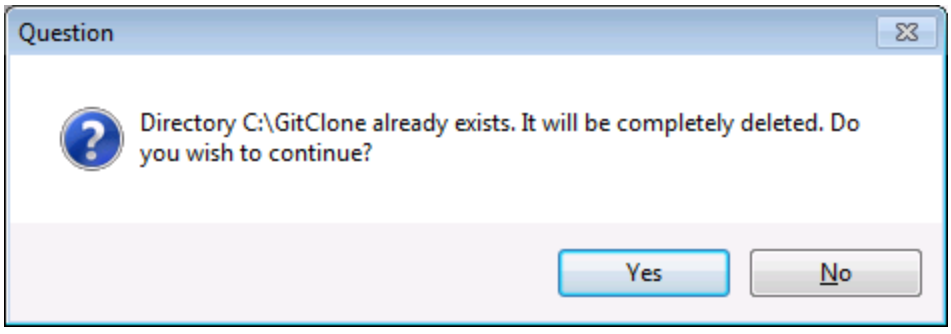
13.4.3 Cloning a Project from Git Source Control

Projects that have been previously added to Git source control (see Adding a Project to Git Source Control 633) can be opened from the Git repository as follows:

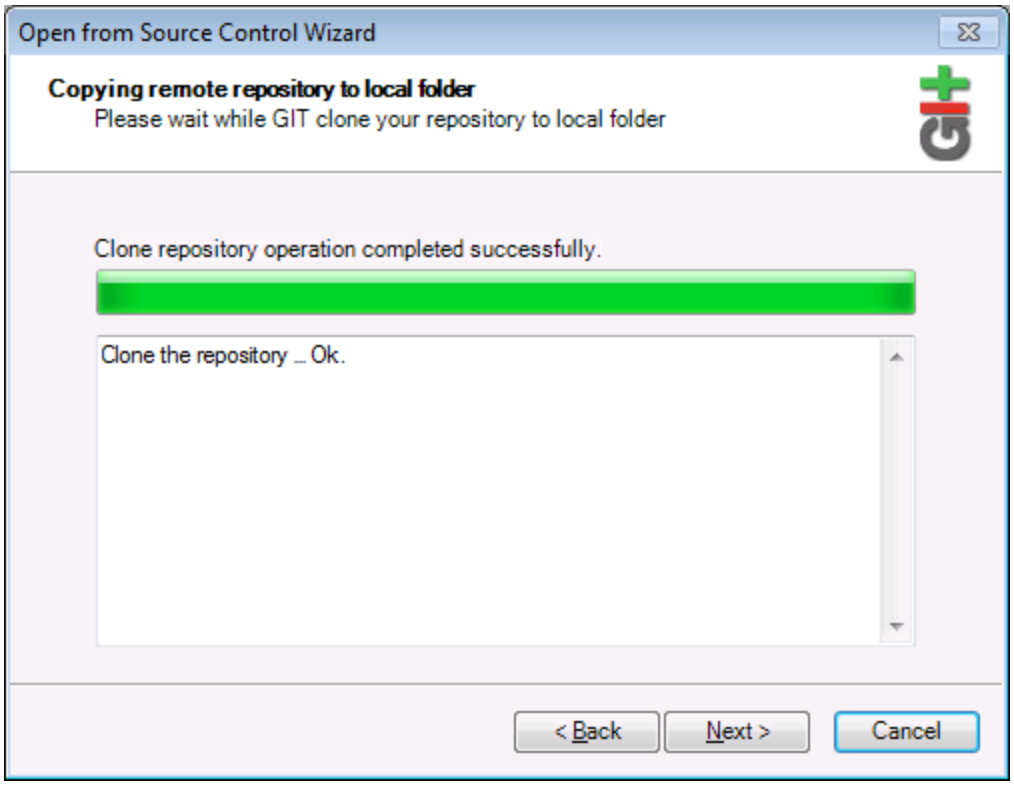
1. Make sure that **PushOK GIT SCC Plug-in** is set as source control provider (see Enabling Git Source Control with GIT SCC Plug-in 633).
2. On the **Project** menu, click **Source Control | Open from Source Control**.
3. Enter the path or the URL of the source repository. Click **Check** to verify the validity of the path or URL.



4. Under **Local Path**, enter the path to local folder where you want the project to be created, and click **Next**. If the local folder exists (even if it is empty), the following dialog box opens:



5. Click **Yes** to confirm, and then click **Next**.



6. Follow the remaining wizard steps, as required by your specific case.
7. When the wizard completes, a Browse dialog box appears, asking you to open the UModel Project (*.ump) file. Select the project file to load the project contents into UModel.

© 2019-2025 Altova GmbH

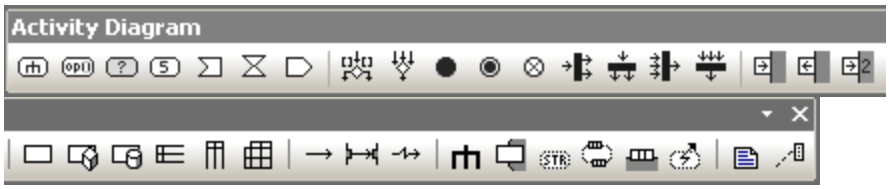
14 UModel Diagram icons

The following section is a quick guide to the icons that are made available in each of the modeling diagrams.

The icons are split up into two sections:

- **Add** - displays a list of elements that can be added to the diagram.
- **Relationship** - displays a list of relationship types that can be created between elements in the diagram.

14.1 Activity Diagram



Add

Action (CallBehaviorAction)
Action (CallOperationAction)
AcceptEventAction
AcceptEventAction (TimeEvent)
SendSignalAction

DecisionNode (Branch)
MergeNode
InitialNode
ActivityFinalNode
FlowFinalNode
ForkNode (vertical)
ForkNode (horizontal)
JoinNode
JoinNode (horizontal)

InputPin
OutputPin
ValuePin

ObjectNode
CentralBufferNode
DataStoreNode
ActivityPartition (horizontal)
ActivityPartition (vertical)
ActivityPartition 2-Dimensional

ControlFlow
ObjectFlow
ExceptionHandler

Activity
ActivityParameterNode
StructuredActivityNode
ExpansionRegion
ExpansionNode
InterruptibleActivityRegion

© 2019-2025 Altova GmbH

Note
Note Link

14.2 Class Diagram

Relationship

Association
Aggregation
Composition
AssociationClass
Dependency
Usage
InterfaceRealization
Generalization

Add

Package
Class
Interface
Enumeration
Datatype
PrimitiveType
Profile
Stereotype
ProfileApplication
InstanceSpecification

Note
Note Link

© 2019-2025 Altova GmbH

14.3 Communication diagram

Add

Lifeline
Message (Call)
Message (Reply)
Message (Creation)
Message (Destruction)

Note
Note Link

14.4 Composite Structure Diagram

Add

Collaboration
CollaborationUse
Part (Property)
Class
Interface
Port

Relationship

Connector
Dependency (Role Binding)
InterfaceRealization
Usage

Note
Note Link

© 2019-2025 Altova GmbH

14.5 Component Diagram

Add

Package
Interface
Class
Component
Artifact

Relationship

Realization
InterfaceRealization
Usage
Dependency

Note
Note Link

14.6 Deployment Diagram

Add

Package
Component
Artifact
Node
Device
ExecutionEnvironment

Relationship

Manifestation
Deployment
Association
Generalization
Dependency

Note
Note Link

© 2019-2025 Altova GmbH

14.7 Interaction Overview diagram

Add

CallBehaviorAction (Interaction)
CallBehaviorAction (InteractionUse)
DecisionNode
MergeNode
InitialNode
ActivityFinalNode
ForkNode
ForkNode (Horizontal)
JoinNode
JoinNode (Horizontal)
DurationConstraint

Relationship

ControlFlow

Note
Note Link

14.8 Object Diagram

Relationship

Association
AssociationClass
Dependency
Usage
InterfaceRealization
Generalization

Add

Package
Class
Interface
Enumeration
Datatype
PrimitiveType
InstanceSpecification

Note
Note Link

© 2019-2025 Altova GmbH

14.9 Package diagram

Add

Package
Profile

Relationship

Dependency
PackageImport
PackageMerge
ProfileApplication

Note
Note Link

14.10 Profile Diagram

Add

Profile
Stereotype

Relationship

Generalization
ProfileApplication
PackageImport
ElementImport

Note
NoteLink

© 2019-2025 Altova GmbH

14.11 Protocol State Machine

Add

Simple state
Composite state
Orthogonal state
Submachine state

FinalState
InitialState

EntryPoint
ExitPoint
Choice
Junction
Terminate
Fork
Fork (horizontal)
Join
Join (horizontal)
ConnectionPointReference

Relationship

Protocol Transition

Note
Note link

14.12 Sequence Diagram

Add

Lifeline
CombinedFragment
CombinedFragment (Alternatives)
CombinedFragment (Loop)
InteractionUse
Gate
StateInvariant
DurationConstraint
TimeConstraint

Message (Call)
Message (Reply)
Message (Creation)
Message (Destruction)

Asynchronous Message (Call)
Asynchronous Message (Reply)
Asynchronous Message (Destruction)

Note
Note Link

No message numbering
Simple message numbering
Nested message numbering

Toggle dependent message movement
Toggle automatic creation of replies for messages
Toggle automatic creation of operations in target by typing operation names

© 2019-2025 Altova GmbH

14.13 State Machine Diagram

Add

Simple state
Composite state
Orthogonal state
Submachine state

FinalState
InitialState

EntryPoint
ExitPoint
Choice
Junction
Terminate
Fork
Fork (horizontal)
Join
Join (horizontal)
DeepHistory
ShallowHistory
ConnectionPointReference

Relationship

Transition

Note
Note link

Toggle automatic creation of operations in target by typing operation names

14.14 Timing Diagram

Add

Lifeline (State/Condition)
Lifeline (General value)
TickMark
Event/Stimulus
DurationConstraint
TimeConstraint

Message (Call)
Message (Reply)
Asynchronous Message (Call)

Note
Note Link

© 2019-2025 Altova GmbH

14.15 Use Case diagram

Add

Package
Actor
UseCase

Relationship

Association
Generalization
Include
Extend

Note
Note Link

14.16 XML Schema diagram

Add

XSD TargetNamespace
XSD Schema
XSD Element (global)
XSD Group
XSD ComplexType
XSD ComplexType (simpleContent)
XSD SimpleType
XSD List
XSD Union
XSD Enumeration
XSD Attribute
XSD AttributeGroup
XSD Notation
XSD Import

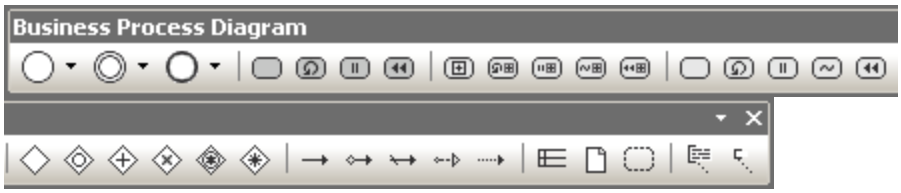
Relationship

XSD Include
XSD Redefine
XSD Restriction
XSD Extension
XSD Substitution

Note
Note link

© 2019-2025 Altova GmbH

14.17 Business Process Modeling Notation



Add

Start Event
Intermediate Event
Stop Event

Task
Loop Task
Multi Instance Task
Compensation Task

Collapsed Sub Process
Collapsed Loop Sub Process
Collapsed Multi Instance Sub Process
Collapsed Ad Hoc Process
Collapsed Compensation Sub Process

Expanded Sub Process
Expanded Loop Sub Process
Expanded Multi Instance Sub Process
Expanded Ad Hoc Process
Expanded Compensation Sub Process

Gateway
Inclusive Gateway (OR)
Parallel Gateway (AND)
Data Based Exclusive Gateway (XOR)
Event Based Exclusive Gateway (XOR)
Complex Gateway (Decision/Merge)

Relationship

Sequence Flow
Conditional Flow
Default Flow
Message Flow
Association

Pool
Data Object

Group

Text Annotation
Annotation Association

© 2019-2025 Altova GmbH

14.18 Business Process Modeling Notation 2.0

Add

Start Event
Catch Event
Throw Event
End Event

Task
Expanded Sub Process
Collapsed Sub Process
Call Activity
Gateway

Relationship

Sequence Flow
Default Sequence Flow
Conditional Sequence Flow
Message Flow
Association

Pool
Group
Data Object
Data Output
Data Input
Collection Data Object
Data Store
Message

Text Annotation
Annotation Association

14.19 Database Modeling

Add

Table
CheckConstraint
PrimaryKey
ForeignKey
UniqueKey
Index

Relationship

Database Relationship Association
Database Relationship with Attributes

© 2019-2025 Altova GmbH

15 Menu Reference

The following section lists all the menus and menu options in UModel, and supplies a short description of each.

15.1 File

New

Clears the diagram tab, if a previous project exists, and creates a new UModel project.

Open

Opens previously defined modeling project. Select a previously saved project file ***.ump** from the Open dialog box. See Creating, Opening, and Saving Projects 154 and Opening Projects from a URL 155.

Reload

Reloads the current project and saves or discards the changes made since you opened the project file.

Save

Saves the currently active modeling project using the currently active file name.

Save as

Saves the currently active modeling project with a different name, or allows you to give the project a new name if this is the first time you save it.

Save Copy As

Saves a copy of the currently active UModel project with a different file name.

Save Diagram as Image

Opens the "Save as..." dialog box and allows you to save the currently active diagram as a .png file. Very large .png files, in the gigabyte range, can also be saved.

Save all Diagrams as Images

Save all diagrams of the currently active project as .png files.

Import from XMI File

Imports a previously exported XMI file. If the file was produced with UModel, then all extensions etc. will be retained.

Export to XMI File

Exports the model as an XMI file. You can select the UML version, as well as the specific IDs that you want to export, see XMI - XML Metadata Interchange 568.

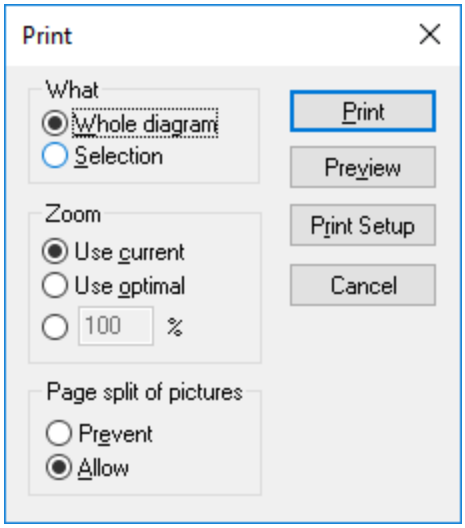
Send by Mail

Opens your default mail application and inserts the current UModel project as an attachment.

© 2019-2025 Altova GmbH

Print

Opens the Print dialog box, from where you can print out the current diagram (or a selection on the diagram) as hard copy.



Use current retains the currently defined zoom factor of the modeling project. Selecting this option enables the "Page split of pictures" group. **Use optimal** scales the modeling project to fit the page size. You can also specify the zoom factor numerically. The **Prevent** option prevents modeling elements from being split over a page, and keeps them as one unit.

Print all Diagrams

Opens the Print dialog box and prints out all UML diagrams contained in the current project file.

Print Preview

Opens the same Print dialog box with the same settings as described above.

Print Setup

Opens the Print Setup dialog box in which you can define the printer you want to use and the paper settings.

Recent files

This section of the **File** menu lists up to four most recent files you have been working with.

Exit

The **Exit** command exist UModel. If any of your current files have unsaved changes, UModel will prompt you to save the changes.

15.2 Edit

Undo

UModel has an unlimited number of "Undo" steps that you can use to retrace your modeling steps.

Redo

The redo command allows you to redo previously undone commands. You can step backward and forward through the undo history using both these commands.

Cut/Copy/Paste/Delete

These are the standard Windows text editing commands. You can use them not only for text but also for modeling elements, see Renaming, Moving, and Copying Elements 112.

Paste in Diagram only

Adds a "link" (or "view") of the copied element to the current diagram but not to the Model Tree, see Renaming, Moving, and Copying Elements 112.

Delete from Diagram only

Deletes the selected modeling elements from the currently active diagram. The deleted elements are not deleted from the modeling project and are available in the Model Tree tab. Note that this option is not available to delete properties or operations from a class, they can be selected and deleted there directly.

Select all

Select all modeling elements of the currently active diagram. Equivalent to the **Ctrl+A** shortcut.

Find

Allows you to search for specific text in the current window, see Finding and Replacing Text 114.

Find Next F3

Searches for the next occurrence of the same search string in the currently active window.

Find Previous (Shift+F3)

Searches for the previous occurrence of the same search string in the currently active tab or diagram.

Replace

Allows you to search and replace any modelling elements in the project, see Finding and Replacing Text 114.

© 2019-2025 Altova GmbH

Copy as Bitmap

Copies the currently active diagram to clipboard, from where you can paste it into the application of your choice.

Copy Selection as Bitmap

Copies the currently selected diagram elements to the clipboard from where you can paste them into the application of your choice.

15.3 Project

Check Project Syntax

Checks the UModel project syntax, see Checking Project Syntax 173.

Source Control

See Source control systems 607 for detailed information on source control servers and clients and how to use them.

Import Source Directory

Opens the Import Source Directory wizard. For a specific example, see Reverse Engineering (from Code to Model) 73.

Import Source Project

Opens the Import Source Project wizard, see Importing Source Code 192.

Import Binary Types

Opens the Import Binary Types dialog box allowing you to import Java, C#, and VB binary files, see Importing Java, C#, and VB.NET Binaries 204.

Import XML Schema Directory

Opens the Import XML Schema Directory allowing you to import all XML Schemas in that directory and optionally all XML Schemas in any of the subfolders.

Import XML Schema File

Opens the Import XML Schema File dialog box allowing you to import schema files, see XML Schema Diagrams 424.

Import SQL Database

Opens the Import Database dialog box from where you can import database structure into the model, see Importing SQL Databases into UModel 488.

Generate Sequence Diagrams from Code...

See Generate Multiple Sequence Diagrams 371.

Generate Code from Sequence Diagrams

UModel can create code from a sequence diagram which is linked to at least one operation. For more information, see this section 372.

© 2019-2025 Altova GmbH

Generate State Machine Code

UModel enables you to select one or more state machines in which code should be generated. For details, see this topic 326.

Merge Program Code from UModel Project / Overwrite Program Code from UModel Project

Updates program code from the model (assuming that your project is set up for code engineering, see Generating Program Code 170). The name of this command can be either **Merge Program Code from UModel Project** or **Overwrite Program Code from UModel Project**, depending on the settings in the Synchronization Settings dialog box. By default, the Synchronization Settings dialog box opens every time when you run this command. For more information, see Code Synchronization Settings 221.

Merge UModel Project from Program Code / Overwrite UModel Project from Program Code

Updates the model (the UModel Project) from the program code. The name of this command can either be **Merge UModel Project from Program Code** or **Overwrite UModel Project from Program Code**, depending on the settings in the Synchronization Settings dialog box. By default, the Synchronization Settings dialog box opens every time when you run this command. For more information, see Code Synchronization Settings 221.

Project Settings

When generating program code into a UModel project, you may want to set or change project settings 175.

Synchronization Settings

Opens the Synchronization Settings dialog box, see Code Synchronization Settings 221.

Merge Project

Merges two UModel project files into one model. The first file you open is the one the second file will be merged into. Please see Merging UModel projects 276 for more information.

Merge Project (3-way)

UModel supports the merging of multiple UModel projects that have been simultaneously edited by different developers, in a 3-way project merge 276.

Include Subproject

See Including other UModel projects 164.

Open Subproject Individually

Opens the selected subproject as a new project.

Clear Messages

Clears the syntax check and code merging messages, warnings and errors from the Messages Window 96.

Note: Errors are generally problems that must be fixed before code can be generated, or the model code can be updated during the code engineering process. Warnings can generally be deferred until later. Errors and warnings are generated by the syntax checker, the compiler for the specific language, the UModel parser that reads the newly generated source file, as well as during the import of XMI files.

Generate Documentation

Generates documentation for the currently open project in HTML, Microsoft Word, and RTF formats, see Generating UML documentation 285.

List Elements not used in any Diagram

Creates a list of all elements not used in any diagram in the project, see Checking Where and If Elements Are Used 116.

List shared Packages

Lists all shared packages of the current project.

List included Packages

Lists all include packages in the current project.

© 2019-2025 Altova GmbH

15.4 Layout

The commands of the Layout menu allow you to line up and align the elements of your modeling diagrams, see Aligning and Resizing Modeling Elements 130.

Align

The align command allows you to align modeling elements along their borders, or centers depending on the specific command you select.

Space Evenly

This set of commands allow you to space selected elements evenly both horizontally and vertically.

Make Same Size

This set of commands allow you to adjust the width and height of selected elements based on the active element.

Line Up

This set of commands allow you to line up the selected elements vertically or horizontally.

Line Style

This set of commands allow you to select the type of line used to connect the various modeling elements. The lines can be any type of dependency, association lines used in the various model diagrams.

Autosize

This command resizes the selected elements to their respective optimal size(s).

Autolayout all

This command arranges automatically the modeling elements on the diagram, using one of the options below.

Force Directed	Displays the modeling elements from a centric viewpoint.
Hierarchic	Displays elements according to their hierarchical relationships. For example, a superclass will be placed above any of its derived classes. The hierarchical layout options can be customized from the **Tools | Options** menu, **View** tab, **Autolayout Hierarchic** group.
Block	Displays elements grouped by element size in rectangular fashion.

Reposition Text Labels

Repositions modeling element names (of the selected elements) to their default positions.

15.5 View

The commands available in this menu allow you to:

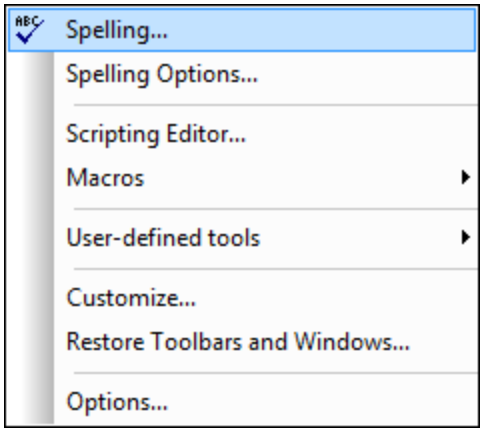
- Show or hide any of the UModel helper windows, see UModel Graphical User Interface 81
- Define the sort criteria of elements inside the Model Tree window 83 and Favorites window 88
- Define the grouping criteria of diagrams in the Diagram Tree window 87
- Show or hide specific UML elements in the Favorites window and Model Tree window
- Define the zoom factor of the current diagram, see Zooming into/out of Diagrams 135.

© 2019-2025 Altova GmbH

15.6 Tools

The commands available in this menu allow you to:

- Spell check your UModel project and define the spell checker options.
- Access the Scripting Environment[707] of UModel. You can create, manage and store your own forms, macros and event handlers.
- View and execute the currently defined macros.
- Customize[675] the interface: define your own toolbars, keyboard shortcuts, menus, and macros.
- Restore toolbars and windows to their default state.
- Define the global program settings/options[685].

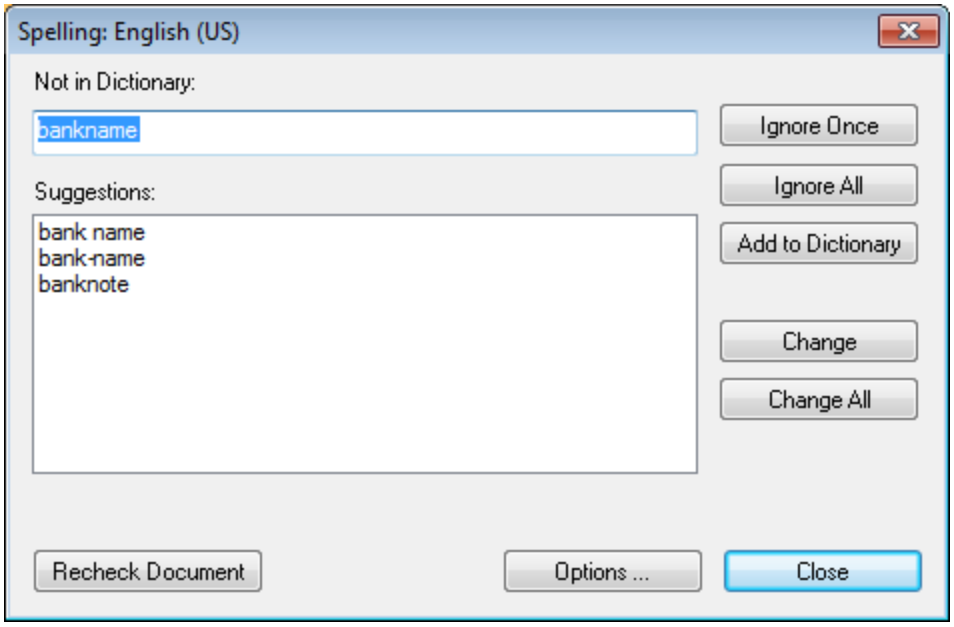


15.6.1 Spelling

Select **Tools | Spelling** to start the spell check process. The standard spell checker options are available in this dialog box.

To define the specific spell checker options, click **Options** in the **Spelling** dialog or select the menu command **Tools | Spelling Options**.

You can spell check entries in the Model Tree as well as in UML diagrams. Right clicking in the Model Tree and selecting "Documentation Spelling" spell checks the comments and notes of the Model Tree.



Not in Dictionary

This text box contains the word that cannot be found in either the selected language dictionary or user dictionary.

Suggestions

This list box displays words resembling the unknown word (supplied from the language and user dictionaries). Double-clicking a word in this list automatically inserts it in the document and continues the spell-checking process.

Ignore once

This command allows you to continue checking the document while ignoring the first occurrence of the unknown word. The same word will be flagged again if it appears in the document.

Ignore all

This command ignores all instances of the unknown word in the whole document.

Add to dictionary

This command adds the unknown word to the **user dictionary**. You can access the user dictionary (in order to edit it) via the Options dialog.

Change

This command replaces the currently highlighted word in the XML document with the (edited) word in the *Not in Dictionary* text box.

© 2019-2025 Altova GmbH

Change all

This command replaces all occurrences of the currently highlighted word in the XML document with the (edited) word in the *Not in Dictionary* text box.

Recheck Document

The "Recheck Document" button restarts the check from the beginning of the document.

Adding dictionaries for the spellchecker

For each dictionary language there are two Hunspell dictionary files that work together: a `.aff` file and `.dic` file. All language dictionaries are installed in a `Lexicons` folder at the following location: `C:\ProgramData\Altova\SharedBetweenVersions\SpellChecker\Lexicons`.

Within the `Lexicons` folder, different language dictionaries are each stored in a different folder: `<language name>\<dictionary files>`. For example, files for the two English-language dictionaries (`English (British)` and `English (US)`) will be stored as below:

```
C:\ProgramData\Altova\SharedBetweenVersions\SpellChecker\Lexicons\English (British)
\en_GB.aff
C:\ProgramData\Altova\SharedBetweenVersions\SpellChecker\Lexicons\English (British)
\en_GB.dic
C:\ProgramData\Altova\SharedBetweenVersions\SpellChecker\Lexicons\English (US)\en_US.aff
C:\ProgramData\Altova\SharedBetweenVersions\SpellChecker\Lexicons\English (US)\en_US.dic
```

In the Spelling Options dialog, the dropdown list of the *Dictionary Language* combo box displays the language dictionaries. These dictionaries are those available in the `Lexicons` folder and have the same names as the language subfolders in the `Lexicons` folder. For example, in the case of the English-language dictionaries shown above, the dictionaries would appear in the Dictionary Language combo box as: *English (British)* and *English (US)*.

All installed dictionaries are shared by the different users of the machine and the different major versions of Altova products (whether 32-bit or 64-bit).

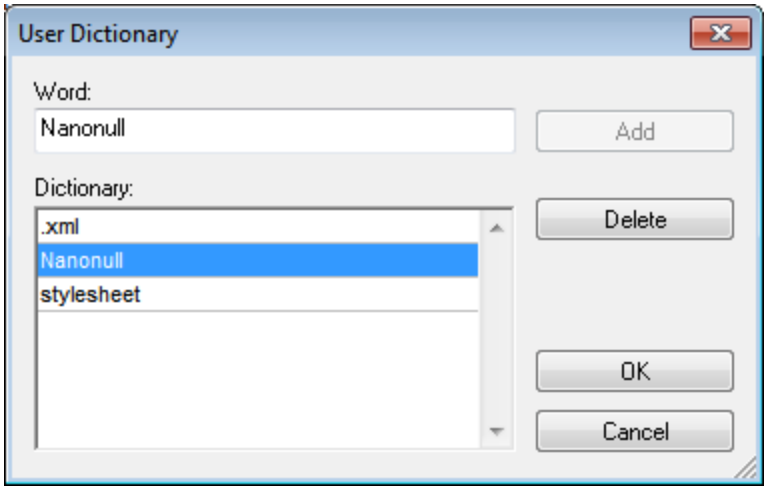
You can add dictionaries for the spellchecker in two ways, neither of which require that the files be registered with the system:

- By adding Hunspell dictionaries into a new subfolder of the `Lexicons` folder. Hunspell dictionaries can be downloaded, for example, from https://wiki.openoffice.org/wiki/Dictionaries or http://extensions.services.openoffice.org/en/dictionaries. (Note that OpenOffice uses the zipped `OXT` format. So change the extension to `.zip` and unzip the `.aff` and `.dic` file to the language folders in the `Lexicons` folder. Also note that Hunspell dictionaries are based on Myspell dictionaries. So Myspell dictionaries can also be used.)
- By using the Altova dictionary installer, which installs a package of multiple language dictionaries by default to the correct location on your machine. The installer can be downloaded via the link in the Dictionary language pane of the Spelling Options dialog (*see screenshot below*). Installation of the dictionaries must be done with administrator rights, otherwise installation will fail with an error.

Note: It is your choice as to whether you agree to the terms of the license applicable to the dictionary and whether the dictionary is appropriate for your use with the software on your computer.

Working with the user dictionary

Each user has one user dictionary, in which user-allowed words can be stored. During a spellcheck, spellings are checked against a word list comprising the words in the language dictionary and the user dictionary. You can add words to and delete words from the user dictionary via the User Dictionary dialog (*screenshot below*). This dialog is accessed by clicking the User Dictionary button in the Spelling Options dialog (*see second screenshot in this section*).



To add a word to the user dictionary, enter the word in the Word text box and click **Add**. The word will be added to the alphabetical list in the Dictionary pane. To delete a word from the dictionary, select the word in the Dictionary pane and click **Delete**. The word will be deleted from the Dictionary pane. When you have finished editing the User Dictionary dialog, click **OK** for the changes to be saved to the user dictionary.

Words may also be added to the User Dictionary during a spelling check. If an unknown word is encountered during a spelling check, then the Spelling dialog 669 pops up prompting you for the action you wish to take. If you click the **Add to Dictionary** button, then the unknown word is added to the user dictionary.

The user dictionary is located at: `C:\Users\<user>\Documents\Altova\SpellChecker\Lexicons\user.dic`

© 2019-2025 Altova GmbH

15.6.2 Spelling Options

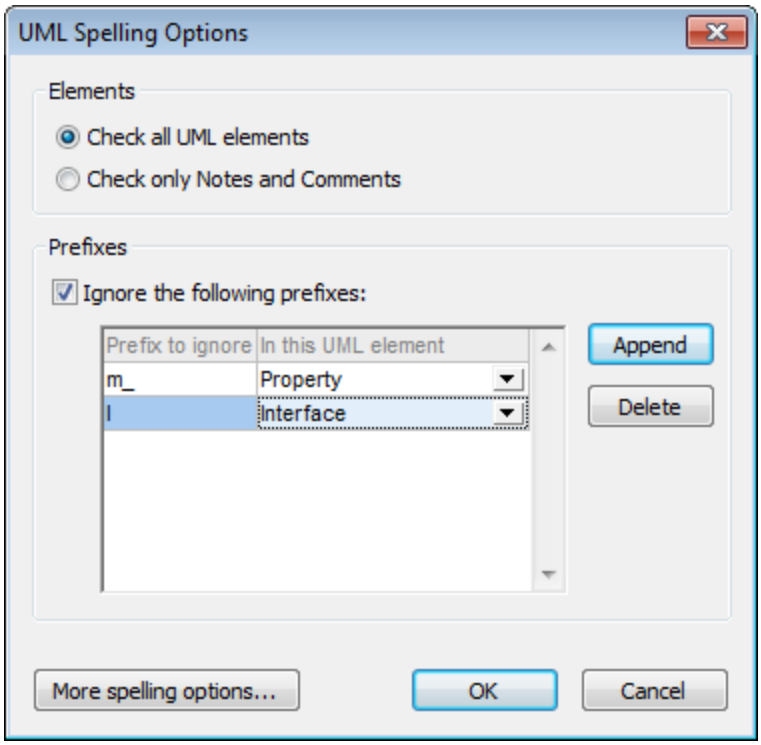
Elements

This group allows you to choose between spell checking all UML elements, or only the Notes and Comments objects.

Prefixes

Double clicking in the "Prefix to ignore" column lets you enter the prefixes, of specific UML elements, you want to ignore during spell checking, e.g. m_ for properties, and I for Interfaces.

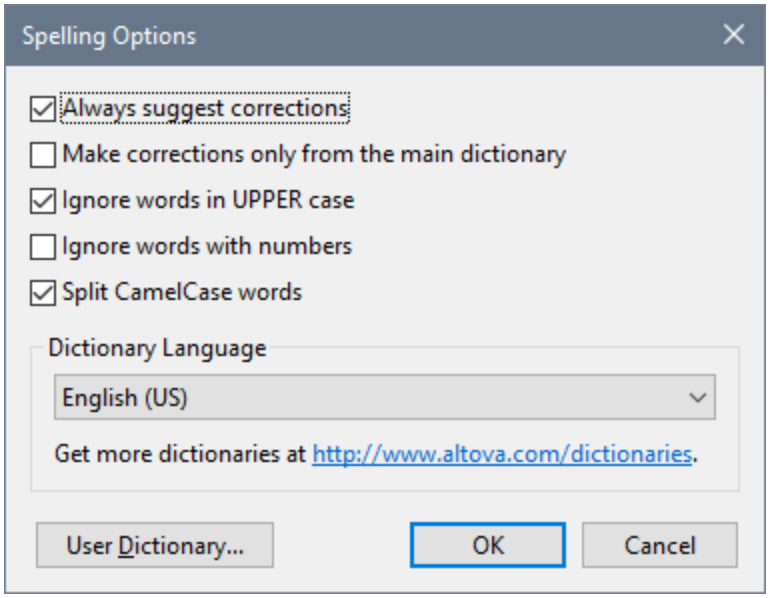
The "Append" button adds a new row to the Prefixes table. "Delete" deletes the currently active row.



Clicking the "More spelling options..." button opens the Spelling Options dialog box shown below.

More Spelling Options

The Spelling Options dialog is used to define global spellchecker options.



Always suggest corrections:
Activating this option causes suggestions (from both the language dictionary and the user dictionary) to be displayed in the Suggestions list box. Disabling this option causes no suggestions to be shown.

Make corrections only from main dictionary:
Activating this option causes only the language dictionary (main dictionary) to be used. The user dictionary is not scanned for suggestions. It also disables the **User Dictionary** button, preventing any editing of the user dictionary.

Ignore words in UPPER case:
Activating this option causes all upper case words to be ignored.

Ignore words with numbers:
Activating this option causes all words containing numbers to be ignored.

Split CamelCase words
CamelCase words are words that have capitalization within the word. For example the word "CamelCase" has the "C" of "Case" capitalized, and is therefore said to be CamelCased. Since CamelCased words are rarely found in dictionaries, the spellchecker would flag them as errors. To avoid this, the *Split CamelCase words* option splits CamelCased words into their capitalized components and checks each component individually. This option is checked by default.

Dictionary Language
Use this combo box to select the dictionary language for the spellchecker. The default selection is US English. Other language dictionaries are available for download free of charge from the Altova website.

© 2019-2025 Altova GmbH

15.6.3 Scripting Editor

The Scripting Editor command opens the Scripting Editor window, see Scripting Editor[707].

Note: The .NET Framework version 2.0 or higher must be installed on your machine in order for the Scripting Editor to run.

15.6.4 Macros

Displays a list of macros that are currently defined in the Scripting Project, see Scripting Editor[707]. The active Scripting Project is defined in the Scripting tab[691] of the Options dialog box.

15.6.5 User-defined Tools

Placing the cursor over the **User-defined Tools** command rolls out a sub-menu containing custom-made commands that use external applications. You can create these commands in the Tools tab[678] of the Customize dialog. Clicking one of these custom commands executes the action associated with this command.

The **User-Defined Tools | Customize** command opens the Tools tab[678] of the Customize dialog (in which you can create the custom commands that appear in the menu of the **User-Defined Tools** command.)

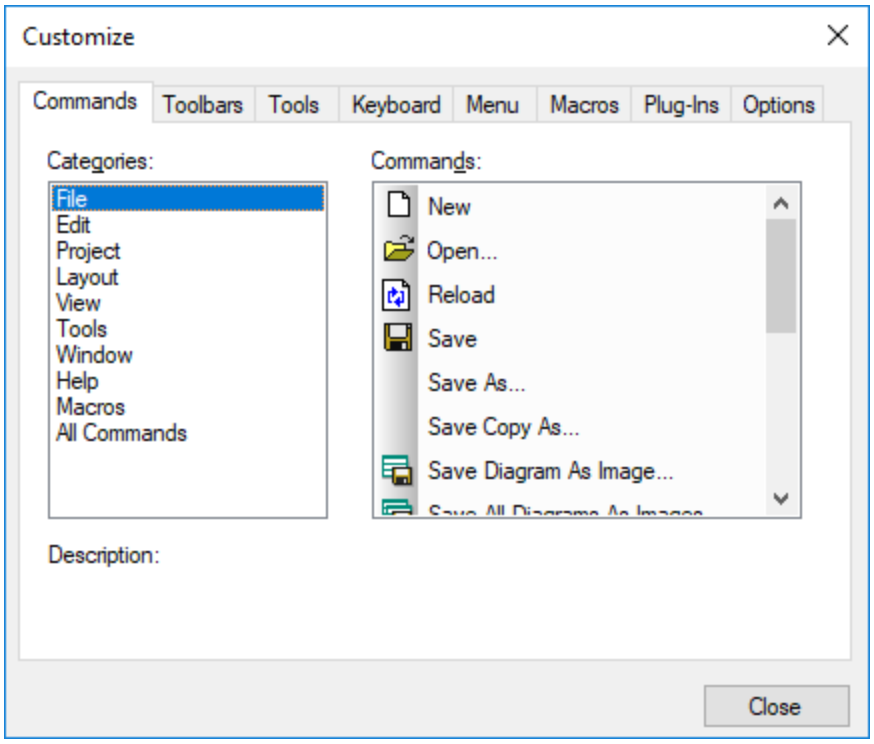
15.6.6 Customize

The **Customize** command displays a dialog box from where you can customize UModel to suit your personal needs. You can customize the following entities:

- Commands[676]
- Toolbars[677]
- Tools[678]
- Keyboard[682]
- Menu[683]
- Macros[684]
- Plug-ins[684]
- Options[684]

15.6.6.1 Commands

The **Commands** tab allows you customize UModel menus or toolbars.



To add a command to a toolbar or menu:

1. On the **Tools** menu, click **Customize**.
2. Select the command category in the **Categories** list box. The commands available appear in the **Commands** list box.
3. Click a command in the **Commands** list box and drag it to an existing menu or toolbar. An I-beam appears when you place the cursor over a valid position to drop the command.
4. Release the mouse button at the position you want to insert the command. A small button appears at the tip of mouse pointer when you drag a command. The check mark below the pointer means that the command cannot be dropped at the current cursor position. The check mark disappears whenever you can drop the command (over a toolbar or menu).

Notes:

- Placing the cursor over a menu when dragging, opens it, allowing you to insert the command anywhere in the menu.
- Commands can be placed in menus or tool bars. If you created you own toolbar, you can populate it with your own commands/icons.
- You can also edit the commands in the context menus 683 (right-click anywhere to open the context menu), using the same method. Click the **Menu** tab and then select the specific context menu available in the Context Menus combo box.

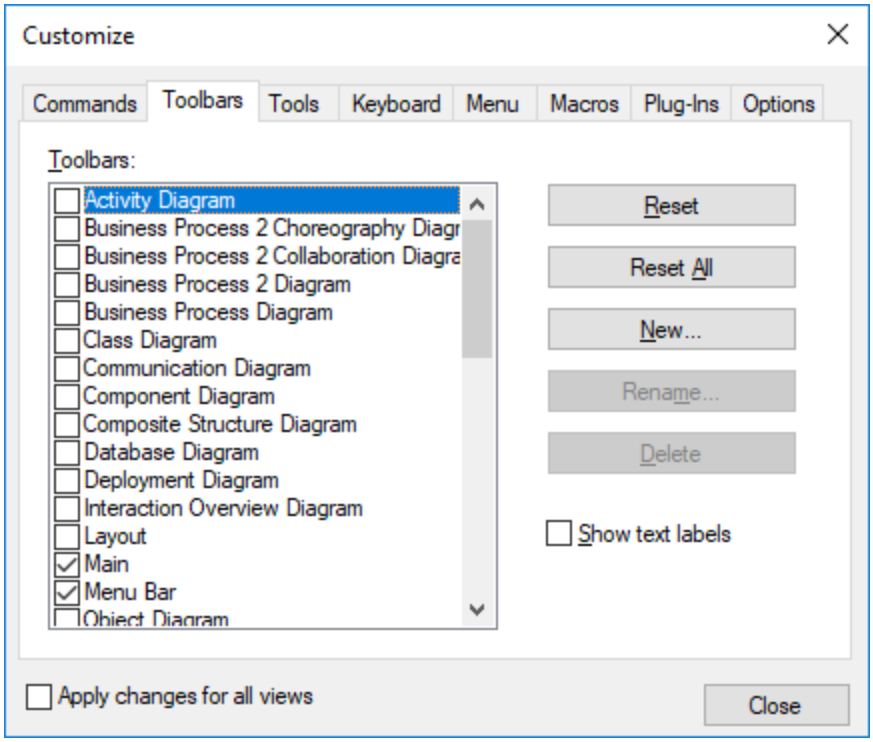
© 2019-2025 Altova GmbH

To delete a command or menu:

1. On the **Tools** menu, click **Customize**.
2. Click the menu entry or icon you want to delete, and drag with the mouse.
3. Release the mouse button whenever the check mark icon appears below the mouse pointer. The command (or menu item) is deleted from the menu or tool bar.

15.6.6.2 Toolbars

The **Toolbars** tab allows you to activate or deactivate specific toolbars, as well as create your own specialized ones.



Toolbars contain symbols for the most frequently used menu commands. For each symbol, you get a brief "tool tip" explanation when the mouse cursor is directly over the item and the status bar shows a more detailed description of the command. You can drag the toolbars from their standard position to any location on the screen, where they appear as a floating window. Alternatively, you can also dock them to the left or right edge of the main window.

To activate or deactivate a toolbar:

- Click the check box to activate (or deactivate) the specific toolbar.

To create a new toolbar:

1. Click the **New...** button, and give the toolbar a name in the Toolbar name dialog box.
2. Add commands to the toolbar using the Commands 676 tab of the Customize dialog box.

To reset the Menu Bar:

1. Click the **Menu Bar** entry, and
2. Click the **Reset** button, to reset the menu commands to the state they were when installed.

To reset all toolbar and menu commands:

1. Click the **Reset All** button, to reset all the toolbar commands to the state they were when the program was installed. A prompt appears stating that all toolbars and menus will be reset.
2. Click **Yes** to confirm the reset.

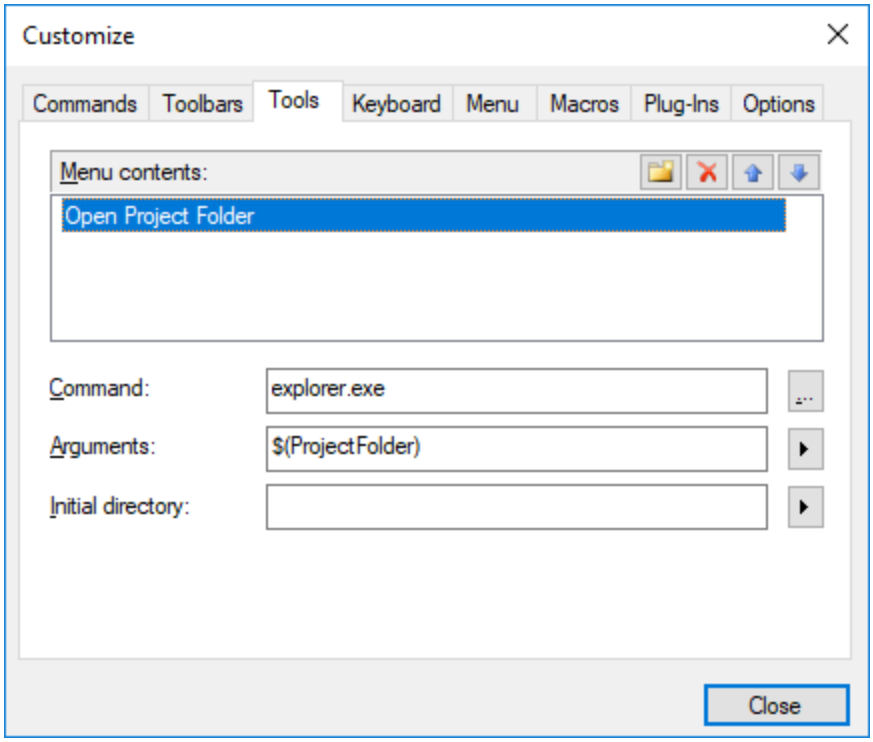
The **Show text labels** option places explanatory text below toolbar icons when activated.

15.6.6.3 Tools

The **Tools** tab allows you to create custom menu commands that can start external tools directly from UModel. The custom menu commands that you define here appear under the menu **Tools | User-defined tools**. External tools can be programs included with Windows, such as Windows Explorer (**explorer.exe**), Notepad (**notepad.exe**), or other custom executables. You can optionally assign arguments to each user-defined tool and set the directory where the external tool should initialize (in order to look for relative file names).

For example, the configuration illustrated below adds a new menu command called "Open Project Folder". When run, this command will open the directory of the current UModel project in Windows Explorer.

© 2019-2025 Altova GmbH



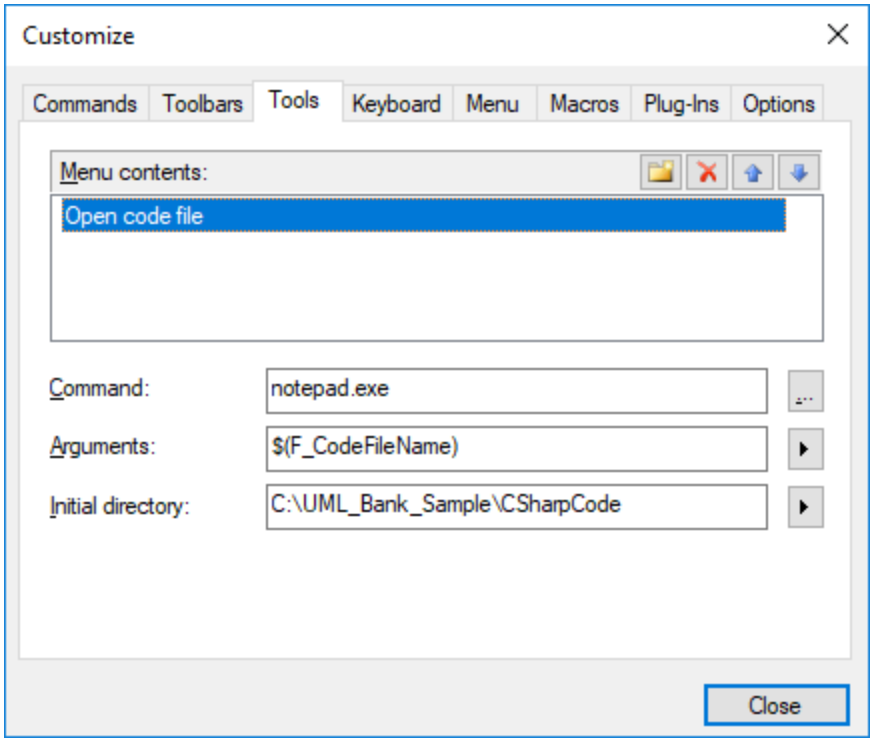
When an external tool takes arguments (like Windows Explorer in the example above), these can be entered in the Arguments input box. To supply multiple arguments, separate them with the space character. The values you can supply as arguments can be plain text (hard-coded values) or be selected with the ▸ button from a list of predefined UModel variables. You can use any of the following UModel predefined variables as arguments:

UModel predefined variable	Purpose
Project File Name	The file name of the active UModel project file, for example **Test.ump**.
Project File Path	The absolute file path of the active UModel project file, for example, **C:\MyDirectory\Test.ump**.
Focused UML Data – Name	The name of the currently focused UML element, for example, **Class1**.
Focused UML Data – UML Qualified Name	The qualified name of the currently focused UML element, for example, **Package1::Package2::Class1**.
Focused UML Data – Code File Name	The code file name of the currently focused UML class, interface or enumeration as shown in the Property window (relative to the realizing component), for example, **Class1.cs** or **MyNamespace\Class1.Java**.

UModel predefined variable	Purpose
Focused UML Data – Code File Path	The code file path of the currently focused UML class, interface or enumeration as shown in the Property window, for example, **C:\Temp\MySource\Class1.cs**.
Focused UML Data – Code Project File Name	The file name of the code project to which the currently focused UML class, interface or enumeration belongs. The code project file name can be relative to the UModel project file and is the same as shown in the Properties of the component, for example, **C:\Temp\MySource\MyProject.vcproj** or **MySource\MyProject.vcproj**.
Focused UML Data – Code Project File Path	The file path of the code project to which the currently focused UML class, interface or enumeration belongs, for example, **C:\Temp\MySource\MyProject.vcproj**.
Project Folder	The directory where the current UModel project is saved, for example, **C:\Users\<user>\Documents\Altova\UModel2025\UModelExamples**.
Temporary Folder	The directory where the application's temporary files are saved, for example, **C:\Users\<user>\AppData\Local\Temp**.

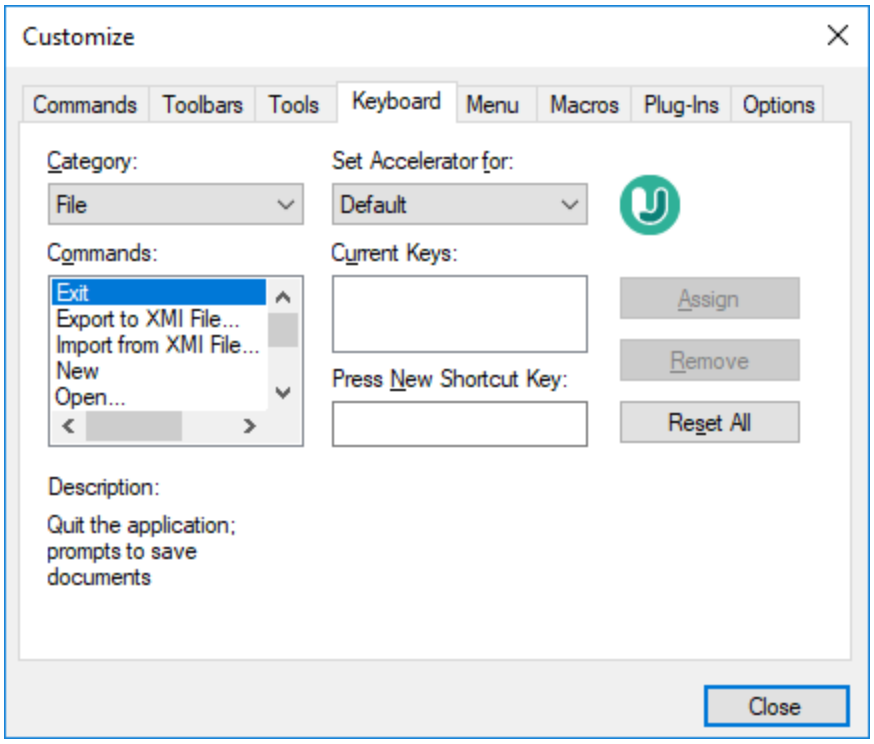
In some cases, you may also need to enter a value in the **Initial Directory** input box. For example, the configuration below opens in Notepad the code file of the currently selected element on a diagram. (Note that, for this command to work, the element currently selected on the diagram must have a value (file name) defined in the **code file name** field of the Properties Window 89, and that file must exist in **C:\UML_Bank_Sample\CSharpCode** directory).

© 2019-2025 Altova GmbH


Customize
Commands
Toolbars
Tools
Keyboard
Menu
Macros
Plug-Ins
Options
Menu contents:
Open code file
Command:
notepad.exe
Arguments:
$(F_CodeFileName)
Initial directory:
C:\UML_Bank_Sample\CSharpCode
Close

15.6.6.4 Keyboard

The **Keyboard** tab allows you to define (or change) keyboard shortcuts for any command.



To assign a new Shortcut to a command:

1. Select a value from the **Category** combo box.
2. Select the command you want to assign a new shortcut to, in the **Commands** list box.
3. Click inside the **Press New Shortcut Key** text box, and press the shortcut keys that are to activate the command. The shortcuts appear immediately in the text box. If the shortcut was assigned previously, then that function is displayed below the text box.
4. Click **Assign** to permanently assign the shortcut. The shortcut now appears in the **Current Keys** list box. (To clear this text box, press any of the control keys, **Ctrl**, **Alt** or **Shift**).

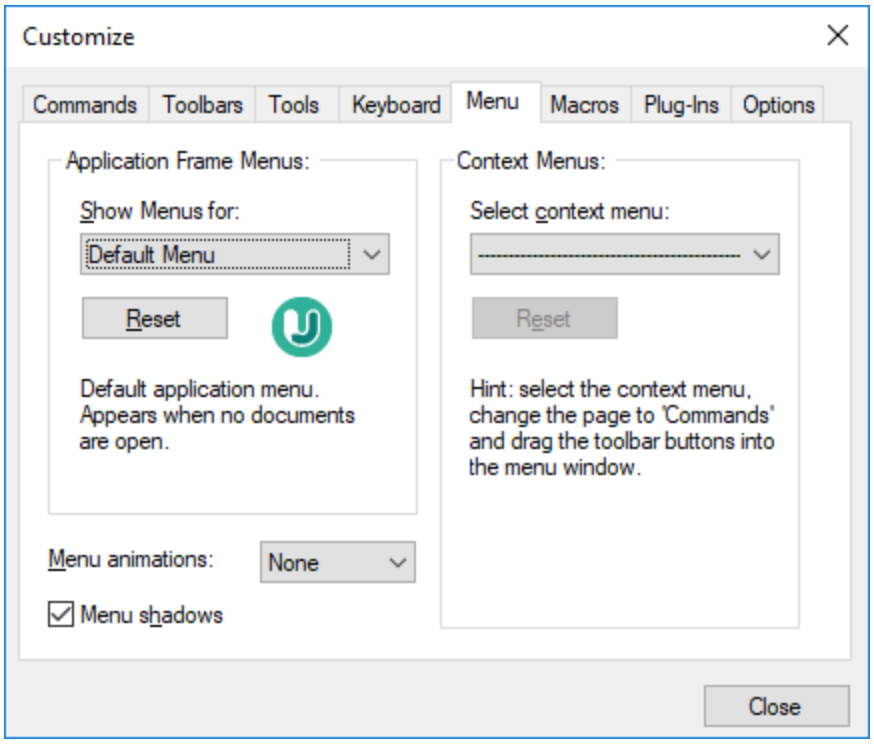
To de-assign (delete) a shortcut:

1. Click the shortcut you want to delete in the **Current Keys** list box, and
2. Click the **Remove** button (which has now become active).
3. Click **Close** to confirm all the changes made in the Customize dialog box.

© 2019-2025 Altova GmbH

15.6.6.5 Menu

The **Menu** tab allows you to customize the menu bars as well as the context menus.



Customizing menus

The **Default Menu** bar is the menu bar that is displayed when no project is open. The **UModel project** menu bar is the menu bar that is displayed when a project is open. Each menu bar can be customized separately, and customization changes made to one do not affect the other.

To customize a menu bar, select it from the **Show Menus For** drop-down list. Then click the **Commands** tab and drag commands from the **Commands** list box to the menu bar or into any of the menus.

Deleting commands from menus and resetting the menu bars

To delete an entire menu or a command inside a menu, do the following:

1. Select from the **Show Menus for** drop-down list the menu bar that is to be customized.
2. With the Customize dialog open, select (i) the menu you want to delete from the application's menu bar, or (ii) the command you want to delete from one of these menus.
3. Either (i) drag the menu from the menu bar or the menu command from the menu, or (ii) right-click the menu or menu command and select **Delete**.

You can reset any menu bar to its original installation state by selecting it from the **Show Menus For** drop-down list and then clicking the **Reset** button.

Customizing the application's context menus

Context menus are the menus that appear when you right-click certain objects in the application's interface. Each of these context menus can be customized by doing the following:

1. Select the context menu from the **Select context menu** drop-down list. This pops up the context menu.
2. Click the **Commands** tab.
3. Drag a command from the **Commands** list box into the context menu.
4. To delete a command from the context menu, right-click that command in the context menu, and select **Delete**. Alternatively, drag the command out of the context menu.

You can reset any context menu to its original installation state by selecting it in the **Select context menu** drop-down list and then clicking the **Reset** button.

Menu shadows

Select the **Menu shadows** check box to give all menus shadows.

You can choose from among several menu animations if you prefer animated menus. The **Menu animations** drop-down list provides the following options:

- None (default)
- Unfold
- Slide
- Fade

15.6.6.6 Macros

The **Macros** tab allows you to select from the macros defined in the Scripting Project that is currently active in UModel.

The active Scripting Projects are specified in the Scripting tab of the Options dialog, or in the Scripting tab 691 of the project settings.

15.6.6.7 Plug-Ins

The **Plug-Ins** tab allows you to add or remove a UModel Plug-in (.dll file) which integrates with UModel, see UModel IDE Plug-Ins 733.

15.6.6.8 Options

The **Options** tab allows you to set general environment settings.

© 2019-2025 Altova GmbH

When active, the **Show ScreenTips on toolbars** check box displays a tooltip label when the mouse pointer is placed over a toolbar button. The label contains a short description of the button function. If the **Show shortcut keys in ScreenTips** check box is selected, the tooltip label displays the associated keyboard shortcut, if one has been assigned.

When active, the **Large Icons** check box switches between the standard size icons, and larger versions of the icons.

15.6.7 Restore Toolbars and Windows

The **Restore Toolbars and Windows** command closes down UModel and re-starts it with the default settings. Before it closes down a dialog pops up asking for confirmation about whether UModel should be restarted.

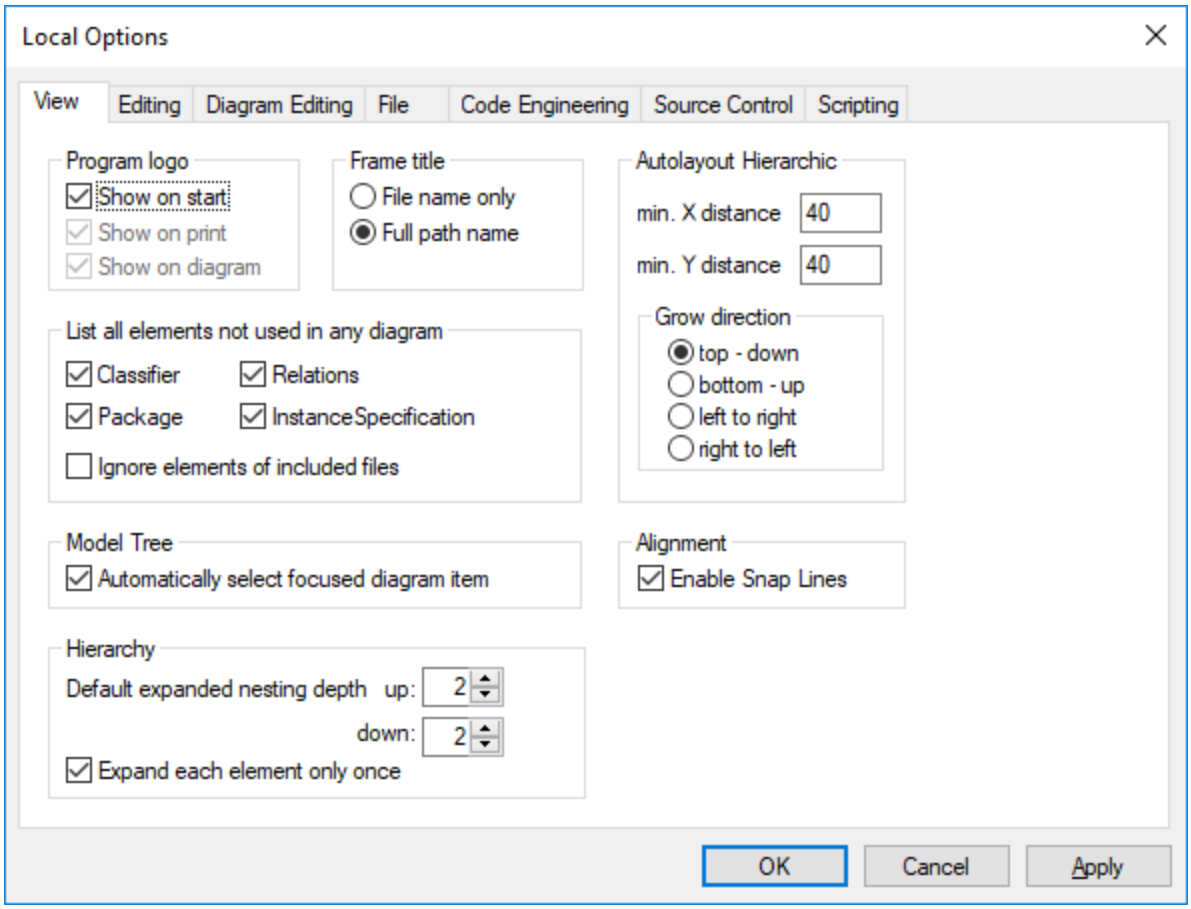
This command is useful if you have been resizing, moving, or hiding toolbars or windows, and would now like to have all the toolbars and windows as they originally were.

15.6.8 Options

Select the menu item **Tools | Options** to define your project options.

The **View** tab allows you to define:

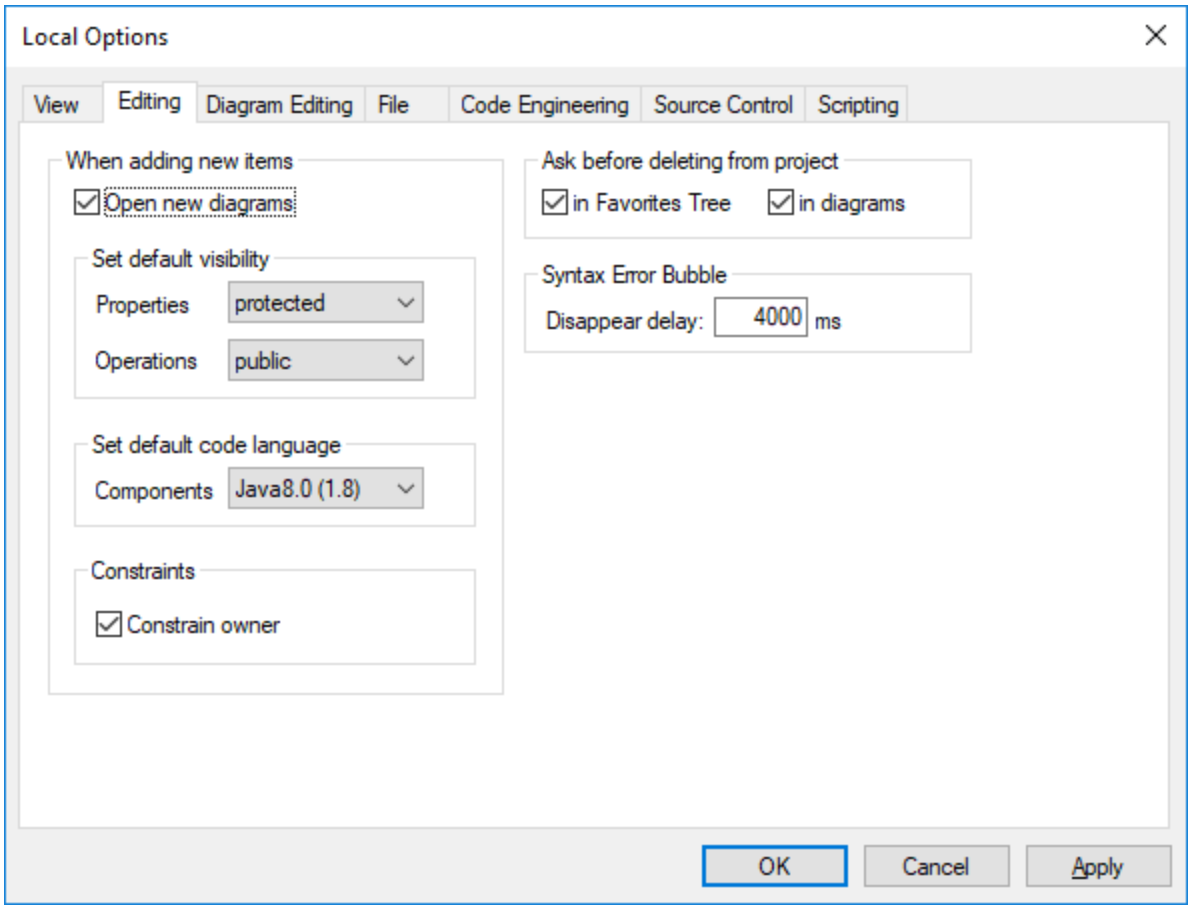
- Where the program logo should appear.
- The application title bar contents.
- The types of elements you want listed when using the "List elements not used in any diagram" context menu option in the Model Tree, or Favorites tab. You also have the option of ignoring elements contained in included files.
- If a selected element in a diagram is automatically selected/synchronized in the Model Tree.
- The default depth of the hierarchy view when using the **Show graph view** in the **Hierarchy** tab.
- The Autolayout Hierarchic settings, which allow you to define the nesting depth up and down in the hierarchy window.
- "Expand each element only once", only allows one of the same classifiers to be expanded in the same image/diagram.
- If you want snap lines to help you align elements when dragging in a diagram.



The **Editing** tab allows you to define:

- If a new Diagram created in the Model Tree tab, is also automatically opened in the main area.
- Default visibility settings when adding new elements - Properties or Operations.
- The default code language when a new component is added.
- If a newly added constraint, is to automatically constrain its owner as well.
- If a prompt should appear when deleting elements from a project, from the Favorites tab or in any of the diagrams. This prompt can be deactivated when deleting items there; this option allows you to reset the "prompt on delete" dialog box.
- The delay with which the syntax error pop-up message should be closed.

© 2019-2025 Altova GmbH

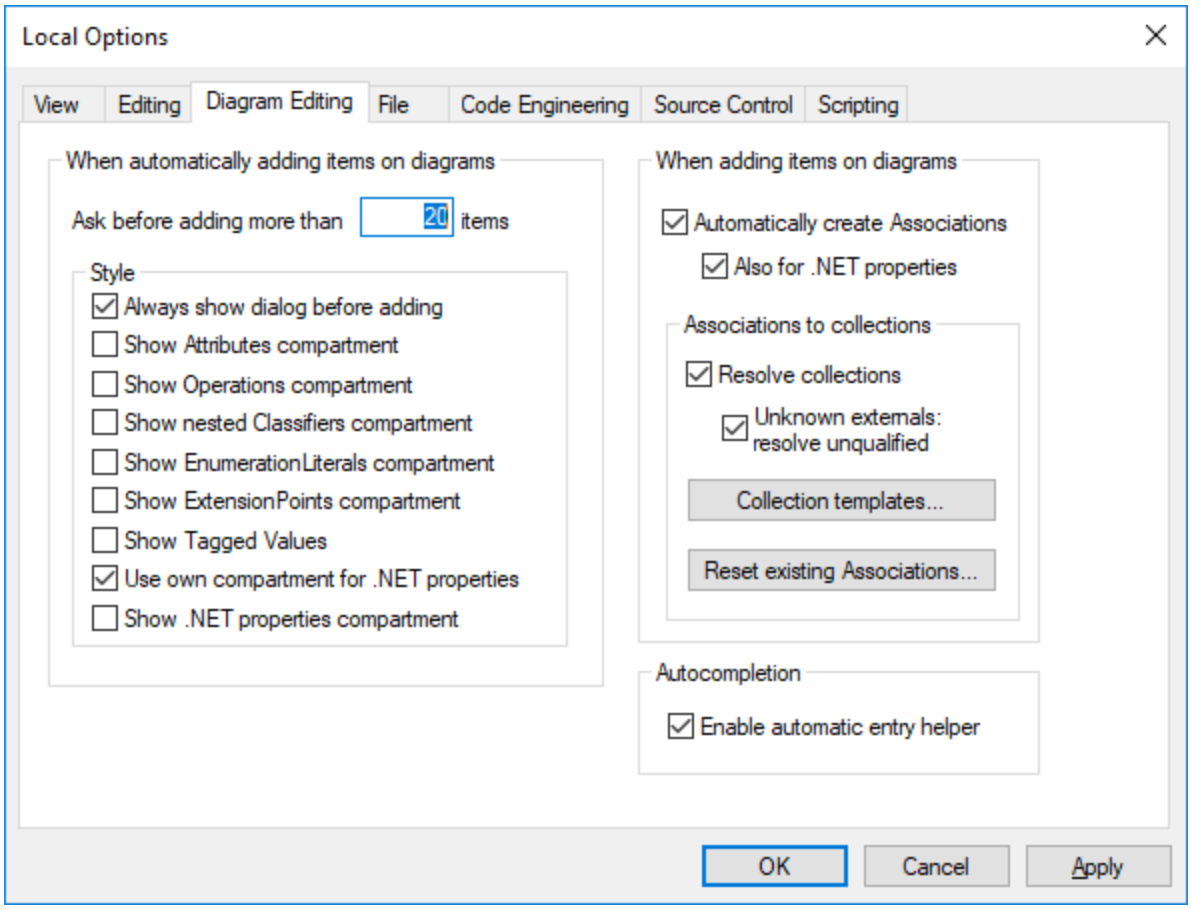


The **Diagram Editing** tab allows you to define:

- The number of items that can be automatically added to a diagram, before a prompt appears.
- The display of Styles when they are automatically added to a diagram.
- If Associations between modeling elements, are to be created automatically when items are added to a diagram.
- If the associations to collections are to be resolved.
- If templates from unknown externals are to be resolved as not fully qualified.
- or use preexisting Collection Templates, or define new ones.

  Collection Templates should be defined as fully qualified i.e. a.b.c.List. If the template has this namespace then UModel automatically creates a Collection Association. Exception: If the template belongs to the Unknown Externals package, and the option "Unknown externals: resolve unqualified", is enabled, then only the template name is considered (i.e. List instead of a.b.c.List).

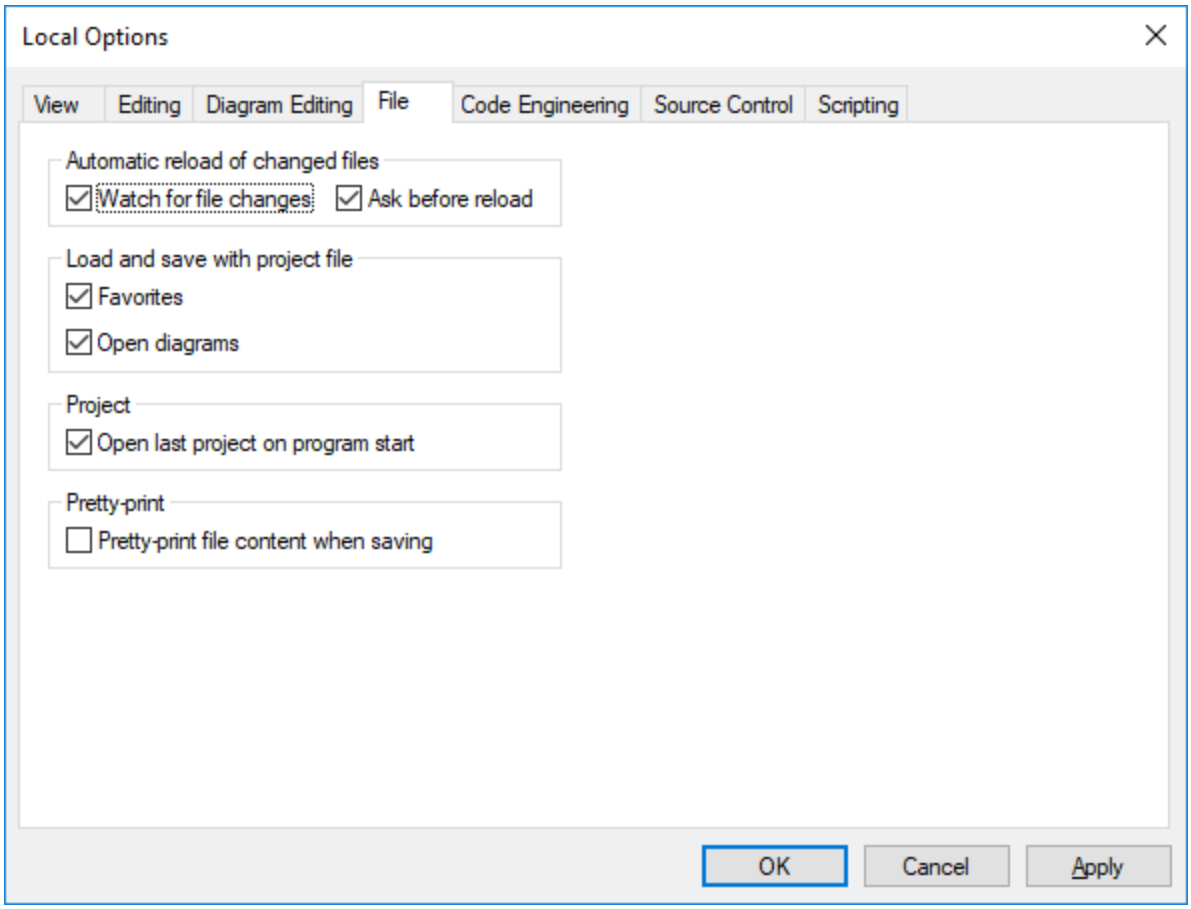
- If the autocompletion window is to be available when editing attributes or operations in the class diagram.



The **File** tab allows you to define:

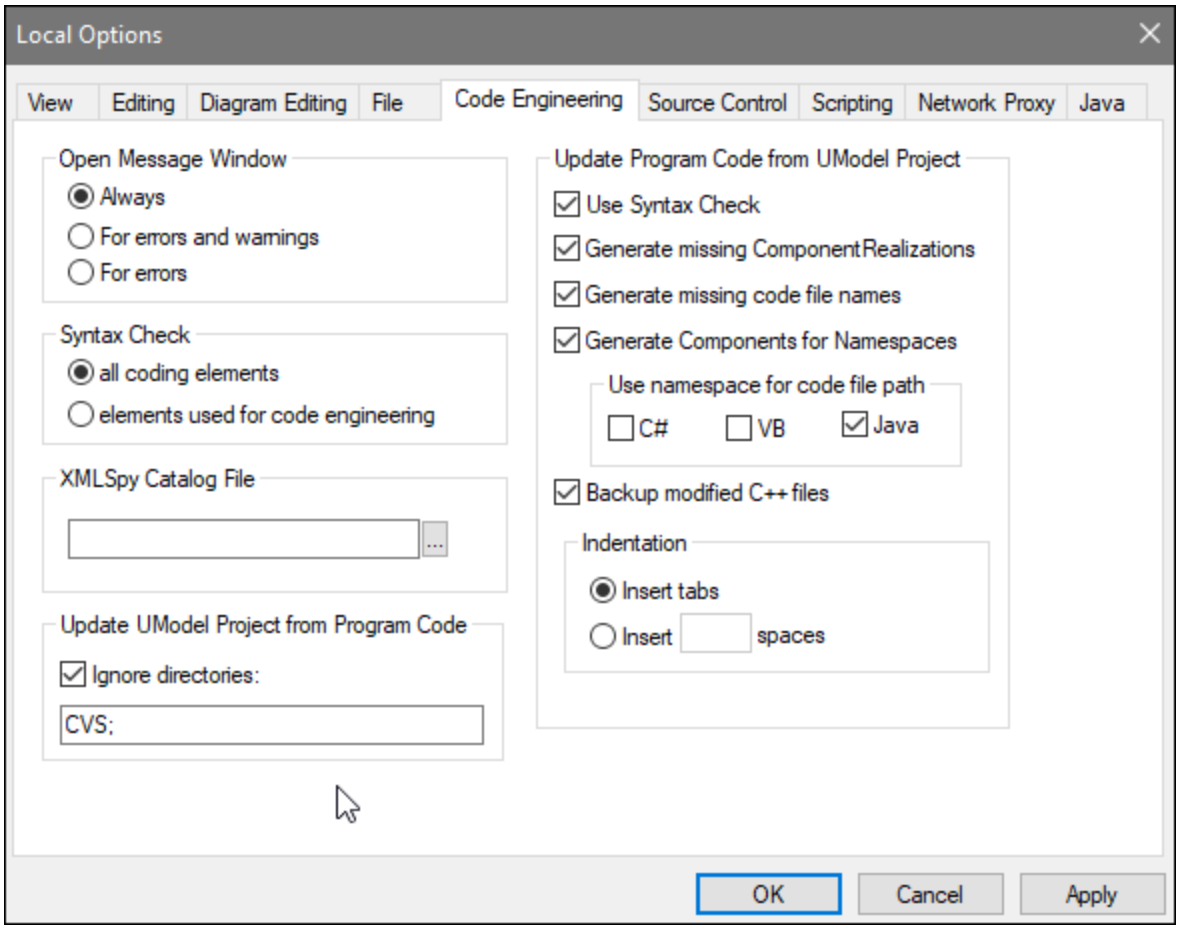
- The actions performed when files are changed.
- If the contents of the Favorites tab are to be loaded and saved with the current project, as well as the any currently open diagrams.
- If the previously opened project is to automatically be opened when starting the application.
- If you want to structure the project file with CR/LF and tab indents in a pretty-print format.

© 2019-2025 Altova GmbH



The **Code Engineering** tab allows you to define the following parameters:

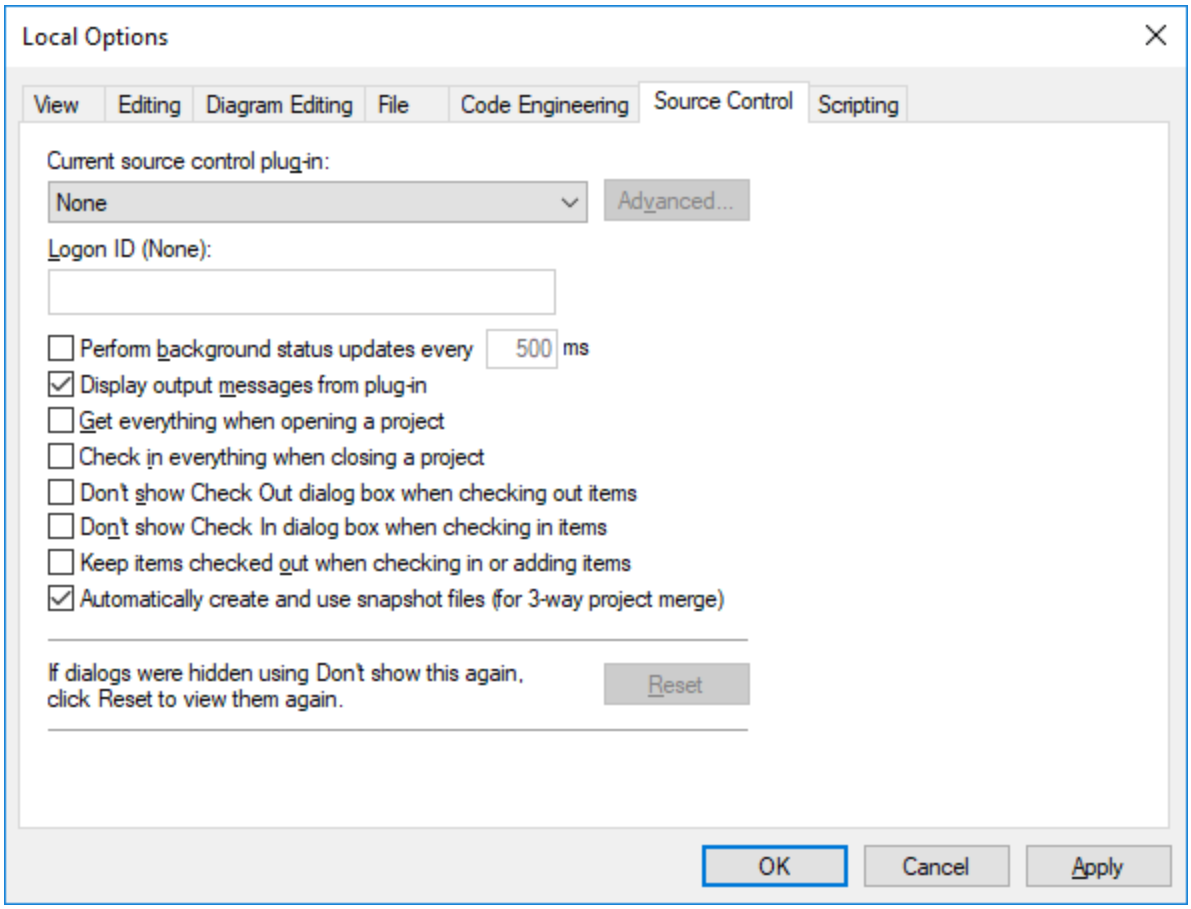
- The circumstances under which the Message window will open.
- If **all coding elements** i.e. those contained in a Java / C# / VB namespace root, as well as those assigned to a Java / C# / VB component, are to be checked, or only **elements used for code engineering**, i.e. where "use for code engineering" check box is active, are to be checked.
- When updating program code if:
  - If a syntax check is to be performed.
  - If missing ComponentRealizations are to be automatically generated.
  - If missing code file names in the merged code are to be generated.
  - If namespaces are to be used in the code file path.
- The Indentation method used in the code, i.e. tabs or any number of spaces.
- The directories to be ignored when updating a UModel project from code, or directory. Separate the respective directories with a semicolon "**;**". Child directories of the same name are also ignored.
- The location of the XMLSpy Catalog File, **RootCatalog.xml**, which enables UModel as well as XMLSpy to retrieve commonly used schemas (as well as stylesheets and other files) from local user folders. This increases the overall processing speed, and enables users to work offline.
- You can also specify whether you want to back up modified C++ files.



The **Source Control** tab allows you to define:

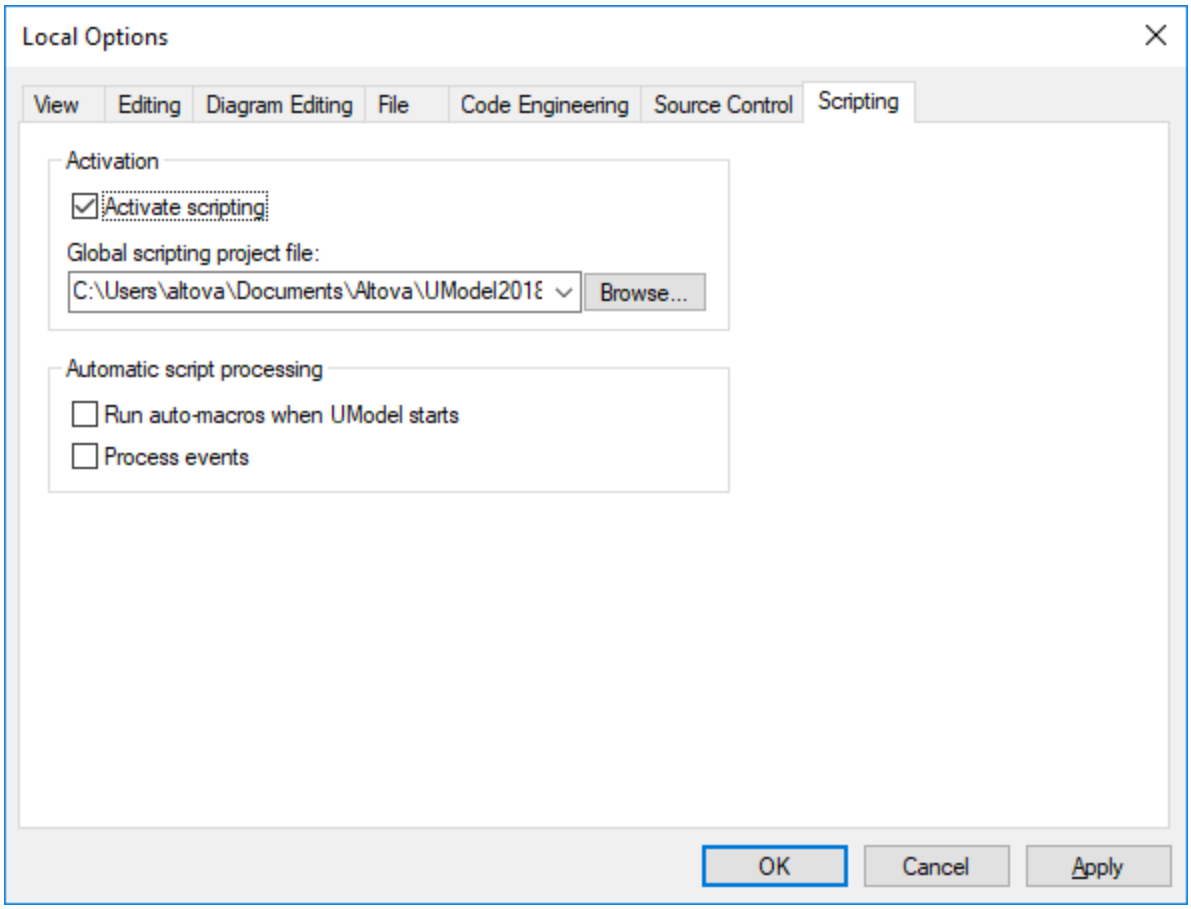
- The current source control plug-in using the combo box. The **Advanced** button allows you to define the specific settings of the source control plug-in that you selected. These settings change depending on the source control plug-in that you use.
- The login ID for the source control provider.
- Specific settings check in/out settings.
- The **Reset** button is made available if you have checked/activated the "Don't show this again" option in one of the dialog boxes. The **Don't show this again** prompt is then reenabled.

© 2019-2025 Altova GmbH



The **Scripting** tab allows you to define:

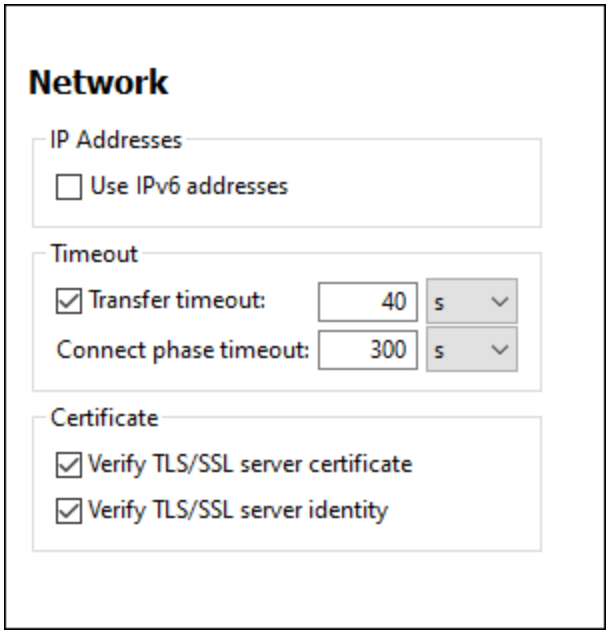
- If the Scripting environment[707] should be active for the current UModel project.
- Which Global scripting file you want to use
- If auto-macros are to be executed when UModel starts
- If Scripting events are to be processed.



For information about the settings available in the **Network Proxy** tab, see Network Proxy Settings [695]. To find out more about Java VM settings, see Java Virtual Machine Settings [694].

The **Network** tab:
The **Network** section (*screenshot below*) enables you to configure important network settings.

© 2019-2025 Altova GmbH



IP addresses
When host names resolve to more than one address in mixed IPv4/IPv6 networks, selecting this option causes the IPv6 addresses to be used. If the option is not selected in such environments and IPv4 addresses are available, then IPv4 addresses are used.

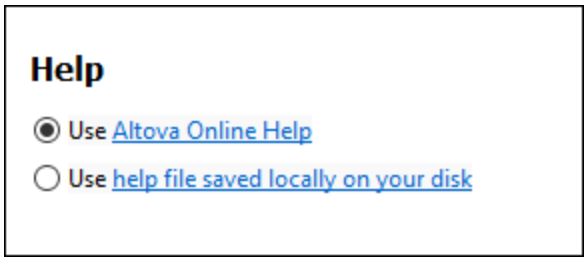
Timeout
- *Transfer timeout:* If this limit is reached for the transfer of any two consecutive data packages of a transfer (sent or received), then the entire transfer is aborted. Values can be specified in seconds [s] or milliseconds [ms], with the default being 40 seconds. If the option is not selected, then there is no time limit for aborting a transfer.
- *Connection phase timeout:* This is the time limit within which the connection has to be established, including the time taken for security handshakes. Values can be specified in seconds [s] or milliseconds [ms], with the default being 300 seconds. This timeout cannot be disabled.

Certificate
- *Verify TLS/SSL server certificate:* If selected, then the authenticity of the server's certificate is checked by verifying the chain of digital signatures until a trusted root certificate is reached. This option is enabled by default. If this option is not selected, then the communication is insecure, and attacks (for example, a man-in-the-middle attack) would not be detected. Note that this option does not verify that the certificate is actually for the server that is communicated with. To enable full security, both the certificate and the identity must be checked (*see next option*).
- *Verify TLS/SSL server identity:* If selected, then the server's certificate is verified to belong to the server we intend to communicate with. This is done by checking that the server name in the URL is the same as the name in the certificate. This option is enabled by default. If this option is not selected, then the server's identify is not checked. Note that this option does not enable verification of the server's certificate. To enable full security, both the certificate as well as the identity must be checked (*see previous option*).

The **Help** tab:
UModel provides Help (the user manual) in two formats:

- Online Help, in HTML format, which is available at the Altova website. In order to access the Online Help you will need Internet access.
- A Help file in PDF format, which is installed on your machine when you install UModel. It is named `UModel.pdf` and is located in the application folder (in the Program Files folder). If you do not have Internet access, you can always open this locally saved Help fie.

The Help option (*screenshot below*) enables you to select which of the two formats is opened when you click the **Help (F1)** command in the **Help** menu.



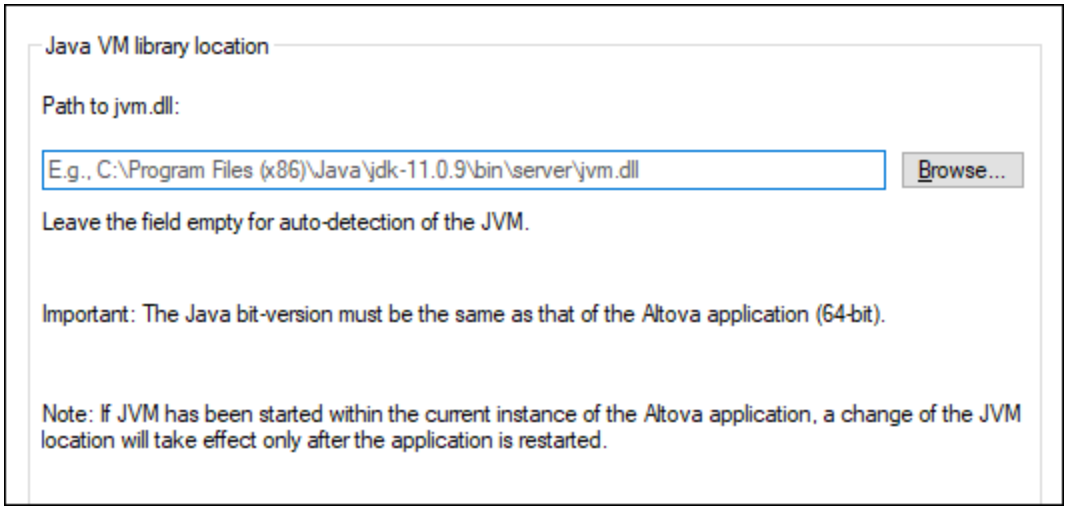
You can change this option at any time for the new selection to take effect. The links in this section (*see screenshot above*) open the respective Help format.

15.6.8.1 Java Virtual Machine Settings

In the *Java* section (*see screenshot below*), you can optionally enter the path to a Java VM (Virtual Machine) on your file system. Note that adding a custom Java VM path is not always necessary. By default, UModel attempts to detect the Java VM path automatically by reading (in this order) the Windows registry and the JAVA_HOME environment variable. The custom path added in this dialog box will take priority over any other Java VM path detected automatically.

You may need to add a custom Java VM path, for example, if you are using a Java virtual machine which does not have an installer and does not create registry entries (e.g., Oracle's OpenJDK). You might also want to set this path if you need to override, for whatever reason, any Java VM path detected automatically by UModel.

© 2019-2025 Altova GmbH



Note the following:

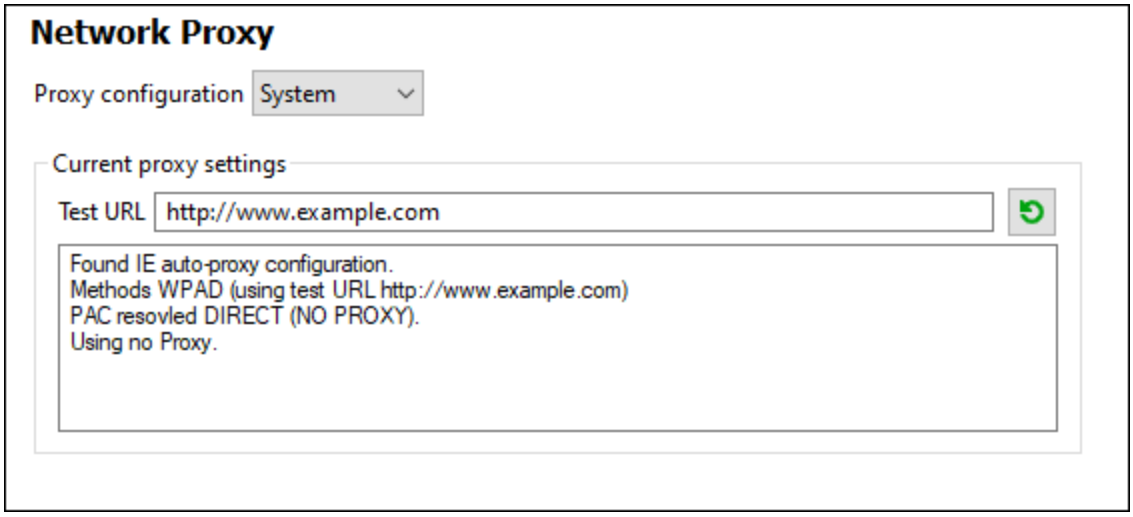
- The Java VM path is shared between Altova desktop (not server) applications. Consequently, if you change it in one application, it will automatically apply to all other Altova applications.
- The path must point to the `jvm.dll` file from the `\bin\server` or `\bin\client` directory, relative to the directory where the JDK was installed.
- The UModel platform (32-bit, 64-bit) must be the same as that of the JDK.
- After changing the Java VM path, you may need to restart UModel for the new settings to take effect.

Changing the Java VM path affects database connectivity via JDBC.This setting does not affect Java code generation and import. Note that the Java runtimes used for importing Java binaries into UModel can be configured separately. For more information, see Adding Custom Java Runtimes 205.

15.6.8.2 Network Proxy Settings

The *Network Proxy* section enables you to configure custom proxy settings. These settings affect how the application connects to the Internet (for XML validation purposes, for example). By default, the application uses the system's proxy settings, so you should not need to change the proxy settings in most cases. If necessary, however, you can set an alternative network proxy by selecting, in the *Proxy Configuration* combo box, either *Automatic* or *Manual* to configure the settings accordingly.

Note: The network proxy settings are shared among all Altova MissionKit applications. So, if you change the settings in one application, all MissionKit applications will be affected.



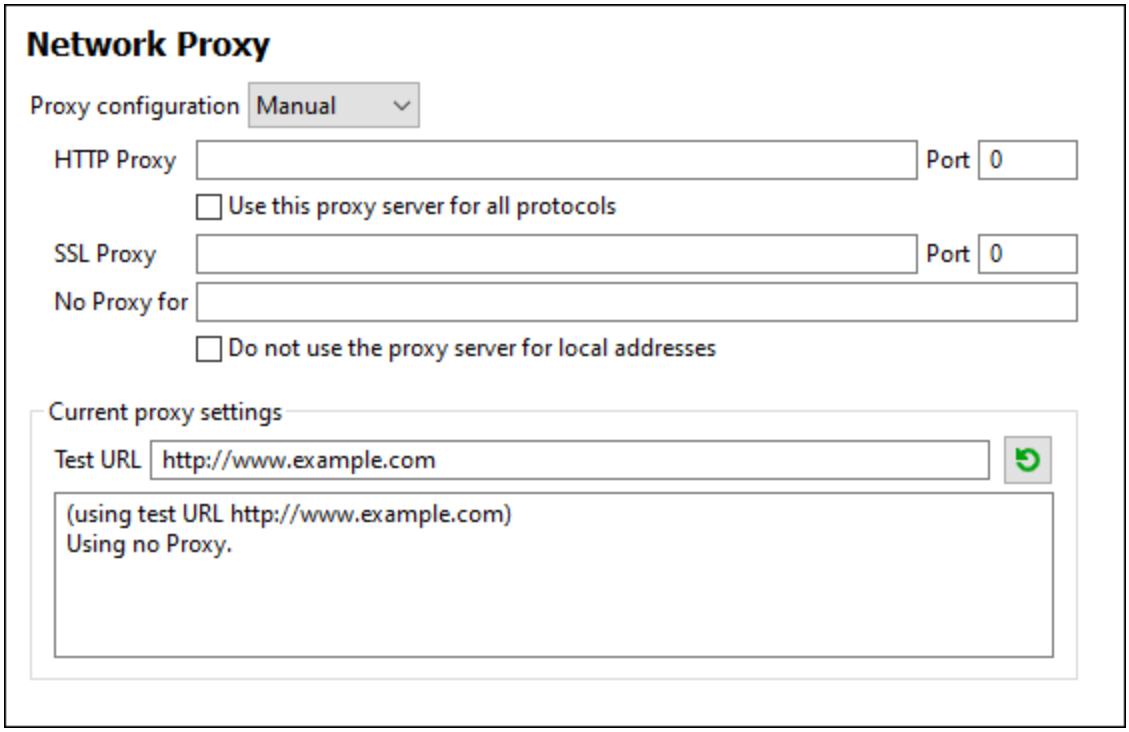
Use system proxy settings
Uses the Internet Explorer (IE) settings configurable via the system proxy settings. It also queries the settings configured with **`netsh.exe winhttp`**.

Automatic proxy configuration
The following options are provided:

- *Auto-detect settings:* Looks up a WPAD script (**`http://wpad.LOCALDOMAIN/wpad.dat`**) via DHCP or DNS, and uses this script for proxy setup.
- *Script URL:* Specify an HTTP URL to a proxy-auto-configuration (**`.pac`**) script that is to be used for proxy setup.
- *Reload:* Resets and reloads the current auto-proxy-configuration. This action requires Windows 8 or newer, and may need up to 30s to take effect.

Manual proxy configuration
Manually specify the fully qualified host name and port for the proxies of the respective protocols. A supported scheme may be included in the host name (for example: **`http://hostname`**). It is not required that the scheme is the same as the respective protocol if the proxy supports the scheme.

© 2019-2025 Altova GmbH



The following options are provided:

- *HTTP Proxy:* Uses the specified host name and port for the HTTP protocol. If *Use this proxy server for all protocols* is selected, then the specified HTTP proxy is used for all protocols.
- *SSL Proxy:* Uses the specified host name and port for the SSL protocol.
- *No Proxy for:* A semi-colon (`;`) separated list of fully qualified host names, domain names, or IP addresses for hosts that should be used without a proxy. IP addresses may not be truncated and IPv6 addresses have to be enclosed by square brackets (for example: `[2606:2800:220:1:248:1893:25c8:1946]`). Domain names must start with a leading dot (for example: `.example.com`).
- *Do not use the proxy server for local addresses:* If checked, adds `<local>` to the *No Proxy for* list. If this option is selected, then the following will not use the proxy: (i) `127.0.0.1`, (ii) `[::1]`, (iii) all host names not containing a dot character (`.`).

Current proxy settings
Provides a verbose log of the proxy detection. It can be refreshed with the **Refresh** button to the right of the *Test URL* field (for example, when changing the test URL, or when the proxy settings have been changed).

- *Test URL:* A test URL can be used to see which proxy is used for that specific URL. No I/O is done with this URL. This field must not be empty if proxy-auto-configuration is used (either through *Use system proxy settings* or *Authomatic proxy configuration*).

15.7 Window

Cascade

This command rearranges all open document windows so that they are all cascaded (i.e. staggered) on top of each other.

Tile horizontally

This command rearranges all open document windows as horizontal tiles, making them all visible at the same time.

Tile vertically

This command rearranges all open document windows as vertical tiles, making them all visible at the same time.

Arrange icons

Arranges haphazardly positioned, iconized diagrams, along the base of the diagram viewing area.

Close

Closes the currently active diagram tab.

Close All

Closes all currently open diagram tabs.

Close All But Active

Closes all diagram tabs except for the currently active one.

Forward

Whenever you change focus from a diagram window to another one, or navigate a hyperlink, UModel "remembers" this as an event. This command takes you "forward" in the history of such events. It is only meaningful and available if you already used the **Back** menu command (see below).

Back

This command takes you back to the window that was previously in focus. This can be useful when you work with many diagram windows simultaneously, or when you navigate with hyperlinks, see Hyperlinking Elements 118.

Window list (1, 2)

This list shows all currently open diagram windows, and lets you quickly switch between them. You can also use the **Ctrl+Tab** or **Ctrl F6** keyboard shortcuts to cycle through the open windows.

© 2019-2025 Altova GmbH

Windows

Displays a dialog box where you can layout or close multiple diagram windows simultaneously, see also Diagram Pane[99].

15.8 Help

This section describes all the menu commands available in the **Help** menu.

Help (F1)

The **Help (F1)** command opens the application's Help documentation (its user manual). By default, the Online Help in HTML format at the Altova website will be opened.

If you do not have Internet access or do not want, for some other reason, to access the Online Help, you can use the locally stored version of the user manual. The local version is a PDF file named `UModel.pdf` that is stored in the application folder (in the Program Files folder).

If you want to change the default format to open (Online Help or local PDF), do this in the Help section of the Options dialog (menu command **Tools | Options**).

Software Activation

License your product
After you download your Altova product software, you can license—or activate—it using either a free evaluation key or a purchased permanent license key.

- ***Free evaluation license.*** When you first start the software after downloading and installing it, the **Software Activation** dialog will pop up. In it is a button to request a free evaluation license. Click it to get your license. When you click this button, your machine-ID will be hashed and sent to Altova via HTTPS. The license information will be sent back to the machine via an HTTP response. If the license is created successfully, a dialog to this effect will appear in your Altova application. On clicking **OK** in this dialog, the software will be activated for a period of 30 days **on this particular machine**.

- ***Permanent license key.*** The **Software Activation** dialog allows you to purchase a permanent license key. Clicking this button takes you to Altova's online shop, where you can purchase a permanent license key for your product. Your license will be sent to you by e-mail in the form of a license file, which contains your license-data.

  There are three types of permanent license: *installed*, *concurrent user*, and *named user*. An installed license unlocks the software on a single computer. If you buy an installed license for *N* computers, then the license allows use of the software on up to *N* computers. A concurrent-user license for *N* concurrent users allows *N* users to run the software concurrently. (The software may be installed on 10N computers.) A named-user license authorizes a specific user to use the software on up to 5 different computers. To activate your software, click **Upload a New License**, and, in the dialog that appears, enter the path to the license file, and click **OK**.

Note: For multi-user licenses, each user will be prompted to enter his or her own name.

> *Your license email and the different ways to license (activate) your Altova product*
> The license email that you receive from Altova will contain your license file as an attachment. The license file has a `.altova_licenses` file extension.
>
> To activate your Altova product, you can do one of the following:

© 2019-2025 Altova GmbH

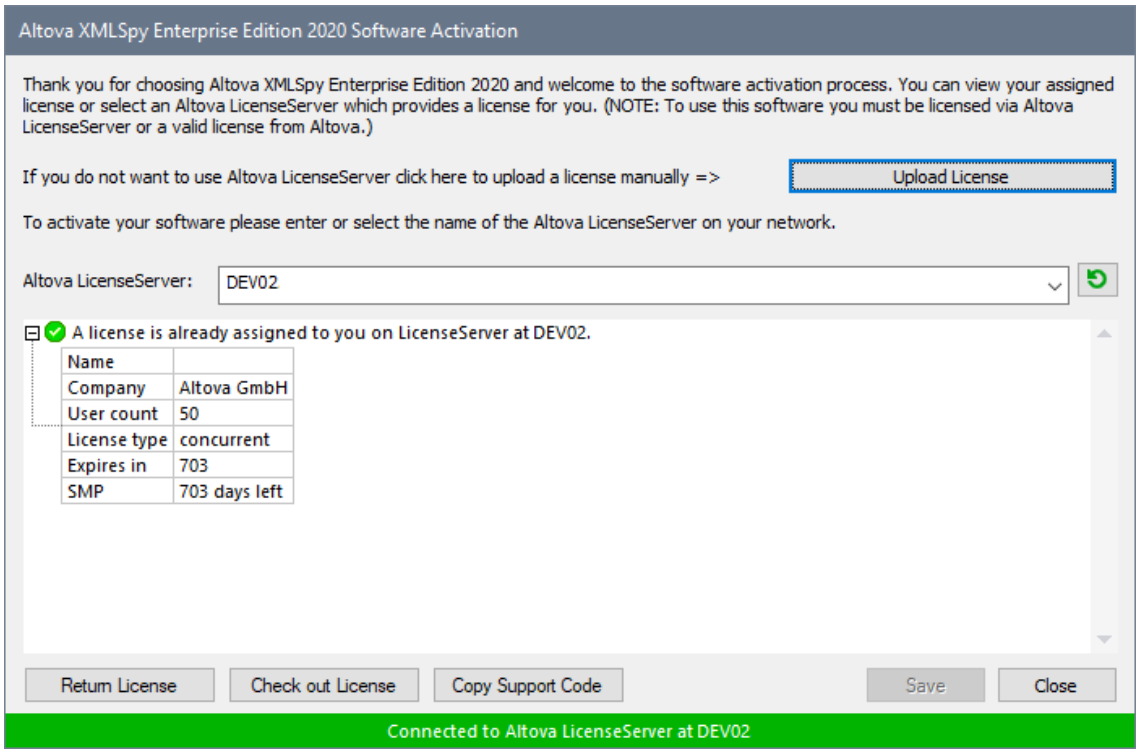
- Save the license file (`.altova_licenses`) to a suitable location, double-click the license file, enter any requested details in the dialog that appears, and finish by clicking **Apply Keys**.
- Save the license file (`.altova_licenses`) to a suitable location. In your Altova product, select the menu command **Help | Software Activation**, and then **Upload a New License**. Browse for or enter the path to the license file, and click **OK**.
- Save the license file (`.altova_licenses`) to any suitable location, and upload it from this location to the license pool of your Altova LicenseServer. You can then either: (i) acquire the license from your Altova product via the product's Software Activation dialog (*see below*) or (ii) assign the license to the product from Altova LicenseServer. *For more information about licensing via LicenseServer, read the rest of this topic.*

You can access the **Software Activation** dialog (*screenshot below*) at any time by clicking the **Help | Software Activation** command.

Activate your software
You can activate the software by registering the license in the Software Activation dialog or by licensing via Altova LicenseServer (*see details below*).

- *Registering the license in the Software Activation dialog*. In the dialog, click **Upload a New License** and browse for the license file. Click **OK** to confirm the path to the license file and to confirm any data you entered (your name in the case of multi-user licenses). Finish by clicking **Save**.

- *Licensing via Altova LicenseServer on your network:* To acquire a license via an Altova LicenseServer on your network, click **Use Altova LicenseServer**, located at the bottom of the **Software Activation** dialog. Select the machine on which the LicenseServer you want to use has been installed. Note that the auto-discovery of License Servers works by means of a broadcast sent out on the LAN. As these broadcasts are limited to a subnet, License Server must be on the same subnet as the client machine for auto-discovery to work. If auto-discovery does not work, then type in the name of the server. The Altova LicenseServer must have a license for your Altova product in its license pool. If a license is available in the LicenseServer pool, this is indicated in the **Software Activation** dialog (*see screenshot below showing the dialog in Altova XMLSpy*). Click **Save** to acquire the license.



After a machine-specific (aka installed) license has been acquired from LicenseServer, it cannot be returned to LicenseServer for a period of seven days. After that time, you can return the machine license to LicenseServer (click **Return License**) so that this license can be acquired from LicenseServer by another client. (A LicenseServer administrator, however, can unassign an acquired license at any time via the administrator's Web UI of LicenseServer.) Note that the returning of licenses applies only to machine-specific licenses, not to concurrent licenses.

Check out license

You can check out a license from the license pool for a period of up to 30 days so that the license is stored on the product machine. This enables you to work offline, which is useful, for example, if you wish to work in an environment where there is no access to your Altova LicenseServer (such as when your Altova product is installed on a laptop and you are traveling). While the license is checked out, LicenseServer displays the license as being in use, and the license cannot be used by any other machine. The license automatically reverts to the checked-in state when the check-out period ends. Alternatively, a checked-out license can be checked in at any time via the **Check in** button of the **Software Activation** dialog.

To check out a license, do the following: (i) In the **Software Activation** dialog, click **Check out License** (*see screenshot above*); (ii) In the **License Check-out** dialog that appears, select the check-out period you want and click **Check out**. The license will be checked out. After checking out a license, two things happen: (i) The **Software Activation** dialog will display the check-out information, including the time when the check-out period ends; (ii) The **Check out License** button in the dialog changes to a **Check In** button. You can check the license in again at any time by clicking **Check In**. Because the license automatically reverts to the checked-in status after the check-out period elapses, make sure that the check-out period you select adequately covers the period during which you will be working offline.

© 2019-2025 Altova GmbH

If the license being checked out is a Installed User license or Concurrent User license, then the license is checked out to the machine and is available to the user who checked out the license. If the license being checked out is a Named User license, then the license is checked out to the Windows account of the named user. License check-out will work for virtual machines, but not for virtual desktop (in a VDI). Note that, when a Named User license is checked out, the data to identify that license check-out is stored in the user's profile. For license check-out to work, the user's profile must be stored on the local machine that will be used for offline work. If the user's profile is stored at a non-local location (such as a file-share), then the checkout will be reported as invalid when the user tries to start the Altova application.

License check-ins must be to the same major version of the Altova product from which the license was checked out. So make sure to check in a license before you upgrade your Altova product to the next major version.

Note: For license check-outs to be possible, the check-out functionality must be enabled on LicenseServer. If this functionality has not been enabled, you will get an error message to this effect when you try to check out. In this event, contact your LicenseServer administrator.

Copy Support Code
Click **Copy Support Code** to copy license details to the clipboard. This is the data that you will need to provide when requesting support via the online support form.

Altova LicenseServer provides IT administrators with a real-time overview of all Altova licenses on a network, together with the details of each license as well as client assignments and client usage of licenses. The advantage of using LicenseServer therefore lies in administrative features it offers for large-volume Altova license management. Altova LicenseServer is available free of cost from the Altova website. For more information about Altova LicenseServer and licensing via Altova LicenseServer, see the Altova LicenseServer documentation.

- Order Form

  When you are ready to order a licensed version of the software product, you can use either the **Purchase a Permanent License Key** button in the **Software Activation** dialog (*see previous section*) or the **Order Form** command to proceed to the secure Altova Online Shop.

- Registration

  Opens the Altova Product Registration page in a tab of your browser. Registering your Altova software will help ensure that you are always kept up to date with the latest product information.

- Check for Updates

  Checks with the Altova server whether a newer version than yours is currently available and displays a message accordingly.

- Support Center

  A link to the Altova Support Center on the Internet. The Support Center provides FAQs, discussion forums where problems are discussed, and access to Altova's technical support staff.

- Download Components and Free Tools

  A link to Altova's Component Download Center on the Internet. From here you can download a variety of companion software to use with Altova products. Such software ranges from XSLT and XSL-FO processors

to Application Server Platforms. The software available at the Component Download Center is typically free of charge.

- UModel on the Internet

  A link to the Altova website on the Internet. You can learn more about UModel, related technologies and products on the Altova website.

- About UModel

  Displays the splash window and version number of your product. If you are using the 64-bit version of UModel, this is indicated with the suffix (x64) after the application name. There is no suffix for the 32-bit version.

© 2019-2025 Altova GmbH

16 UModel Programmer's Reference

UModel is an Automation Server. That is, it is an application that exposes programmable objects to other applications (called Automation Clients). As a result, an Automation Client can directly access the objects and functionality that the Automation Server makes available. This is beneficial to an Automation Client because it can make use of the functionality of UModel. For example, an Automation Client can use the reverse engineering functionality of UModel. Developers can therefore improve their applications by using the ready-made functionality of UModel.

The programmable objects of UModel are made available to Automation Clients via the UModel API, which is a COM API. The object model of the API and a complete description of all available objects are provided in this documentation (see UModel API Reference 814).

The UModel API can be accessed from within the following environments:

- Scripting Editor 707
- IDE Plug-ins 733
- External programs 752

Each of these environments is described briefly below.

Scripting Editor

You can customize your installation of UModel by modifying and adding functionality to it. You can also create Forms for user input and modify the user interface so that it contains new menu commands and toolbar shortcuts. All these features are achieved by writing scripts that interact with objects of the Application API. To aid you in carrying out these tasks efficiently, UModel offers you an in-built Scripting Editor. A complete description of the functionality available in the Scripting Editor and how it is to be used is given in the Scripting Editor 707 section of this documentation. The supported programming languages are **JScript** and **VBScript**.

IDE Plug-ins

UModel enables you to create your own plug-ins, as DLL files, and integrate them into UModel. The UModel graphical user interface provides commands to enable or disable a plug-in. Typical languages used to implement an IDE plug-in are **C#** and **C++**. For more information, see IDE Plug-ins 733 .

External programs

Additionally, you can manipulate UModel with external scripts. For example, you could write a script to open UModel at a given time, then open a UModel project generate UML documentation, and print it out. External scripts would again make use of the API to carry out these tasks, see The UModel API 752.

Using the UModel API from outside UModel requires an instance of UModel to be started first, see Accessing the API 752.

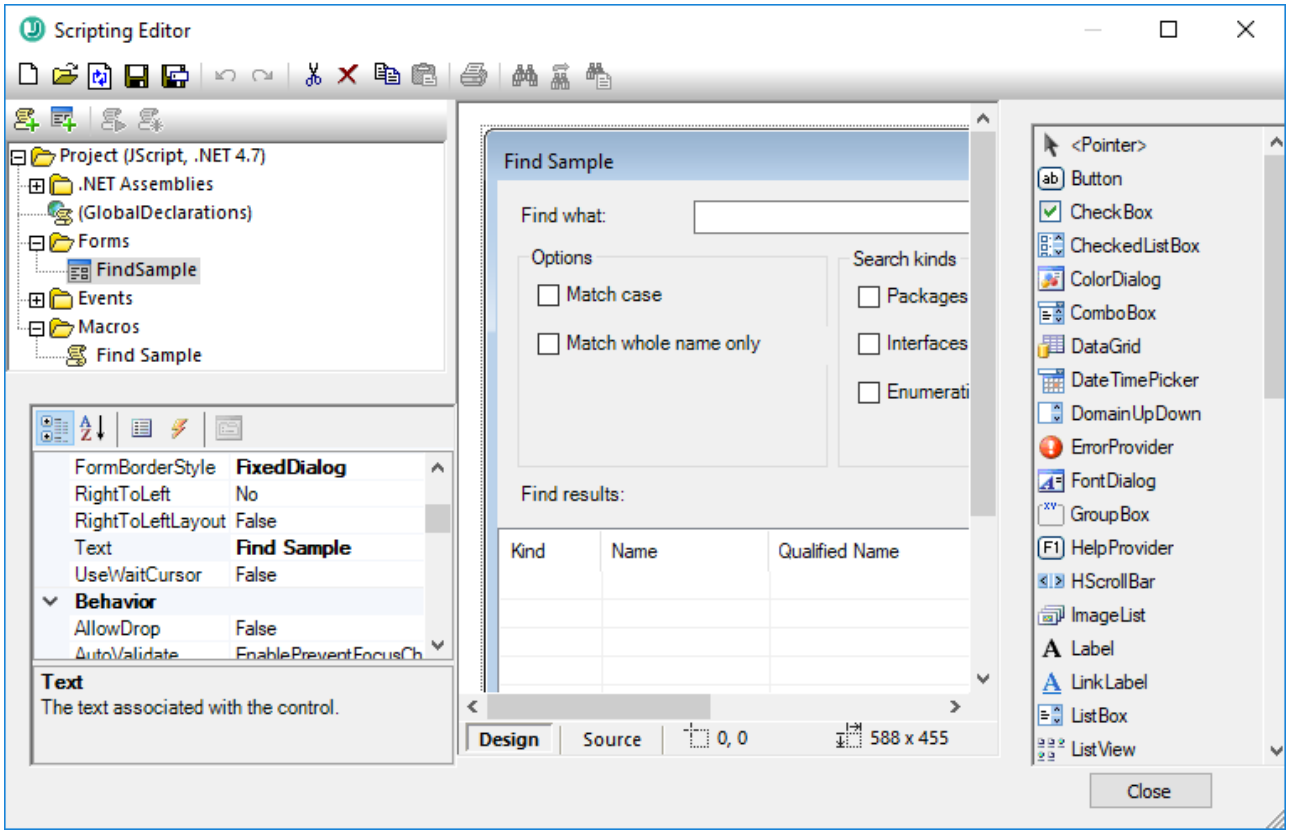
Essentially, UModel will be started via its COM registration. Then the `Application` object associated with the UModel instance is returned. Depending on the COM settings, an object associated with an already running UModel can be returned. Any programming language that supports creation and invocation of COM objects can be used. The most common of these are listed below.

- JScript[799] and VBScript script files have a simple syntax and are designed to access COM objects. You can run such scripts directly from the command line or with a double click from Windows Explorer. They are best used for simple automation tasks.
- C#[771] is a full-fledged programming language that provides support for COM interoperability.
- Java[797]: Altova products come with native Java classes that wrap the Application API and provide a full Java look-and-feel.
- Other programming languages that make useful alternatives are: Visual Basic for Applications, Perl, and Python.

© 2019-2025 Altova GmbH

16.1 Scripting Editor

Scripting Editor is a development environment built into UModel from where you can customize the functionality of UModel with the help of JScript or VBScript scripts. For example, you can add a new menu item to perform a custom project task, or you can have UModel trigger some behavior each time when a document is opened or closed. To make this possible, you create scripting projects—files with .asprj extension (Altova Scripting Project).



Scripting Editor

Scripting projects typically include one or several macros—these are programs that perform miscellaneous custom tasks when invoked. You can run macros either explicitly from a menu item (or a toolbar button, if configured), or you can set up a macro to run automatically whenever UModel starts. The scripting environment also integrates with the UModel COM API. For example, your VBScript or JScript scripts can handle application or document events such as starting or shutting down UModel, opening or closing a project, and so on. Scripting projects can include Windows Forms that you can design visually, in a way similar to Visual Studio. In addition, several built-in commands are available that help you instantiate and use .NET classes from VBScript or JScript code.

Once your scripting project is complete, you can enable it either globally in UModel, or only for specific projects.

Scripting Editor requires .NET Framework 2.0 or later to be installed before UModel is installed.

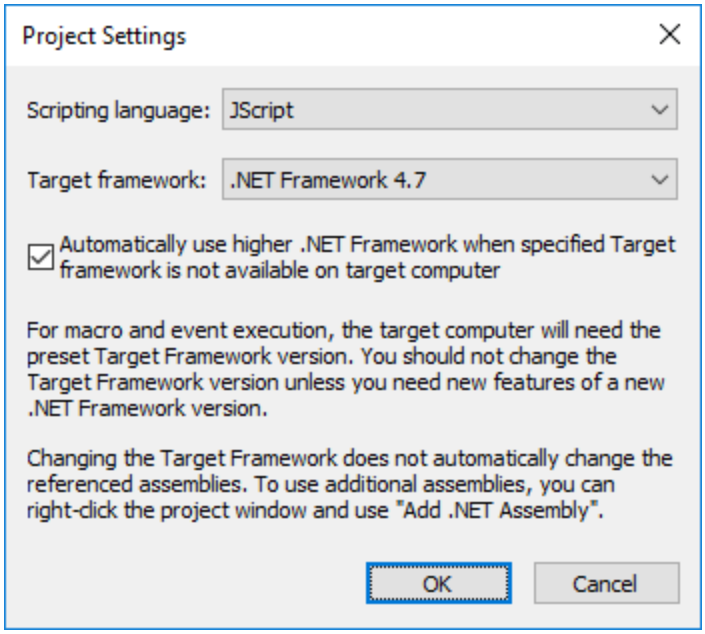
16.1.1 Creating a Scripting Project

All scripts and scripting information created in the Scripting Editor are stored in Altova Scripting Projects (.asprj files). A scripting project may contain macros, application event handlers, and forms (which can have their own event handlers). In addition, you can add global variables and functions to a "Global Declarations" script—this makes such variables and functions accessible across the entire project.

To start a new project, run the menu command **Tools | Scripting Editor**.

The languages supported for use in a scripting project are JScript and VBScript (not to be confused with Visual Basic, which is not supported). These scripting engines are available by default on Windows and have no special requirements to run. You can select a scripting language as follows:

1. Right-click the **Project** item in the upper-left pane, and select **Project settings** from the context menu.
2. Select a language (JScript or VBScript), and click **OK**.



From the Project settings dialog box above, you can also change the target .NET Framework version. This is typically necessary if your scripting project requires features available in a newer .NET Framework version. Note that any clients using your scripting project will need to have the same .NET Framework version installed (or a later compatible version).

By default, a scripting project references several .NET assemblies, like `System`, `System.Data`, `System.Windows.Forms`, and others. If necessary, you can import additional .NET assemblies, including assemblies from .NET Global Assembly Cache (GAC) or custom .dll files. You can import assemblies as follows:

1. Statically, by adding them manually to the project. Right-click **Project** in the top-left pane, and select **Add .NET Assembly** from the context menu.
2. Dynamically, at runtime, by calling the `CLR.LoadAssembly` 723 command from the code.

© 2019-2025 Altova GmbH

You can create multiple scripting projects if necessary. You can save a scripting project to the disk, and then load it back into the Scripting Editor later. To do this, use the standard Windows buttons available in the toolbar: **New**, **Open**, **Save**, **Save As**. Once the scripting project has been tested and is ready for deployment, you can load it into UModel and run any of its macros or event handlers. For more information, see Enabling Scripts and Macros 730.

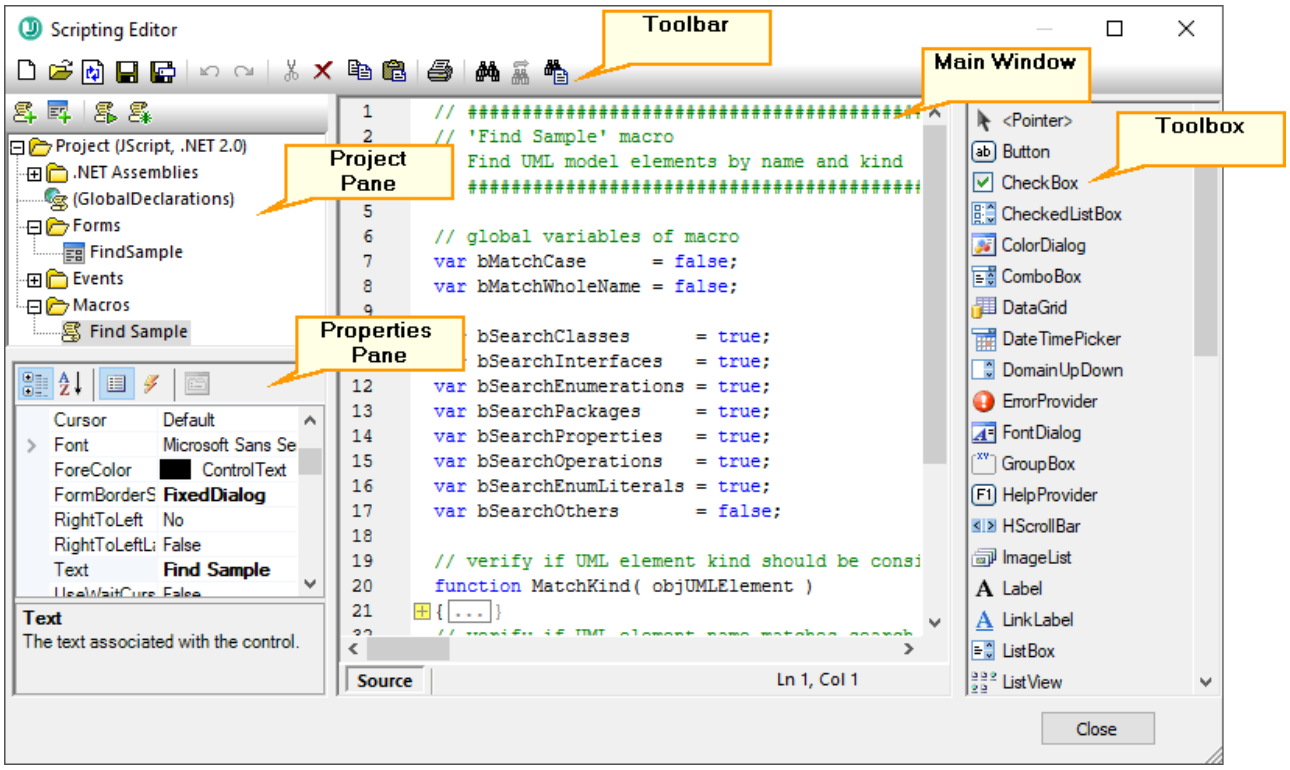
You can also find an example scripting project at the following path: **C:\Users\<user>\Documents\Altova\UModel2025\UModelExamples\Scripting\ScriptSampleFind.asprj**.

The next sections focus on the parts that your scripting project may need: global declarations, macros, forms, and events.

16.1.1.1 Overview of the Environment

The Scripting Editor consists of the following parts:

- Toolbar
- Project pane
- Properties pane
- Main window
- Toolbox



Toolbar

The toolbar includes standard Windows file management commands (**New**, **Open**, **Save**, **Save As**) and editor commands (**Copy**, **Cut**, **Delete**, **Paste**). When editing source code, the **Find** and **Replace** commands are additionally available, as well as the **Print** command.

Project pane

The project pane helps you view and manage the structure of the project. A scripting project consists of several components that can work together and may be created in any order:

- A *"Global Declarations"* script. As the name suggests, this script stores information available globally across the project. You can declare in this script any variables or functions that you need to be available in all forms, event handler scripts, and macros.
- *Forms.* Forms are typically necessary to collect user input, or provide some informative dialog boxes. A form is invoked by a call to it either within a function (in the Global Declarations script) or directly in a macro.
- *Events.* The "Events" folder displays UModel application events provided by the COM API. To write a script that will be executed when an event occurs, double-click any event, and then type the handling code in the editor. The application events should not be confused with form events; the latter are handled at form level, as further detailed below.
- *Macros*. A macro is a script that can be invoked either on demand from a context menu or be executed automatically when UModel starts. Macros do not have parameters or return values. A macro can access all variables and functions declared in the Global Declarations script and it can also display forms.

Right-click any of the components to see the available context menu commands and their shortcuts. Double-click any file (such as a form or a script) to open it in the main window.

The toolbar buttons provide the following quick commands:

	New macro	Adds a new macro to the project, in the **Macros** directory.
	New form	Adds a new form to the project, in the **Forms** directory.
	Run macro	Runs the selected macro.
	Debug macro	Runs the selected macro in debug mode.

Properties pane

The Properties pane is very similar to the one in Visual Studio. It displays the following:

- Form properties, when a form is selected
- Object properties, when an object in a form is selected
- Form events, when a form is selected
- Object events, when an object in a form is selected

To switch between the properties and events of the selected component, click the **Properties** or **Events** buttons, respectively.

© 2019-2025 Altova GmbH

The **Categorized** and **Alphabetical** icons display the properties or events either organized by category or organized in ascending alphabetical order.

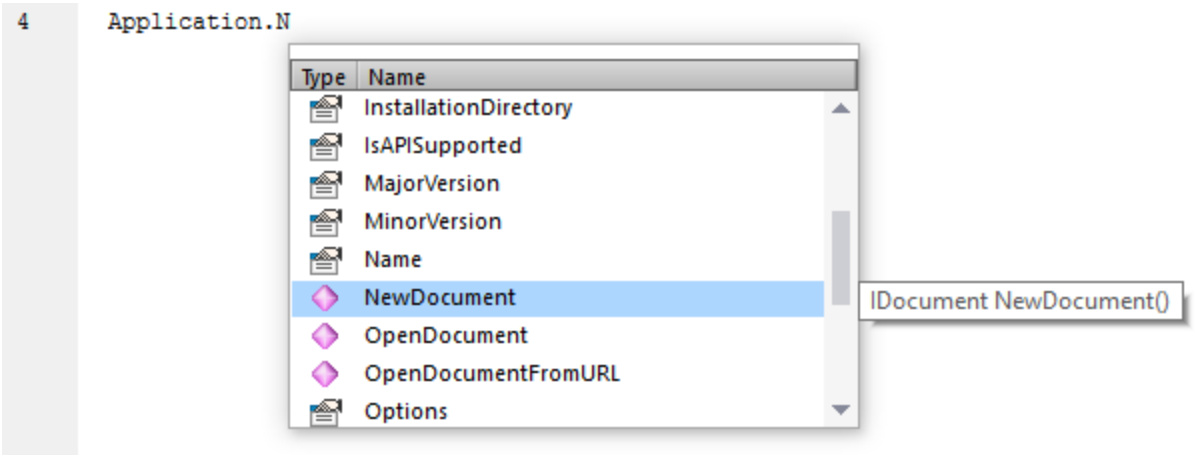
When a property or event is selected, a short description of it is displayed at the bottom of the Properties pane.

Main window

The main window is the working area where you can enter source code or modify the design of the form. When editing forms, you can work in two tabs: the **Design** tab and the **Source** tab. The **Design** tab shows the layout of the form, while the **Source** tab contains the source code such as handler methods for the form events.

The source code editor provides code editing aids such as syntax coloring, source code folding, highlighting of starting and ending braces, zooming, autocompletion suggestions, bookmarks.

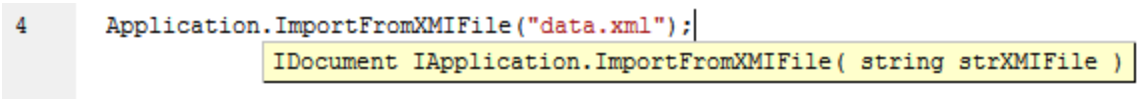
Autocompletion suggestions
JScript and VBScript are untyped languages, so autocompletion is limited to COM API names and UModel built-in commands 720. The full method or property signature is shown next to the autocompletion entry helper.



If names start with `objUMLxxx`, members of the corresponding `IUMLxxx` interface will be shown. For example, the UModel COM API has an interface, `IUMLClass`. If you use names like `objUMLClass`, `objUMLClass123`, or `objUMLClassParent`, the members of the corresponding `IUMLClass` will be displayed.

If names start with `objApplication`, `objDocument`, or `objDiagramWindow`, then members of the corresponding interface will be shown. This also applies to all other interfaces defined in the UModel API.

Placing the mouse over a known method or property displays its signature (and documentation if available), for example:



The auto-completion entry helper is normally shown automatically during editing, but it can also be obtained on demand by pressing **Ctrl+Space**.

Bookmarks

- To set or remove a bookmark, click inside a line, and then press **Ctrl+F2**
- To navigate to the next bookmark, press **F2**
- To navigate to the previous bookmark, press **Shift+F2**
- To delete all bookmarks, press **Ctrl+Shift+F2**

Zooming in/out

- To zoom in or out, hold the **Ctrl** key pressed and then press the "+" or "-" keys or rotate the mouse wheel.

Text view settings

To trigger text settings, right-click inside the editor, and select **Text View Settings** from the context menu.

Font settings

To change the font, right-click inside the editor, and select **Text View Font** from the context menu.

Toolbox

The Toolbox contains all the objects that are available for designing forms, such as buttons, text boxes, combo boxes, and so on.

To add a Toolbox item to a form:

1. Create or open a form and make sure that the **Design** tab is selected.
2. Click the Toolbox object (for example, **Button**), and then click at the location in the form where you wish to insert it. Alternatively, drag the object directly onto the form.

Some objects such as `Timer` are not added to the Form but are created in a tray at the bottom of the main window. You can select the object in the tray and set properties and event handlers for the object from the Properties pane. For an example of handling tray components from the code, see Handling form events 714.

You can also add registered ActiveX controls to the form. To do this, right-click the Toolbox area and select **Add ActiveX Control** from the context menu.

16.1.1.2 Global Declarations

The "Global Declarations" script is present by default in any scripting project; you do not need to create it explicitly. Any variables or functions that you add to this script are considered global across the entire project. Consequently, you can refer to such variables and functions from any of the project's macros and events. The following is an example of a global declarations script that imports the `System.Windows.Forms` namespace into the project. To achieve that, the code below invokes the `CLR.Import` command built into Scripting Editor.

```
// import System.Windows.Forms namespace for all macros, forms and events:
CLR.Import( "System.Windows.Forms" );
```

Note: Every time a macro is executed or an event handler is called, the global declarations are re-initialized.

© 2019-2025 Altova GmbH

16.1.1.3 Macros

Macros are scripts that contain JScript (or VBScript, depending on your project's language) statements, such as variable declarations and functions.

If your projects should use macros, you can add them as follows: right-click inside the Project pane, select **Add Macro** from the context menu, and then enter the macro's code in the main form. The code of a macro could be as simple as an alert, for example:

```
alert("Hello, I'm a macro!");
```

More advanced macros can contain variables and local functions. Macros can also contain code that invokes forms from the project. The listing below illustrates an example of a macro that shows a form. It is assumed that this form has already been created in the "Forms" folder and has the name "SampleForm", see also Forms 713.

```
// display a form
ShowForm( "SampleForm" );
```

In the code listing above, `ShowForm` is a command built into Scripting Editor. For reference to other similar commands that you can use to work with forms and .NET objects, see the Built-in Commands 720.

You can add multiple macros to the same project, and you can designate any macro as "auto-macro". When a macro is designated as "auto-macro", it runs automatically when UModel starts. To designate a macro as auto-macro, right-click it, and select **Set as Auto-Macro** from the context menu.

Only one macro can be run at a time. After a macro (or event) is executed, the script is closed and global variables lose their values.

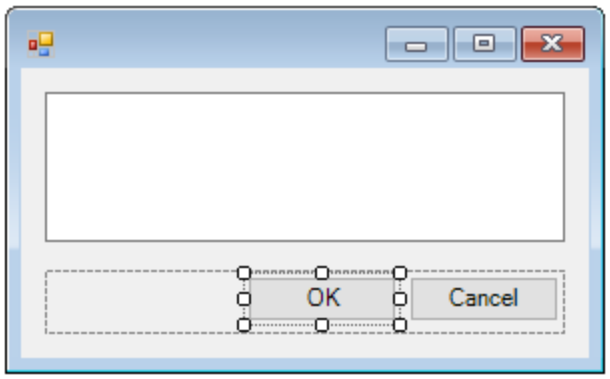
To run a macro directly in Script Editor, click **Run Macro** . To debug a macro using the Visual Studio debugger, click **Debug Macro** . For information about enabling and running macros in UModel, see Enabling Scripts and Macros 730.

16.1.1.4 Forms

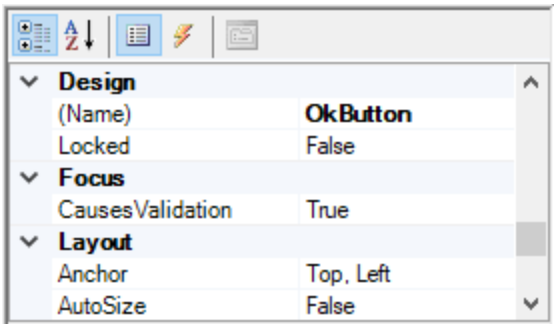
Forms are particularly useful if you need to collect input data from users or display data to users. A form can contain miscellaneous controls to facilitate this, such as buttons, check boxes, combo boxes, and so on.

To add a form, right-click inside the Project pane, and then select **Add Form** from the context menu. To add a control to a form, drag it from the Toolbox available to the right side of Scripting Editor and drop it onto the form.

You can change the position and size of the controls directly on the form, by using the handles that appear when you click any control, for example:



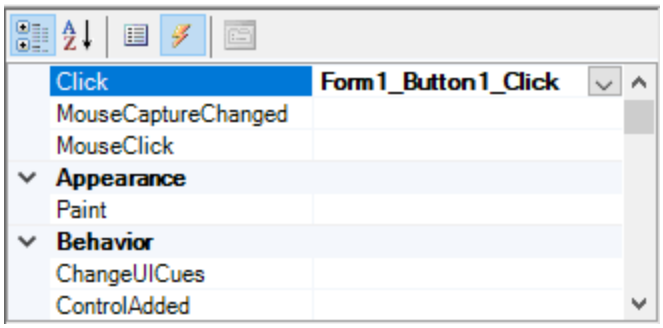
All form controls have properties that you can easily adjust in the Properties pane. To do this, first select the control on the form, and then edit the required properties in the Properties pane.



Handling form events

Each form control also exposes various events to which your scripting project can bind. For example, you might want to invoke some UModel COM API method whenever a button is clicked. To create a function that binds to a form event, do the following:

1. In the Properties pane, click **Events** .
2. In the **Action** column, double-click the event where you need the method (for example, in the image below, the handled event is "Click").



© 2019-2025 Altova GmbH

You can also add handler methods by double-clicking a control on the form. For example, double-clicking a button in the form design generates a handler method for the "Click" event of that button.

Once the body of the handler method is generated, you can type code that handles this event, for example:

```
//Occurs when the component is clicked.
function MyForm_ButtonClick( objSender, e_EventArgs )
{
   alert("A button was clicked");
}
```

To display a work-in-progress form detached from the Scripting Editor, right-click the form in the Project window, and select **Test Form** from the context menu. Note that the **Test Form** command just displays the form; the form's events (such as button clicks) are still disabled. To have the form react to events, call it from a macro, for example:

```
// Instantiate and display a form
ShowForm( "SampleForm" );
```

Accessing form controls

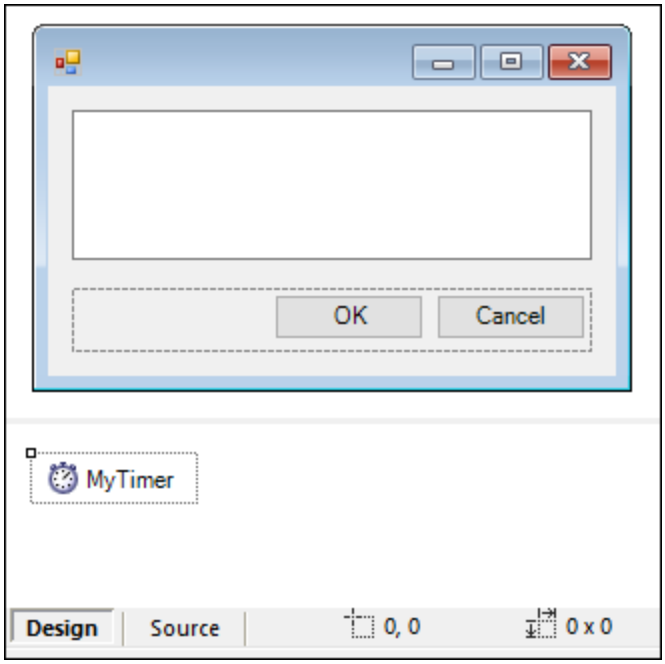
You can access any components on a form from your code by using field access syntax. For example, suppose there is a form designed as follows:

```
// MyForm
//   ButtonPanel
//      OkButton
//      CancelButton
//   TextEditor
//      AxMediaPlayer1
// TrayComponents
//   MyTimer
```

The code below shows how to instantiate the form, access some of its controls using field access syntax, and then display the form:

```
// Instantiate the form
var objForm = CreateForm("MyForm");
// Disable the OK button
objForm.ButtonPanel.OkButton.Enabled = false;
// Change the text of TextEditor
objForm.TextEditor.Text = "Hello";
// Show the form
objForm.ShowDialog();
```

When you add certain controls such as timers to the form, they are not displayed on the form; instead, they are shown as tray components at the base of the form design, for example:



To access controls from the tray, use the `GetTrayComponent` method on the form object, and supply the name of the control as argument. In this example, to get a reference to `MyTimer` and enable it, use the following code:

```
var objTimer = objForm.GetTrayComponent("MyTimer");
objTimer.Enabled = true;
```

For ActiveX Controls, you can access the underlying COM object via the OCX property:

```
var ocx = lastform.AxMediaPlayer1.OCX; // get underlying COM object
ocx.enableContextMenu = true;
ocx.URL = "mms://apasf.apa.at/fm4_live_worldwide";
```

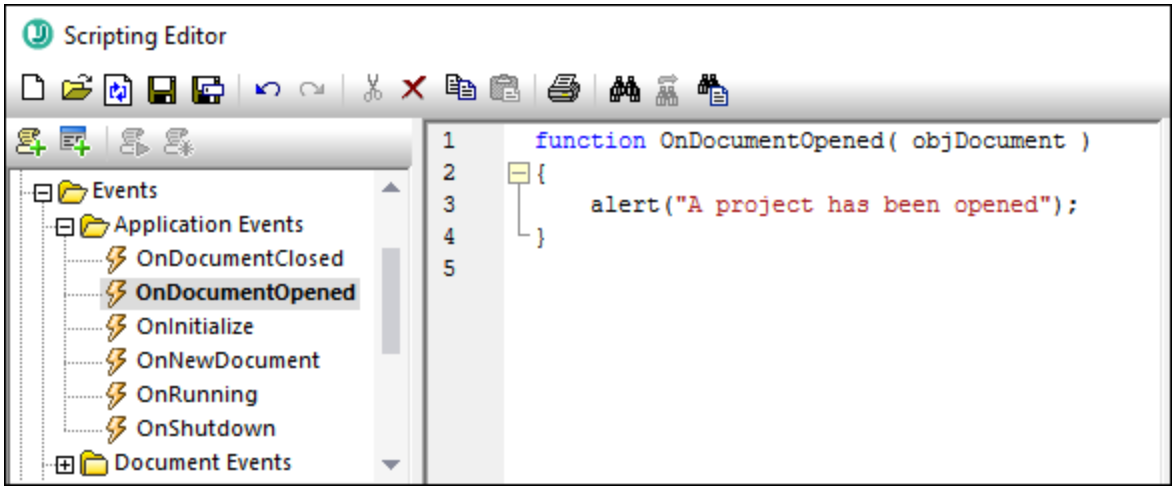
16.1.1.5 Events

Your scripting project may optionally include scripts that handle UModel events such as opening, closing, or saving a document, starting or closing UModel, adding an element to a diagram, and others. These events are provided by the UModel COM API, and you can find them in the "Events" folder of your scripting project. Note that these events are UModel-specific, as opposed to form events. Events are organized into folders as follows:

- Application Events
- Document Events
- Transaction Events
- UMLData Events
- Focused UMLData Events

© 2019-2025 Altova GmbH

To create an event handler script, right-click an event, and select **Open** from the context menu (or double-click the event). The event handler script is displayed in the main window, where you can start editing it. For example, the event handler illustrated below displays an alert each time a project is opened in UModel:



Note the following:

- The `alert` command is applicable to JScript. The VBScript equivalent is `MsgBox`. See also alert [721].
- The name of the event handler function must not be changed; otherwise, the event handler script will not be called.
- In order for events to be processed, the **Process Events** check box must be selected when you enable the scripting project in UModel. For more information, see Enabling Scripts and Macros [730].

You can optionally define local variables and helper functions within event handler scripts, for example:

```
var local;

function OnInitialize( objApplication )
{
   local = "OnInitialize";
   Helper();
}

function Helper()
{
   alert("I'm a helper function for " + local);
}
```

16.1.1.6 JScript Programming Tips

Below are a few JScript programming tips that you may find useful while developing a scripting project in UModel Scripting Editor.

Out parameters

Out parameters from methods of the.NET Framework require special variables in JScript. For example:

```
var dictionary =
CLR.Create("System.Collections.Generic.Dictionary<System.String,System.String>");
dictionary.Add("1", "A");
dictionary.Add("2", "B");

// use JScript method to access out-parameters
var strOut = new Array(1);
if ( dictionary.TryGetValue("1", strOut) ) // TryGetValue will set the out parameter
  alert( strOut[0] ); // use out parameter
```

Integer arguments

.NET Methods that require integer arguments should not be called directly with JScript number objects which are floating point values. For example, instead of:

```
var objCustomColor = CLR.Static("System.Drawing.Color").FromArgb(128,128,128);
```

use:

```
var objCustomColor =
CLR.Static("System.Drawing.Color").FromArgb(Math.floor(128),Math.floor(128),Math.floor(12
8));
```

Iterating .NET collections

To iterate .NET collections, the JScript Enumerator as well as the .NET iterator technologies can be used, for example:

```
// iterate using the JScript iterator
var itr = new Enumerator( coll );
for ( ; !itr.atEnd(); itr.moveNext() )
  alert( itr.item() );

// iterate using the .NET iterator
var itrNET = coll.GetEnumerator();
while( itrNET.MoveNext() )
  alert( itrNET.Current );
```

.NET templates

.NET templates can be instantiated as shown below:

```
var coll = CLR.Create( "System.Collections.Generic.List<System.String>" );
```

or

```
CLR.Import( "System" );
```


© 2019-2025 Altova GmbH

```
CLR.Import( "System.Collections.Generic" );
var dictionary = CLR.Create( "Dictionary<String,Dictionary<String,String>>" );
```

.NET enumeration values

.NET enumeration values are accessed as shown below:

```
var enumValStretch = CLR.Static( "System.Windows.Forms.ImageLayout" ).Stretch;
```

Enumeration literals

The enumeration literals from the UModel API can be accessed as shown below (there is no need to know their numerical value).

```
objExportXMIFileDlg.XMIType = eXMI21ForUML23;
```

16.1.1.7 Example Scripting Project

A demo project that illustrates scripting with UModel is available at the following path: **C:\Users\<user>\Documents\Altova\UModel2025\UModelExamples\Scripting\ScriptSampleFind.asprj**.

This scripting project consists of a macro and a Windows form. The form is where you can search for UML packages, interfaces, operations, and other element kinds in the currently opened UModel project. You can choose the element kinds to be searched for, and you can also make the search case insensitive, and match whole words only.

To load the scripting project into Scripting Editor:

1. On the **Tools** menu, click **Scripting Editor**.
2. Click **Open** and browse for the **ScriptSampleFind.asprj** file from the path above.

Notice that the project contains a macro called **Find Sample** in the "Macros" directory. Also, a search form is available in the "Forms" directory, and it includes various form event handlers.

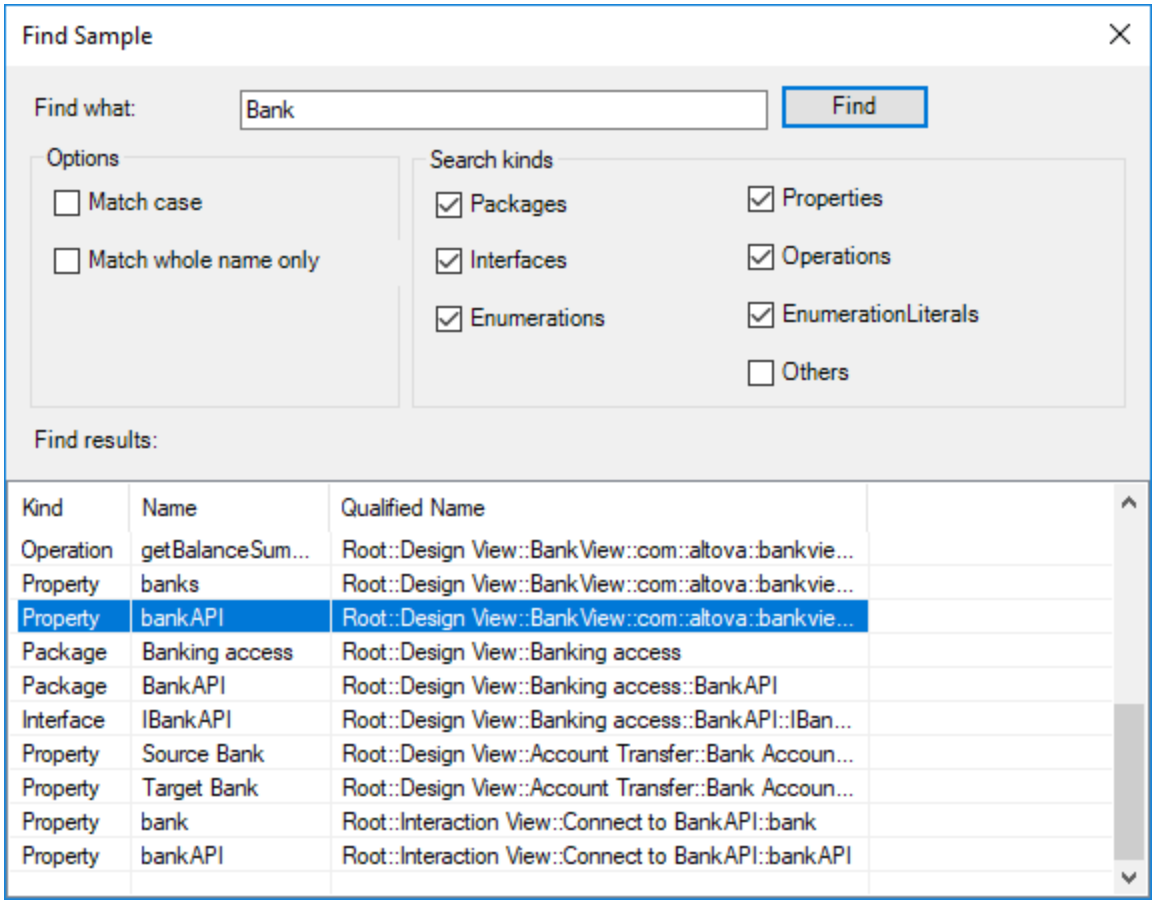
To enable the scripting project as global UModel scripting project:

1. On the **Tools** menu, click **Options**.
2. Click the **Scripting** tab.
3. Under "Global scripting project file", click **Browse** and select the **ScriptSampleFind.asprj** file from the path above.
4. This scripting project does not have auto-macros and application event handlers; therefore, you don't need to select either the **Run auto-macros...** or **Process events** check boxes.
5. Click **Apply**.

At this stage, a new menu item called **Find Sample** becomes available under the **Tools | Macros** menu. This new menu item calls the macro of the scripting project.

To run the macro:

1. Open a UModel project that contains several packages, operations, and so on (in this example, **C:\Users\<user>\Documents\Altova\UModel2025\UModelExamples\Bank_Java.ump**).
2. On the **Tools** menu, click **Macros**, and then click **Find Sample**.
3. Type the search term, and click **Find**.



As shown above, all project elements whose name contains the search term are now listed. You can click on any element in the grid to select it in the Project window.

16.1.2 Built-in Commands

This section provides reference to all the commands you can use in the UModel Scripting Editor.

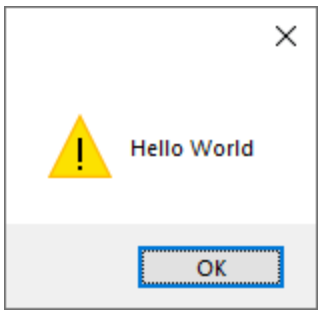
- alert 721
- confirm 721
- CLR.Create 722
- CLR.Import 723
- CLR.LoadAssembly 723
- CLR.ShowImports 724
- CLR.ShowLoadedAssemblies 725

© 2019-2025 Altova GmbH

- CLR.Static 725
- CreateForm 726
- doevents 727
- lastform 727
- prompt 728
- ShowForm 729
- watchdog 729

16.1.2.1 alert

Displays a message box that shows a given message and the "OK" button. To proceed, the user will have to click "OK".



Signature

For JScript, the signature is:

```
alert(strMessage : String) -> void
```

For VBScript, the signature is:

```
MsgBox(strMessage : String) -> void
```

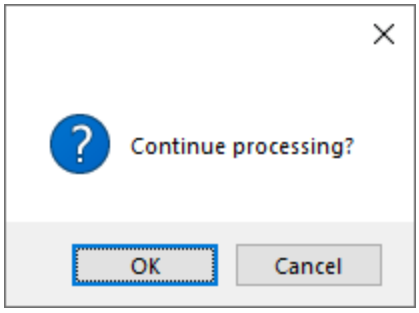
Example

The following JScript code displays a message box with the text "Hello World".

```
alert("Hello World");
```

16.1.2.2 confirm

Opens a dialog box that shows a given message, a confirmation button, and a cancel button. The user will have to click either "OK" or "Cancel" to proceed. Returns a Boolean that represents the user's answer. If the user clicked "OK", the function returns **true**; if the user clicked "Cancel", the function returns **false**.



Signature

```
confirm(strMessage : String) -> result : Boolean
```

Example (JScript)

```
if ( confirm( "Continue processing?" ) == false )
  alert("You have cancelled this action");
```

Example (VBScript)

```
If ( confirm( "Continue processing?" ) = false ) Then
  MsgBox("You have cancelled this action")
End If
```

16.1.2.3 CLR.Create

Creates a new .NET object instance of the type name supplied as argument. If more than one argument is passed, the successive arguments are interpreted as the arguments for the constructor of the .NET object. The return value is a reference to the created .NET object

Signature

```
CLR.Create(strTypeNameCLR : String, constructor arguments ... ) -> object
```

Example

The following JScript code illustrates how to create instances of various .NET classes.

```
// Create an ArrayList
var objArray = CLR.Create("System.Collections.ArrayList");
// Create a ListViewItem
var newItem = CLR.Create( "System.Windows.Forms.ListViewItem", "NewItemText" );
// Create a List<string>
```


© 2019-2025 Altova GmbH

```
var coll = CLR.Create( "System.Collections.Generic.List<System.String>" );
// Import required namespaces and create a Dictionary object
CLR.Import( "System" );
CLR.Import( "System.Collections.Generic" );
var dictionary = CLR.Create( "Dictionary< String, Dictionary< String, String > >" );
```

16.1.2.4 CLR.Import

Imports a namespace. This is the scripting equivalent of C# `using` and VB.Net `imports` keyword. Calling `CLR.Import` makes it possible to leave out the namespace part in subsequent calls like `CLR.Create()` and `CLR.Static()`.

Note: Importing a namespace does not add or load the corresponding assembly to the scripting project. You can add assemblies to the scripting project dynamically (at runtime) in the source code by calling CLR.LoadAssembly 723.

Signature

```
CLR.Import(strNamespaceCLR : String) -> void
```

Example

Instead of having to use fully qualified namespaces like:

```
if ( ShowForm( "FormName" ) == CLR.Static( "System.Windows.Forms.DialogResult" ).OK )
{
    var sName = lastform.textboxFirstName.Text + " " + lastform.textboxLastName.Text;
    CLR.Static( "System.Windows.Forms.MessageBox" ).Show( "Hello " + sName );
}
```

One can import namespaces first and subsequently use the short form:

```
CLR.Import( "System.Windows.Forms" );

if ( ShowForm( "FormName" ) == CLR.Static( "DialogResult" ).OK )
{
    var sName = lastform.textboxFirstName.Text + " " + lastform.textboxLastName.Text;
    CLR.Static( "MessageBox" ).Show( "Hello " + sName );
}
```

16.1.2.5 CLR.LoadAssembly

Loads the .NET assembly with the given long assembly name or file path. Returns Boolean **true** if the assembly could be loaded; **false** otherwise.

Signature

```
CLR.LoadAssembly(strAssemblyNameCLR : String, showLoadErrors : Boolean) -> result :
Boolean
```

Example

The following JScript code attempts to set the clipboard text by loading the required assembly dynamically.

```
// set clipboard text (if possible)
// System.Windows.Clipboard is part of the PresentationCore assembly, so load this
assembly first:
if ( CLR.LoadAssembly( "PresentationCore, Version=3.0.0.0, Culture=neutral,
PublicKeyToken=31bf3856ad364e35", true ) )
{
    var clipboard = CLR.Static( "System.Windows.Clipboard" );
    if ( clipboard != null )
        clipboard.SetText( "HelloClipboard" );
}
```

16.1.2.6 CLR.ShowImports

Opens a message box that shows the currently imported namespaces. The user will have to click "OK" to proceed.

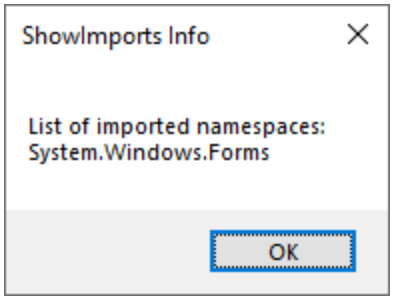
Signature

```
CLR.ShowImports() -> void
```

Example

The following JScript code first imports a namespace, and then displays the list of imported namespaces:

```
CLR.Import( "System.Windows.Forms");
CLR.ShowImports();
```



© 2019-2025 Altova GmbH

16.1.2.7 CLR.ShowLoadedAssemblies

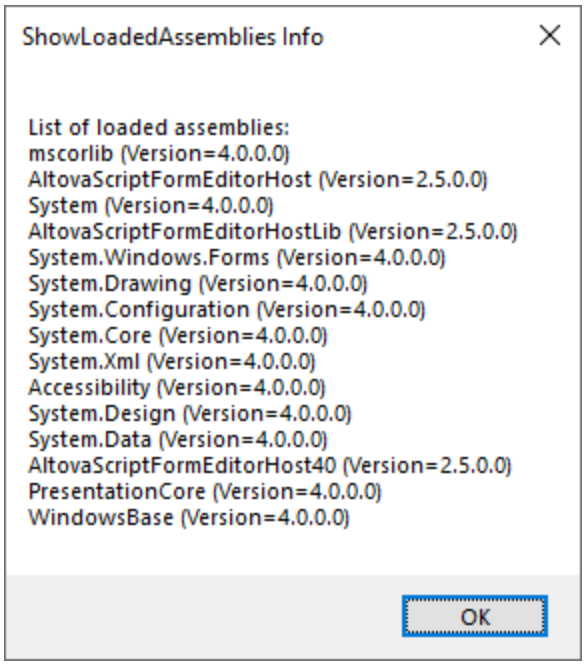
Opens a message box that shows the currently loaded assemblies. The user will have to click "OK" to proceed.

Signature

```
CLR.ShowLoadedAssemblies() -> void
```

Example

```
CLR.ShowLoadedAssemblies();
```



16.1.2.8 CLR.Static

Returns a reference to a static .NET object. You can use this function to get access to .NET types that have no instances and contain only static members.

Signature

```
CLR.Static(strTypeNameCLR : String) -> object
```

Example (JScript)

```
// Get the value of a .NET Enum into a variable
var enumValStretch = CLR.Static( "System.Windows.Forms.ImageLayout" ).Stretch

// Set the value of the Windows clipboard
var clipboard = CLR.Static( "System.Windows.Clipboard" );
clipboard.SetText( "HelloClipboard" );

// Check the buttons pressed by the user on a dialog box
if ( ShowForm( "FormName" ) == CLR.Static( "System.Windows.Forms.DialogResult" ).OK )
    alert( "ok" );
else
    alert( "cancel" );
```

16.1.2.9 CreateForm

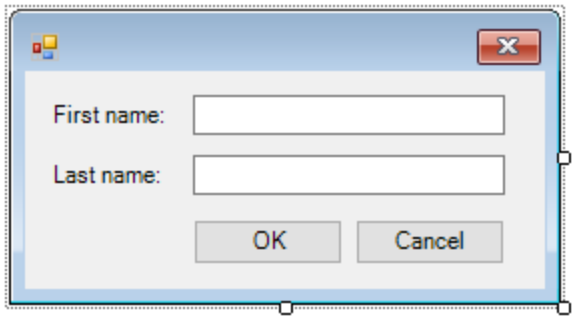
Instantiates the `Form` object identified by the name supplied as argument. The form must exist in the "Forms" folder of the scripting project. Returns the form object (`System.Windows.Forms.Form`) corresponding to the given name, or `null` if no form with such name exists.

Signature

```
CreateForm (strFormName : String) -> System.Windows.Forms.Form | null
```

Example

Let's assume that a form called "FormName" exists in the scripting project.



The following JScript code instantiates the form with some default values and displays it to the user.

```
var myForm = CreateForm( "FormName" );
if ( myForm != null )
{
   myForm.textboxFirstName.Text = "Daniela";
   myForm.textboxLastName.Text = "Heidegger";
```


© 2019-2025 Altova GmbH

```
   var dialogResult = myForm.ShowDialog();
}
```

The `dialogResult` can subsequently be evaluated as follows:

```
if ( dialogResult == CLR.Static( "System.Windows.Forms.DialogResult" ).OK )
    alert( "ok" );
else
    alert( "cancel" );
```

Note: The code above will work only if the **DialogResult** property of the "OK" and "Cancel" buttons is set correctly from the Properties pane (for example, it must be **OK** for the "OK" button).

16.1.2.10 doevents

Processes all Windows messages currently in the message queue.

Signature

```
doevents() -> void
```

Example (JScript)

```
for ( i=0; i < nLongLastingProcess; ++i )
{
    // do long lasting process

    doevents(); // process Windows messages; give UI a chance to update
}
```

16.1.2.11 lastform

This is a global field that returns a reference to the last form object that was created via `CreateForm()` or `ShowForm()`.

Signature

```
lastform -> formObj : System.Windows.Forms.Form
```

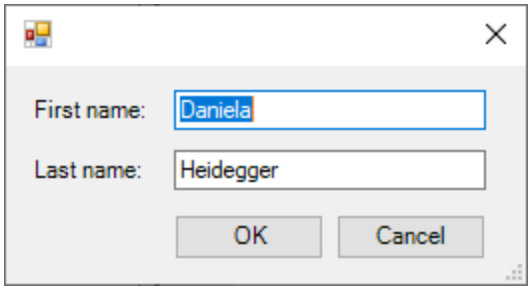
Example

The following JScript code shows the form "FormName" as a dialog box.

```
CreateForm( "FormName" );
if ( lastform != null )
```

```
{
    lastform.textboxFirstName.Text = "Daniela";
    lastform.textboxLastName.Text = "Heidegger";
    var dialogResult = lastform.ShowDialog();
}
```

The values of both textbox controls are initialized with the help of `lastform`.



16.1.2.12 prompt

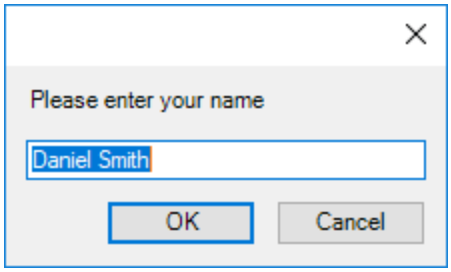
Opens a dialog box that shows a message and a textbox control with a default answer. This can be used to let the user input a simple string value. The return value is a string that contains the textbox value or null if the user selected "Cancel".

Signature

```
prompt(strMessage : String, strDefault : String) -> val : String
```

Example

```
var name = prompt( "Please enter your name", "Daniel Smith" );
if ( name != null )
    alert( "Hello " + name + "!" );
```



© 2019-2025 Altova GmbH

16.1.2.13 ShowForm

Instantiates a new form object from the given form name and immediately shows it as dialog box. The return value is an integer that represents the generated `DialogResult` (`System.Windows.Forms.DialogResult`). For the list of possible values, refer to the documentation of the `DialogResult` Enum (https://docs.microsoft.com/en-us/dotnet/api/system.windows.forms.dialogresult?view=netframework-4.8).

Signature

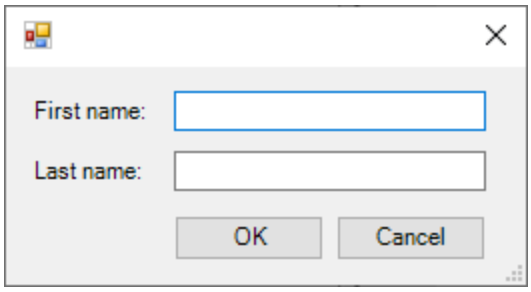
```
ShowForm(strFormName : String) -> result : Integer
```

Example

The following JScript code

```
var dialogResult = ShowForm( "FormName" );
```

Shows the form "FormName" as a dialog box:



The `DialogResult` can subsequently be evaluated, for example:

```
if ( dialogResult == CLR.Static( "System.Windows.Forms.DialogResult" ).OK )
    alert( "ok" );
else
    alert( "cancel" );
```

Note: The code above will work only if the **DialogResult** property of the "OK" and "Cancel" buttons is set correctly from the Properties pane (for example, it must be **OK** for the "OK" button).

16.1.2.14 watchdog

Long running CPU-intensive scripts may ask the user if the script should be terminated. The `watchdog()` method is used to disable or enable this behavior. By default, the watchdog is enabled.

Calling `watchdog(true)` can also be used to reset the watchdog. This can be useful before executing long running CPU-intensive tasks to ensure they have the maximum allowed script processing quota.

Signature

```
watchdog(bEnable : boolean) -> void
```

Example

```
watchdog( false );  // disable watchdog - we know the next statement is CPU intensive but
it will terminate for sure
doCPUIntensiveScript();
watchdog( true );   // re-enable watchdog
```

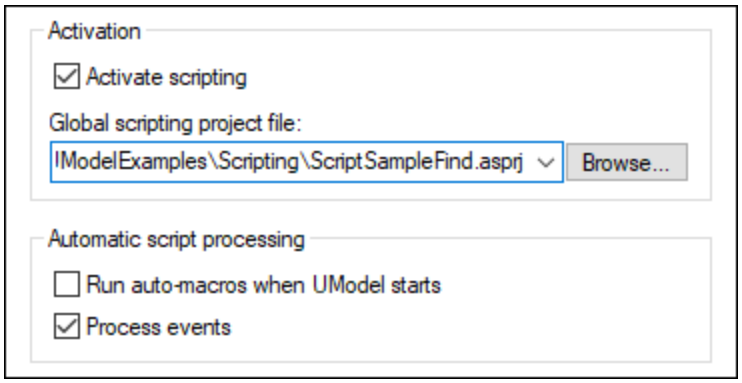
16.1.3 Enabling Scripts and Macros

Once a scripting project is complete and tested, you can use it in the following ways:

1. As the global scripting project for UModel. This means that all the scripts and macros from the scripting project are available to UModel.
2. At UModel project level. This means that a reference to the .asprj file is saved together with the UModel project. When the UModel project is opened, its associated scripts and macros can be called.

To set a scripting project as global:

1. On the **Tools** menu, click **Options**.
2. Click the **Scripting** tab.
3. Select the **Activate scripting** check box and browse for the .asprj file to be used as global scripting project.



You can optionally enable the following additional script processing options:

Run auto-macros when UModel starts	If you select this check box, any macros that were set as "Auto-macro" in the project will be triggerred automatically when UModel starts.
Process events	Select this check box if your scripts bind to any application events. Clear the check box to prevent the

© 2019-2025 Altova GmbH

	scripts from reacting to events.

To enable a scripting project at project level:

1. Open the project.
2. On the **Project** menu, click **Project Settings**.
3. Click the **Scripting** tab.
4. Select the **Activate project scripts** check box and browse for the .asprj file.

The **Run-auto macros...** check box has the same meaning as already described above.

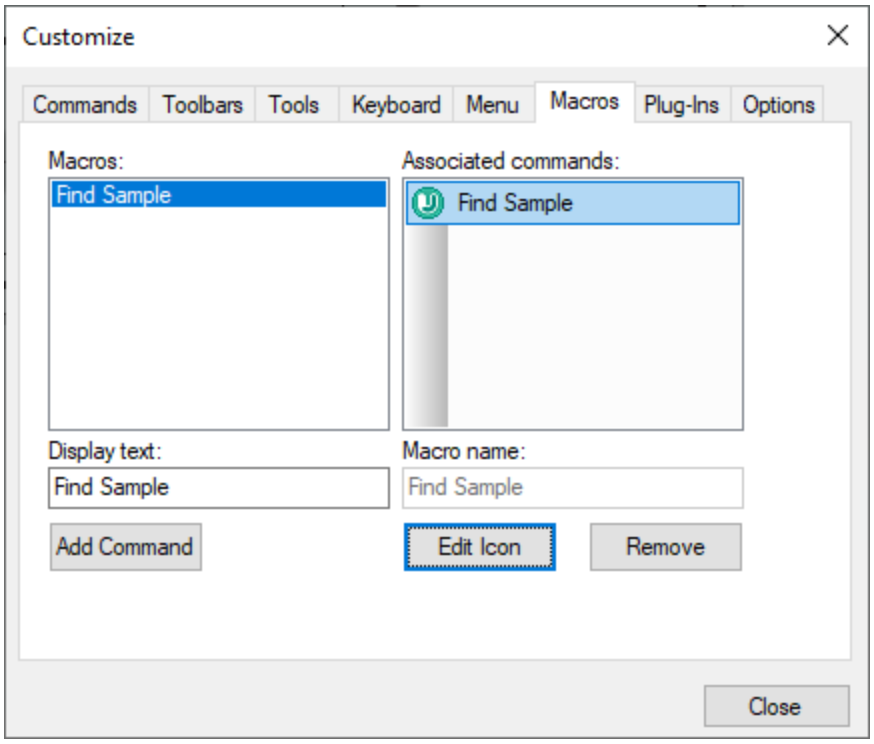
16.1.3.1 Running Macros

When a scripting project is active in UModel, any macros available in that project are displayed in the **Tools | Macros** menu. Therefore, you can run a macro at any time, by triggering the respective menu command, for example **Tools | Macros | <SomeMacro>**.

Macros that were configured as auto-macros will run automatically whenever UModel starts, provided that this behavior is enabled from options, as described in Enabling Scripts and Macros 730.

For convenience, you can create toolbar buttons for macros, as follows:

1. On the **Tools** menu, click **Customize**.
2. Click the **Macros** tab. Any macros that are available at application level (in the *global* scripting project) are listed.
3. Click **Add Command**.



4. Optionally, click **Edit icon** and draw a new icon for the new macro. You can also assign a shortcut to the macro, from the **Keyboard** tab.
5. Drag the macro from the **Associated commands** pane onto the toolbar where you would like it to appear.

To remove a macro from a toolbar:

1. On the **Tools** menu, click **Customize**.
2. Click the **Macros** tab.
3. Drag the macro from the toolbar where it appears back into the **Associated commands** pane.

© 2019-2025 Altova GmbH

16.2 UModel IDE Plug-Ins

One of the ways to interact programmatically with the UModel graphical user interface is creating your own plug-ins for UModel, as DLL libraries. With UModel Integrated Development Environment (IDE) plug-ins, it is possible to achieve the following:

- Customize UModel (for example, add commands through custom menus, icons, or buttons)
- React to events from UModel
- Run your specific code within UModel with access to the complete UModel API
- Integrate your own ActiveX controls into UModel

Plug-ins can be written either as a COM application (in C++) or in a .NET language suitable for COM interoperability, such as C#. Any UModel plug-in must implement the IUModelPlugIn 748 interface. Other prerequisites specific to .NET COM interoperability apply, as further described in this documentation.

A few Visual Studio solutions that illustrate how to access UModel functionality through a custom plug-in are available at the following path: **C:\Users\<username>\Documents\Altova\UModel2025\UModelExamples\IDEPlugIn**.

Limitations

When developing a UModel IDE plug-in, avoid setting the **VisualStyleState** property of the **System.Windows.Forms.Application** object, for example:

```
System.Windows.Forms.Application.VisualStyleState = VisualStyleState.NoneEnabled;
```

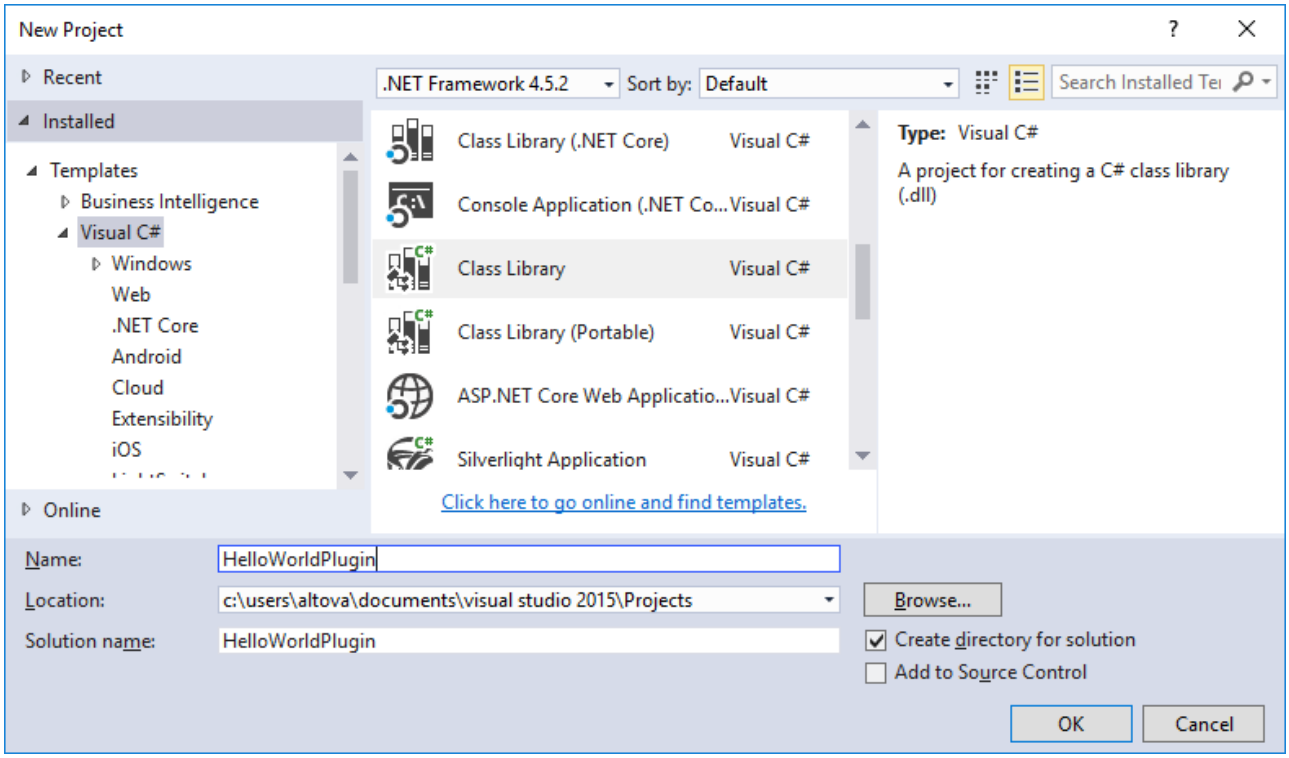
The setting above prevents the COM class from being created and consequently blocks the **File | Open** and **File | Save As** menu commands in UModel when the plug-in is loaded.

16.2.1 How to Create a UModel IDE Plug-In

This section shows how to create a simple UModel IDE plug-in DLL using C# and Visual Studio.

Note: UModel Enterprise or Professional Edition, Visual Studio, and Microsoft .NET Framework must be installed on your computer.

To proceed, run Visual Studio and create a new project of type "Class Library (.dll)".

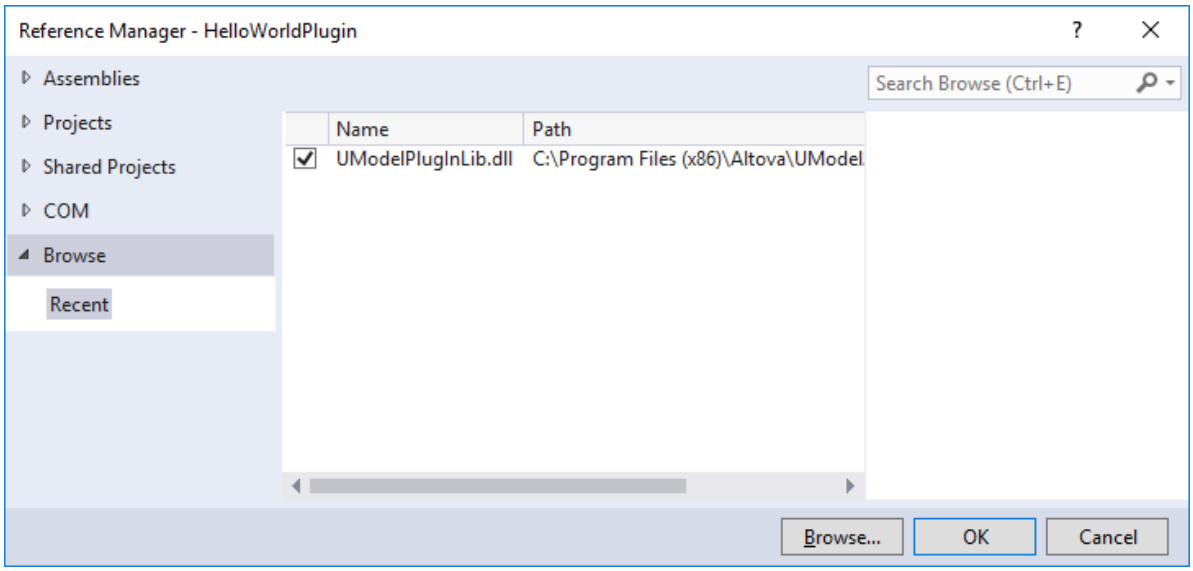


16.2.1.1 Add Reference to UModel Plug-In Library

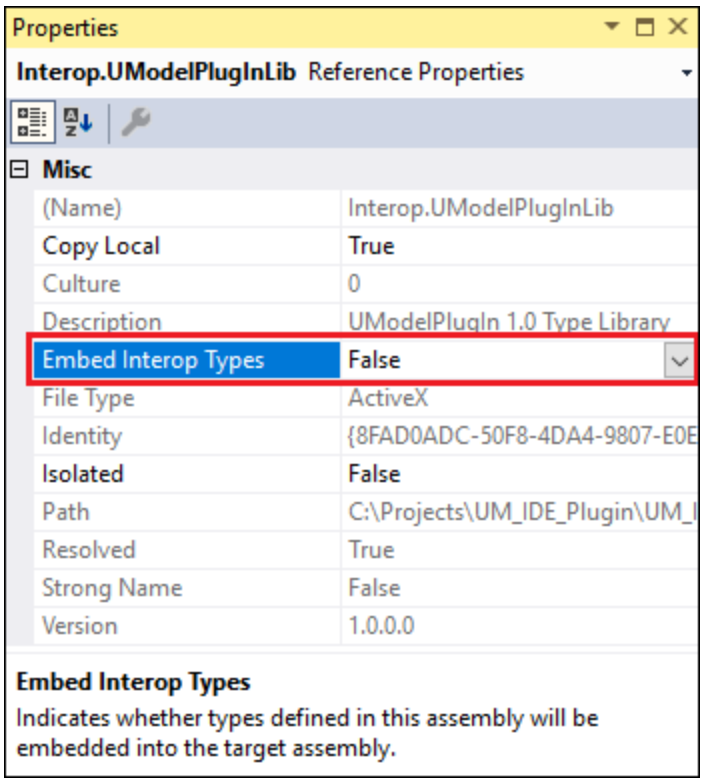
Any DLL library added to UModel as a plug-in must implement the IUModelPlugIn[748] interface. To make this possible, a reference to the **UModelPlugInLib.dll** must first be added in Visual Studio, as follows:

1. Right-click **References** in the Solution Explorer, and select **Add Reference**.
2. On the **Browse** tab, click **Browse** and select **UModelPlugInLib.dll** from the UModel installation directory (for example, **C:\Program Files (x86)\Altova\UModel2025**).

© 2019-2025 Altova GmbH



3. In Solution Explorer, click the referenced library (**UModelPlugInLib**). Find the **Embed Interop Types** property in the Properties window and make sure that this property is set to **False**.



UModelPlugInLib.dll is a .NET assembly and has been created from **IUModelPlugIn.tlb** available in the same folder, using the Microsoft .NET Framework.

If you plan to install your plug-in on a .NET Framework prior to 2.0 (e.g. 1.1), it is necessary that you generate your own **UModelPluginLib.dll** in the respective .NET Framework version.

You can create your own **UModelPlugInLib.dll** assembly using the type library importer of your choice. In .NET, this can be done with the Type Library Importer (**tlbimp.exe**) of the Microsoft .NET Framework SDK:

```
tlbimp.exe IUmodelPlugIn.tlb
```

You can also create the assembly with a strong name key pair and a specific version:

```
tlbimp.exe IUmodelPlugIn.tlb /keyfile:UModelPlugIn.snk /asmversion:1.0.0.0
```

where `UModelPlugIn.snk` is a key file created by the Strong Name Tool (**sn.exe**, also part of the .NET Framework SDK, with a command such as:

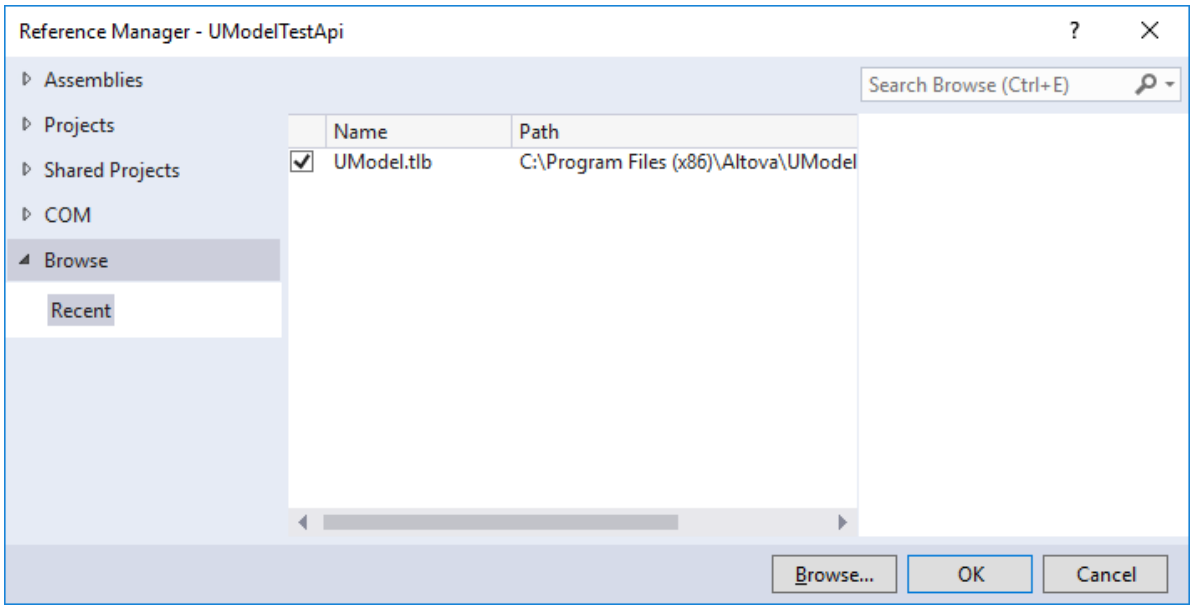
```
sn.exe -k UModelPlugIn.snk
```

For more information about tools included in the .NET Framework, refer to the Microsoft documentation https://docs.microsoft.com/en-us/dotnet/framework/tools/.

16.2.1.2 Add Reference to UModel Type Library

To access the API functionality of UModel from your Visual Studio project, add a reference to the UModel Type Library in Visual Studio, as follows:

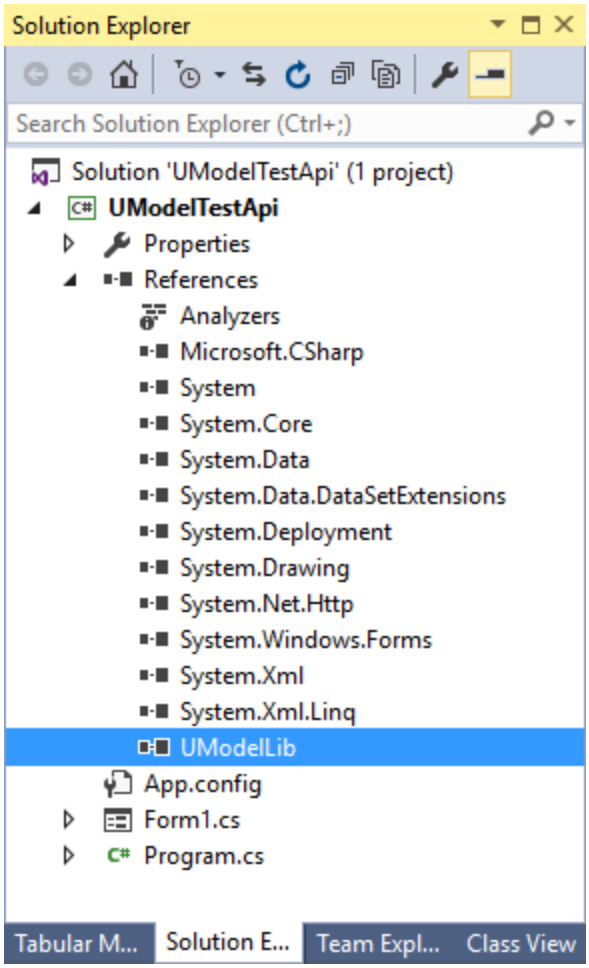
1. Create a new Visual Studio project, or open an existing one.
2. On the **Project** menu, click **Add Reference**.
3. In the COM section, select **UModel Type Library** from the list. If this entry is not available in the COM section, click **Browse** and select the file **UModel.tlb** from the UModel program application folder.



Note: Do not confuse the **UModel Type Library** with the **UModelPlugin Type Library**. The latter can be used to create your own plug-ins and integrate them into UModel, see Add Reference to UModel Plug-In Library 734.

 © 2019-2025 Altova GmbH

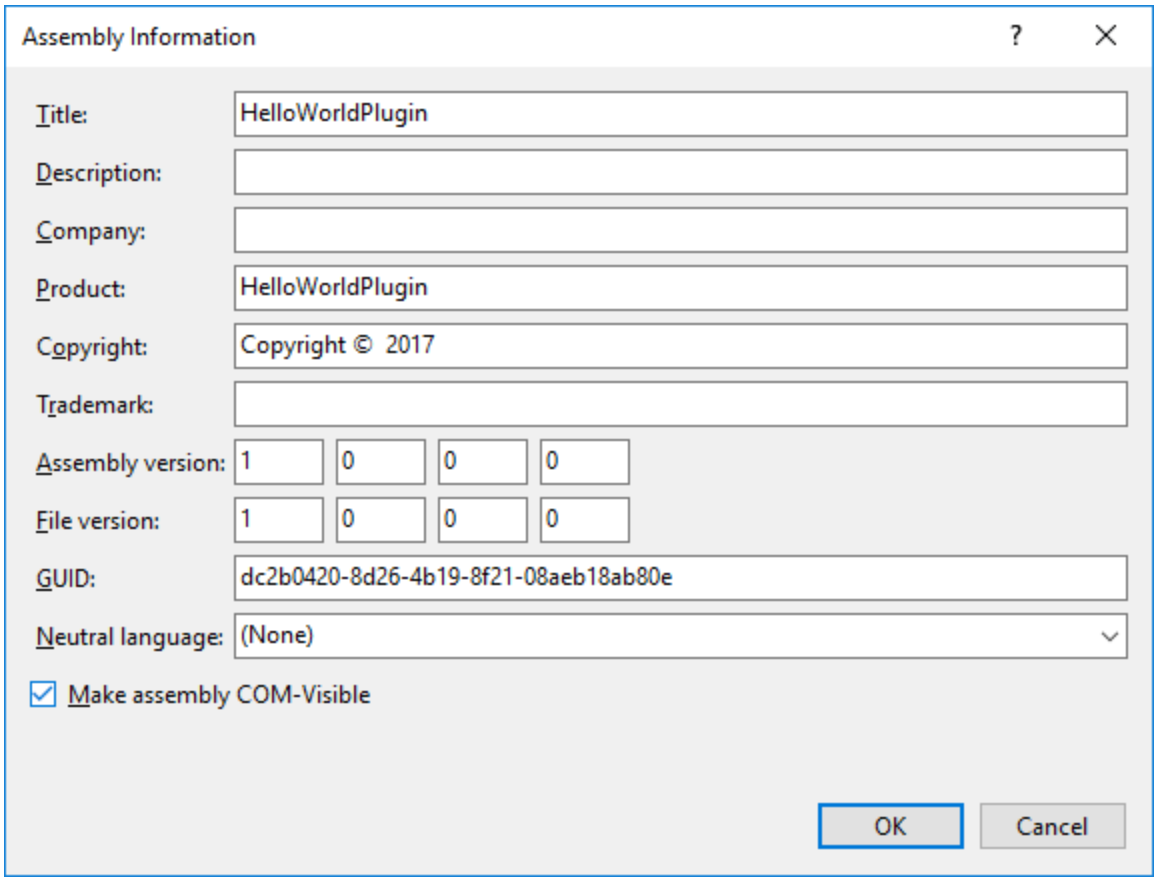
After you follow the steps above, the UModel Type Library should be available in the list of references of your Visual Studio solution, for example:



16.2.1.3 Make the Assembly COM-visible

To make your code accessible to COM, you need to change your compiler settings.

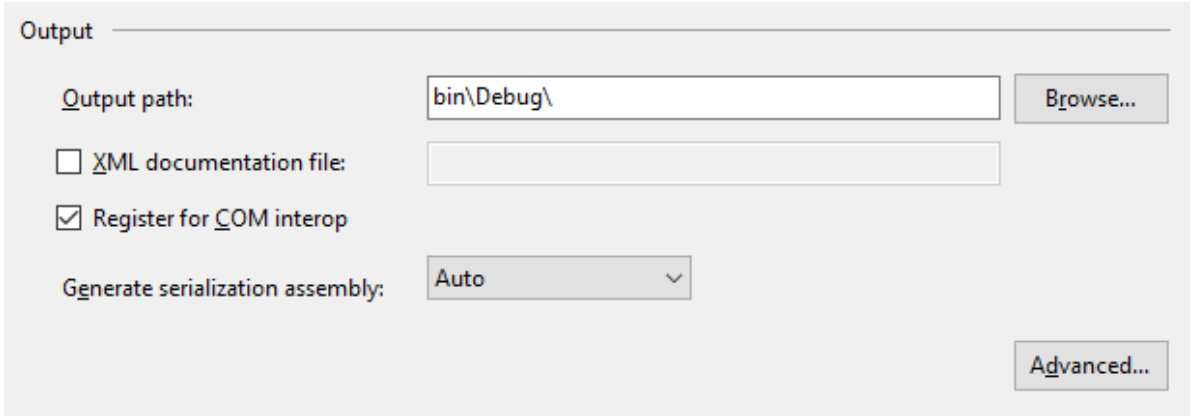
1. Right-click your C# project and select **Properties**.
2. On the **Application** tab, click **Assembly Information...** and select the **Make assembly COM-Visible** check box at the bottom of the dialog box.



16.2.1.4 Expose the COM Wrapper

To expose a COM callable wrapper that can interact with COM objects:

1. Right-click your C# project and select **Properties**.
2. In the **Build** tab, select the **Register for COM interop** check box for all build configurations.

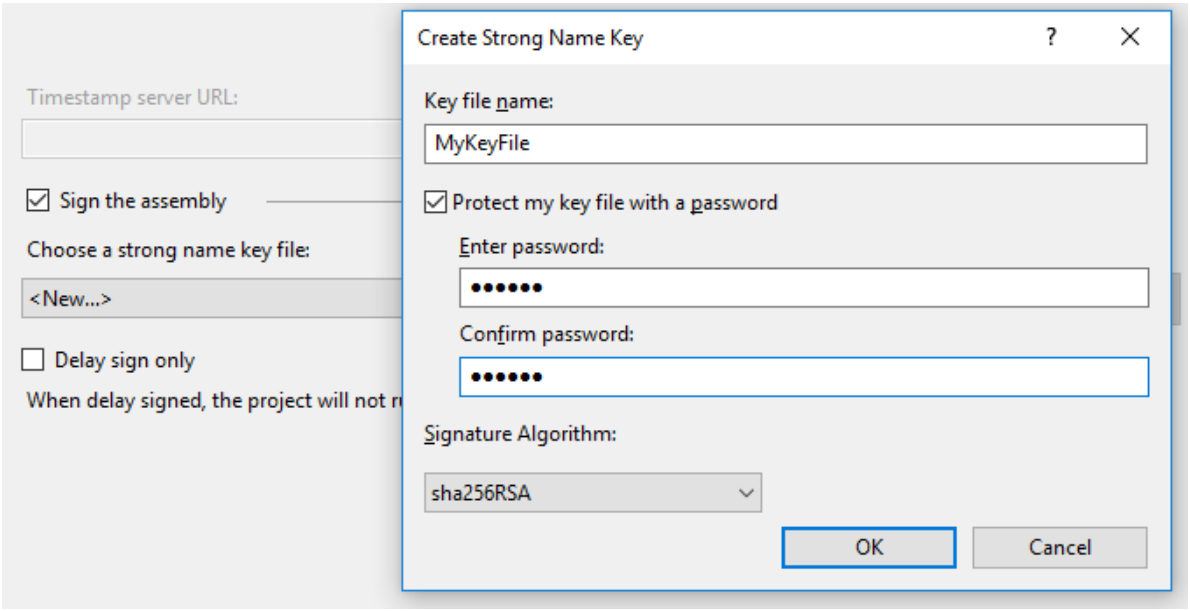


© 2019-2025 Altova GmbH

16.2.1.5 Sign the Plug-In With a Strong Name (Optional)

To sign your assembly with a strong name key pair (e.g. for deployment[742]):

1. Right-click your C# project and select **Properties**.
2. On the **Signing** tab, select the **Sign the assembly check box**.
3. Select either **Browse...** to choose an existing key file or **New...** to create a new one.



16.2.1.6 Implement IUModelPlugIn Interface

UModel IDE plug-ins must implement the IUModelPlugIn[748] interface. The code below shows a simple implementation of this interface. It adds a menu item and a separator (available with UModel) to the **Edit** menu. Clicking the menu item will display a message box with the text "Hello, World!".

Note: Since this sample displays a message box, ensure that your C# project also references `System.Windows.Forms`. To do this, right-click **References** in Solution Explorer, select **Add Reference**, and browse for the `System.Windows.Forms` assembly).

```
using System;
using System.Collections.Generic;
using System.Text;
using UModelPlugInLib;

namespace HelloWorldPlugIn
{
    public class MyHelloWorldUModelPlugIn : IUModelPlugIn
    {
        #region IUModelPlugIn Members

        public string GetDescription()
```

```
        {
            return "HelloWorldPlugIn;HelloWorldPlugIn demonstrates a simple
implementation of an IDE plug-in for UModel";
        }

        public string GetUIModifications()
        {
            return  "<ConfigurationData>"                          +
                        "<Modifications>"                          +
                            // add "Hello World..." to Edit menu
                            "<Modification>"                       +
                                "<Action>Add</Action>"             +
                                "<UIElement type=\"MenuItem\">"    +
                                    "<ID>1</ID>"                   +
                                    "<Name>Hello world...</Name>"  +
                                    "<Info>My hello world</Info>"  +
                                    "<Place>0</Place>"             +
                                    "<MenuID>101</MenuID>"         +
                                    "<Parent>:Edit</Parent>"       +
                                "</UIElement>"                     +
                            "</Modification>"                      +
                        // add Separator to Edit menu
                            "<Modification>"                       +
                                "<Action>Add</Action>"             +
                                "<UIElement type=\"MenuItem\">"    +
                                    "<ID>0</ID>"                   +
                                    "<Place>1</Place>"             +
                                    "<MenuID>101</MenuID>"         +
                                    "<Parent>:Edit</Parent>"       +
                                "</UIElement>"                     +
                            "</Modification>"                      +
                       // finish modification description
                       "</Modifications>"                          +
                   "</ConfigurationData>";
        }

        public void OnInitialize(object pUModel)
        {
            // before processing DDE or batch commands
        }

        public void OnRunning(object pUModel)
        {
            // DDE or batch commands are processed; application is fully initialized
        }

        public void OnShutdown(object pUModel)
        {
            // application will shutdown; release all unused objects
        }

        public UModelUpdateAction OnUpdateCommand(int nID, object pUModel)
        {
            if (nID == 1)
                return UModelUpdateAction.UModelUpdateAction_Enable;

            return UModelUpdateAction.UModelUpdateAction_Disable;
        }
```


© 2019-2025 Altova GmbH

```
        public void OnCommand(int nID, object pUModel)
        {
            System.Windows.Forms.MessageBox.Show("Hello world!");
        }

        #endregion
    }
}
```

16.2.1.7 Build and Run the Plug-In

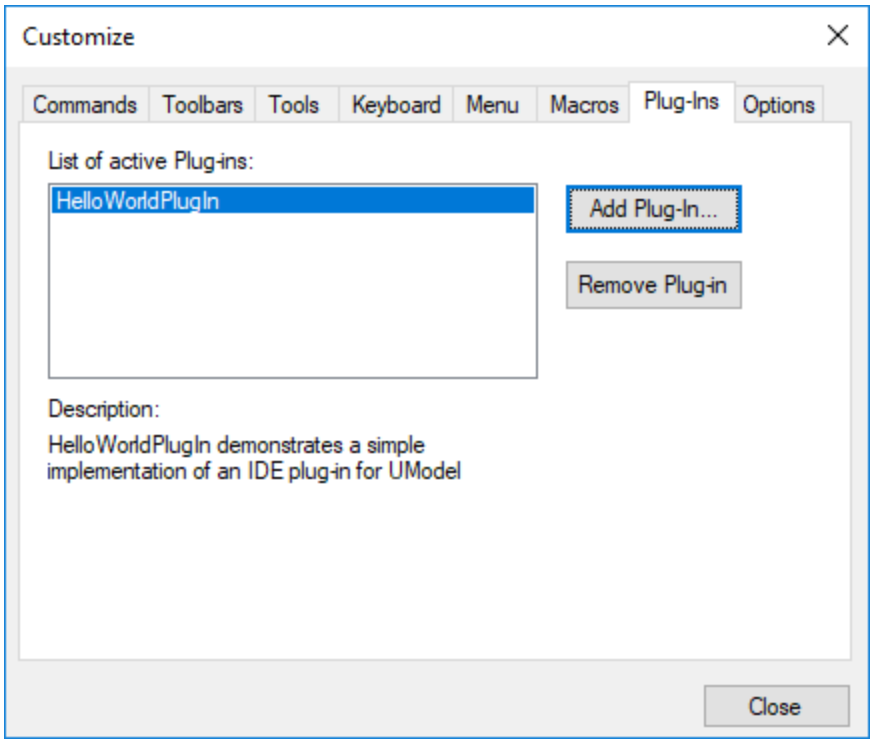
After you have followed the steps above, build the solution with Visual Studio (on the **Build** menu, click **Build Solution**).

Important notes

- Building the plug-in requires access to registry; therefore, make sure to run Visual Studio as administrator.
- If you have a 64-bit operating system and are using a 32-bit installation of UModel, add the x86 platform in the solution's Configuration Manager and build the sample using this configuration. To access Configuration Manager, run the menu command **Build | Configuration Manager**.
- In Solution Explorer, click the referenced library (**UModelPlugInLib**). Find the **Embed Interop Types** property in the Properties window and make sure that this property is set to **False**.

After building your C# project, you can add the plug-in to UModel and test it as follows:

1. Start UModel (or restart it if applicable; this ensures that the plug-in information is read correctly from the registry).
2. On the **Tools** menu, click **Customize**.
3. On the **Plug-Ins** tab, click **Add Plug-In...**, and select the plug-in .dll file (in this example, **HelloWorldPlugIn.dll**):



Note: If you get an error with text similar to "Could not find an implementation of the UModel plug-in interface in type library", make sure that the **Embed Interop Types** property is set to **False** for **UModelPlugInLib** library, as described in Add Reference to UModel Plug-In Library 734.

The **Edit** menu of UModel now contains a new menu command called **Hello world**. Run this command to display a dialog box with the "Hello, World!" message.

16.2.2 Deployment of UModel IDE Plug-Ins

On a development PC, the COM registration takes place when you build the plug-in with Visual Studio; no manual registration is required under normal circumstances. If you intend to deploy a UModel IDE plug-in to a target client system, the target PC must have the following prerequisites:

- UModel Professional or Enterprise edition
- If the plug-in is written in .NET, the corresponding Microsoft .NET Framework.

On a deployment PC, the plug-in can be registered either manually or by the setup. For an example of a Visual Studio setup project, see the "Set Styles" Sample 776.

To register a UModel IDE plug-in manually:

1. On the **Tools** menu of UModel, click **Customize**.
2. Click the **Plug-Ins** tab.
3. Click **Add Plug-In** and browse for the .dll file of the plug-in.

© 2019-2025 Altova GmbH

You can check whether a UModel plug-in is registered by running **regedit.exe** at the command line. UModel maintains the following registry key for all registered plug-ins:

```
HKEY_CURRENT_USER\Software\Altova\UModel\PlugIns
```

All values of this key are treated as references to registered plug-ins and must conform to the following format:

Value name:	`ProgID` of the plug-in
Value type:	must be `REG_SZ`
Value data:	`CLSID` of the component

Every time UModel starts, the values of the "PlugIns" key are scanned, and the registered plug-ins are loaded. If you experience problems, check if the CLSID of your plug-in is correctly registered in the "PlugIns" key. If this is not the case, the name of your plug-in DLL was probably not sufficiently unique. Use a different name in this case.

Note: When deploying your UModel IDE plug-in on .NET framework versions prior to 2.0, the plug-in .dll file must either be installed in the same directory as UModel.exe or signed with a strong name key and registered into the global assembly cache (GAC).

Should you need to perform various assembly-related tasks manually, be aware of the following tools included in the .NET Framework SDK:

- Assembly Registration Tool (**regasm.exe**). Use this to perform manual registration or de-registration of COM assemblies. For example, to maually register the **UModelPlugLib.dll**, use:

  ```
  regasm.exe UModelPlugInLib.dll /codebase
  ```

- Strong Name Tool (**sn.exe**). This can be optionally used to sign your assembly with a strong key, for example:

  ```
  sn.exe -k MyKeyFile.snk
  ```

  The key can also be generated from Visual Studio, see Sign the Plug-In With a Strong Name (Optional) 739.

- Global Assembly Cache Tool (**gacutil.exe**). Use this to add or remove an assembly from the Global Assembly Cache (GAC). For example, to add **MyPlugin.dll** to GAC, use:

  ```
  gacutil.exe /i MyPlugin.dll
  ```

For more information about tools included in the .NET Framework, refer to the Microsoft documentation https://docs.microsoft.com/en-us/dotnet/framework/tools/.

16.2.3 Configuration XML

The plug-in allows you to change the user interface (UI) of UModel. This is done by describing each separate modification using an XML data stream. The XML configuration is passed to UModel using the `GetUIModifications` method of the IUModelPlugIn Interface[748].

The XML file containing the UI modifications for the plug-in must have the following structure:

```
<ConfigurationData>
   <ImageFile>path To image file</ImageFile>
   <Modifications>
      <Modification>
         ...
      </Modification>
      ...
   </Modifications>
</ConfigurationData>
```

You can define icons, or toolbar buttons for the new menu items which are added to the UI of UModel by the plug-in. The path to the file containing the images is set using the **ImageFile** element. Each image must be 16 x 16 pixels. The image references must be arranged from left to right in a single (<ImageFile>...) line. The rightmost image index value is zero.

The **Modifications** element can have any number of **Modification** child elements. Each **Modification** element defines a specific change to the standard UI of UModel. It is also possible to remove UI elements from UModel.

Structure of Modification elements

All Modification elements consist of the following two child elements:

```
<Modification>
   <Action>Type of action</Action>
   <UIElement Type="type of UI element">
   </UIElement>
</Modification>
```

Valid values for the **Action** element are:

- Add - used to add the following UI element to UModel.
- Hide - used to hide the following UI element in UModel.
- Remove - used to remove the UI element from the "Commands" list box, in the customize dialog

You can combine values of the **Action** element e.g. "Hide Remove".

The **UIElement** element describes any new, or existing UI element for UModel. Possible elements are currently: new toolbars, buttons, menus, or menu items. The **Type** attribute defines which UI element is described by the XML element.

© 2019-2025 Altova GmbH

Common UIElement children

The `ID` and `Name` elements are valid for all different types of XML UIElement fragments. It is, however, possible to ignore one of the values for a specific type of UIElement, e.g. `Name` is ignored for a separator.

```
<ID></ID>
<Name></Name>
```

If `UIElement` describes an existing element of the UI, the value of the `ID` element is predefined by UModel. Normally these `ID` values are not known to the public. If the XML fragment describes a new part of the UI, then the `ID` is arbitrary and the value should be less than 1000.

The `Name` element sets the textual value. Existing UI elements can be identified just by name, e.g. menus and menu items with associated sub menus. For new UI elements, the `Name` element sets the caption, e.g. the title of a toolbar, or text for a menu item.

Toolbars and Menus

To define a toolbar, it's necessary to specify the `ID` and/or the name of the toolbar. An existing toolbar can be specified using only the name, or by the ID if it is known. To create a **new** toolbar, both values must be set. The `Type` attribute must be equal to "ToolBar".

```
<UIElement Type="ToolBar">
   <ID>1</ID>
   <Name>Styles</Name>
</UIElement>
```

To specify an UModel menu, you need two parameters:

- The ID of the menu bar which contains the menu. UModel's main menu bar ID is 101.
- The menu name. Menus do not have an associated ID value. The following example defines the "Edit" menu of the menu bar:

```
<UIElement Type="Menu">
   <ID>101</ID>
   <Name>Edit</Name>
</UIElement>
```

An additional element is used if you want to create a new menu. The `Place` element defines the position of the new menu in the menu bar:

```
<UIElement Type="Menu">
   <ID>101</ID>
   <Name>PlugIn Menu</Name>
   <Place>12</Place>
</UIElement>
```

A value of -1 for the **Place** element sets the new button or menu item at the end of the menu or toolbar.

Commands

If you add a new command through a toolbar button or a menu item, the **UIElement** fragment can contain any of these sub elements:

```
<Info></Info>
<ImageID></ImageID>
```

The **Info** element contains a short description string which is displayed in the status bar, when the mouse pointer is over the associated command (button or menu item). **ImageID** defines the index of the icon in the external image file. Please note that all icons are stored in one image file.

To define a toolbar button create an **UIElement** with this structure:

```
<UIElement Type="ToolBarItem">
   <!--don't reuse local IDs even the commands do the same-->
   <ID>6</ID>
   <Name>Fill red</Name>
   <!--Set Place To -1 If this is the first button to be inserted-->
   <Place>-1</Place>
   <ImageID>0</ImageID>
   <ToolBarID>1</ToolBarID>
   <!--instead of the toolbar ID the toolbar name could be used-->
   <ToolBarName>Styles</ToolBarName>
</UIElement>
```

Additional elements to declare a toolbar button are **Place**, **ToolBarID** and **ToolBarName**. **ToolBarID** and **ToolBarName** are used to identify the toolbar which contains the new or existing button. The textual value of **ToolBarName** is case sensitive. The (UIElement) **type** attribute must equal "ToolBarItem".

To define a menu item, the elements **MenuID**, **Place** and **Parent** are available in addition to the standard elements used to declare a command. **MenuID** must be 101. See "Toolbars and Menus" for more information on these values.

The **Parent** element is used to identify the **menu** where the new menu entry should be inserted. As sub menu items have no unique Windows ID, we need some other way to identify the parent of the menu item.

The value of the **Parent** element is a path to the menu item.

The text value of the Parent element, must equal the **parent menu name** of the submenu, where the submenu name is separated by a colon. If the menu has no parent, because it is not a submenu, add a colon to the beginning of the name. The **type** attribute must be set to "MenuItem".

Example for an **UIElement** defining a menu item:

```
<UIElement Type="MenuItem">
   <!--the following element is a Local command ID-->
```

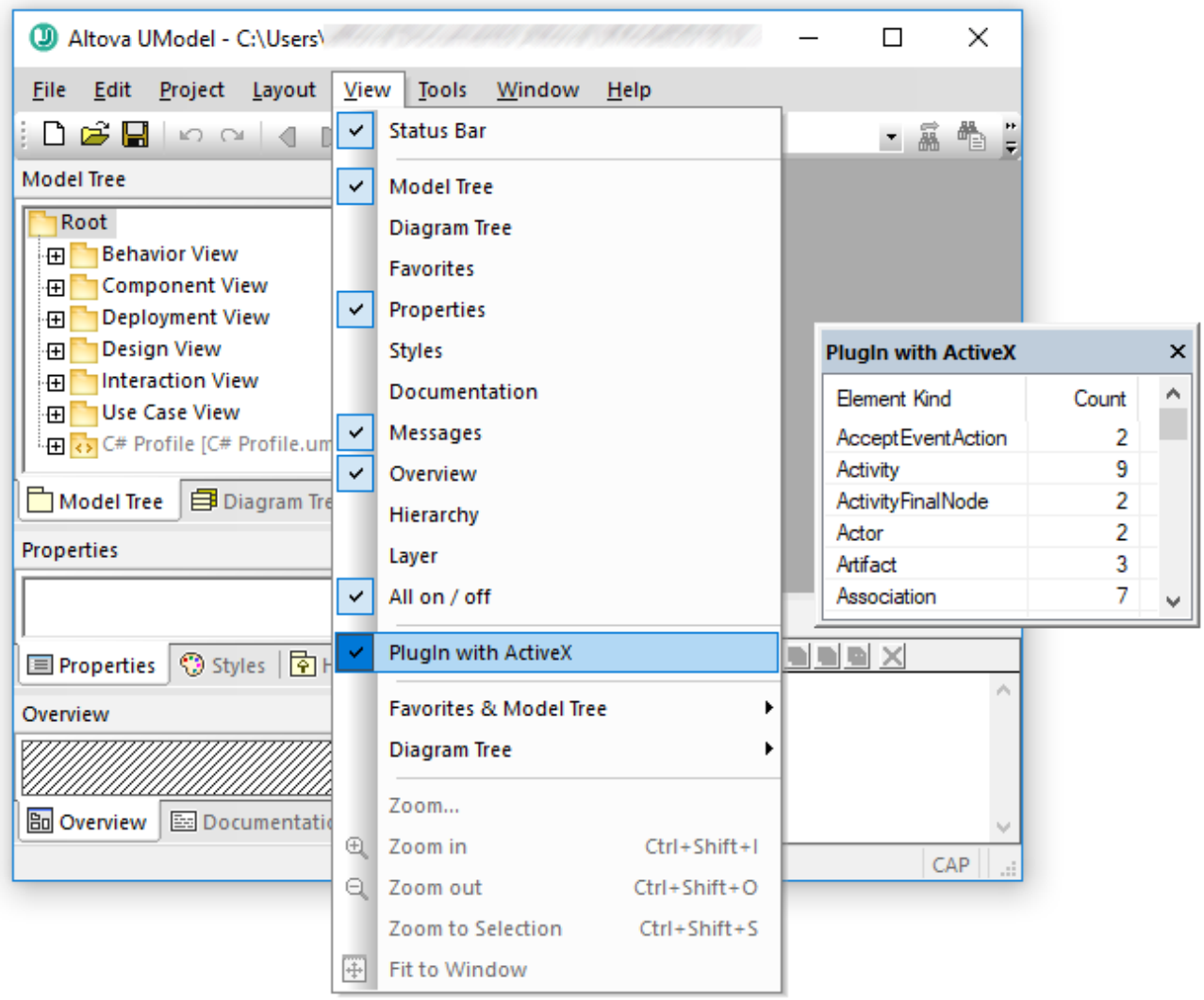

© 2019-2025 Altova GmbH

```
    <ID>3</ID>
    <Name>Fill red</Name>
    <Place>-1</Place>
    <MenuID>101</MenuID>
    <Parent>:PlugIn Menu1</Parent>
    <ImageID>0</ImageID>
  </UIElement>
```

UModel makes it possible to add toolbar separators and menus if the value of the `ID` element is set to 0.

16.2.4 Plug-Ins as ActiveX Controls

To work as an ActiveX control, the IDE plug-in must implement the `IOleControl` interface (C++) or derive from `System.Windows.Forms.UserControl` (C#, VB.NET). Such plug-ins will appear as a new window in the graphical user interface, and will also get a new menu command in the **View** menu.



The source code for the plug-in illustrated above is available in **C:\Users\<username>\Documents\Altova\UModel2025\UModelExamples\IDEPlugIn\StatisticsActiveX\StatisticsActiveX.cs**., see the "Statistics" Sample 791.

16.2.5 IUModelPlugIn Interface

If a DLL is added to UModel as a plug-in, it is necessary that it registers a COM component that answers to an `IUModelPlugIn` interface. The `IUModelPlugin` interface exposes the following methods, all of which must be implemented by a client plug-in.

- `OnInitialize`
- `OnRunning`
- `OnShutdown`
- `GetUIModifications`
- `GetDescription`
- `OnCommand`

© 2019-2025 Altova GmbH

- OnUpdateCommand

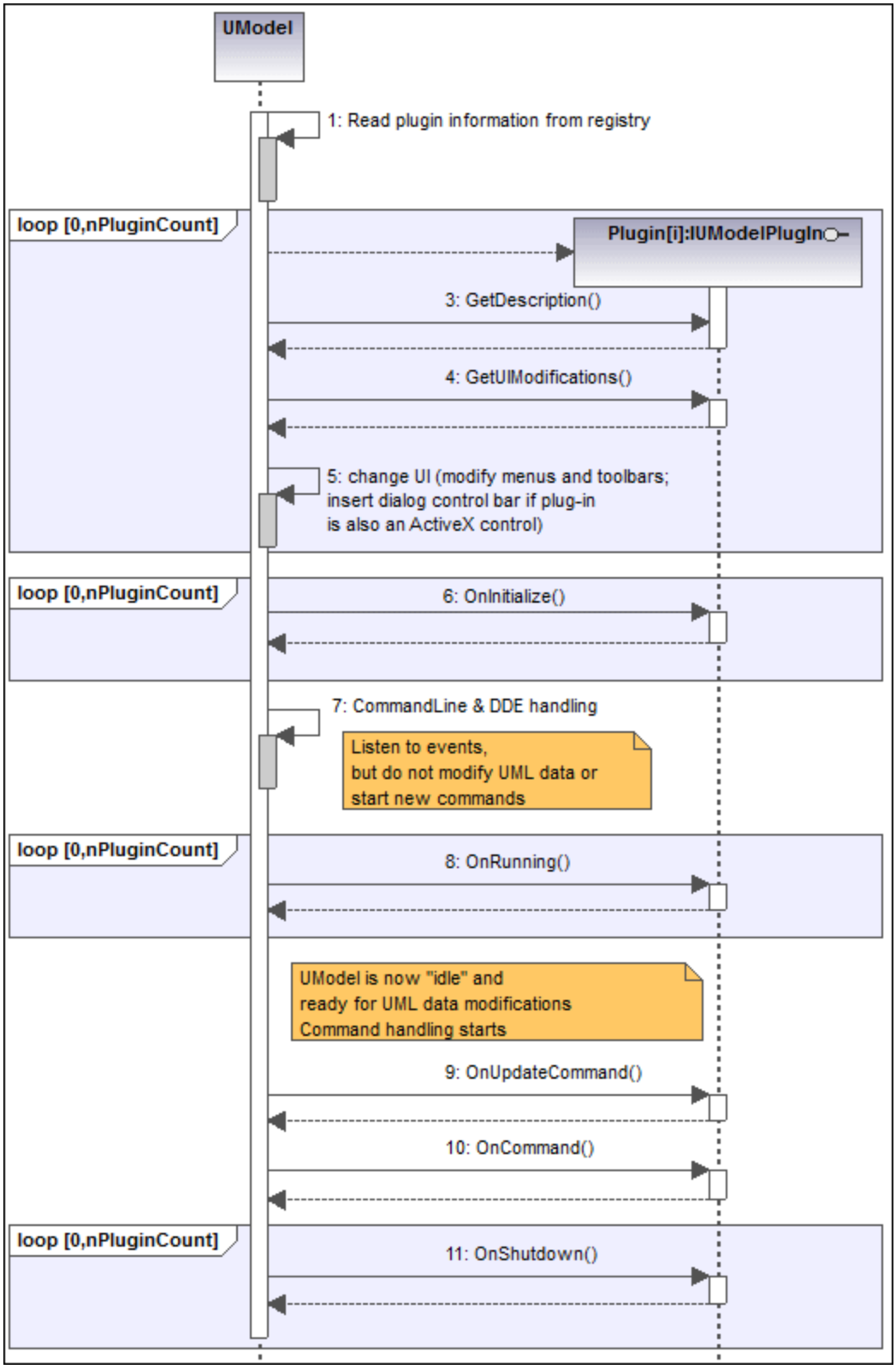
Method Declaration	Usage
`OnInitialize(pUModel as IDispatch)`	The `OnInitialize` method of the interface implementation is called when the plug-in is initialized and before DDE or batch commands are processed. You can attach notifiers and listen to UModel events, but should not start new commands / modifications until the `OnRunning` method is called. `pUModel` holds a reference to the dispatch interface of the `Application` object of UModel.
`OnRunning(pUModel as IDispatch)`	The `OnRunning` method of the interface implementation is called when the plug-in is initialized and after DDE or batch commands are processed. The application is now fully initialized and you can start new commands / modifications and modify UML data. `pUModel` holds a reference to the dispatch interface of the Application object of `UModel`.
`OnShutdown(pUModel as IDispatch)`	The `OnShutdown` method of the interface implementation is called immediately before the plug-in is unloaded (e.g. because the application will shut down). `pUModel` holds a reference to the dispatch interface of the `Application` object of UModel.
`GetUIModifications() as String`	The `GetUIModifications()` method is called during initialization of the plug-in, to get the configuration XML data that defines the changes to the UI of UModel. The method is called when the plug-in is loaded for the first time, and at every start of UModel. See Configuration XML 744 for a detailed description on how to change the UI.
`GetDescription() as String`	`GetDescription()` is used to define the description string for the plug-in entries visible in the Customize dialog box.
`OnCommand(nID as long, pUModel as IDispatch)`	The `OnCommand()` method of the interface implementation, is called each time a command, added by the plug-in (menu item or toolbar button), is processed. `nID` stores the command ID defined by the ID element of the respective UIElement.

Method Declaration	Usage
	`pUModel` holds a reference to the dispatch interface of the `Application` object of UModel.
`OnUpdateCommand(nID as long, pUModel as IDispatch) as UModelUpdateAction`	The `OnUpdateCommand()` method is called each time the visible state of a button, or menu item, needs to be set. `nID` stores the command ID defined by the ID element of the respective UIElement. `pUModel` holds a reference to the dispatch interface of the `Application` object. Possible return values (as defined in UModelUpdateAction)[815] to set the update state are: **UModelUpdateAction_Enable** = 1 **UModelUpdateAction_Disable** = 2 **UModelUpdateAction_Check** = 4 **UModelUpdateAction_Uncheck** = 8 Values can be combined using the bitwise OR operator (for example, `UModelUpdateAction_Enable | UModelUpdateAction_Check`).

For a very simple interface implementation example, see Implement IUModelPlugIn Interface[739]. Other sample implementations are available (as Visual Studio solutions) at the following path: **C:\Users\<username>\Documents\Altova\UModel2025\UModelExamples\IDEPlugIn**.

The sequence diagram below shows how UModel interacts with IUModelPlugIn[814]:

© 2019-2025 Altova GmbH


UModel
1: Read plugin information from registry
loop [0,nPluginCount]
Plugin[i]:IUModelPlugIn
3: GetDescription()
4: GetUIModifications()
5: change UI (modify menus and toolbars;
insert dialog control bar if plug-in
is also an ActiveX control)
loop [0,nPluginCount]
6: OnInitialize()
7: CommandLine & DDE handling
Listen to events,
but do not modify UML data or
start new commands
loop [0,nPluginCount]
8: OnRunning()
UModel is now "idle" and
ready for UML data modifications
Command handling starts
9: OnUpdateCommand()
10: OnCommand()
loop [0,nPluginCount]
11: OnShutdown()

16.3 The UModel API

The COM-based API of UModel enables clients to access the functionality of UModel from a custom code or application and automate a wide range of tasks.

The UModel API follows the common specifications for automation servers as set out by Microsoft. UModel is automatically registered as a COM server object during installation. Once the COM server object is registered, you can invoke it from within applications and scripting languages that have programming support for COM calls. This makes it possible to access the UModel API not only from development environments using .NET, C++ and Visual Basic, but also from scripting languages like JScript and VBScript. In Java, the UModel API is available through Java-COM bridge libraries.

Note: If you use the UModel API to create an application that you intend to distribute to other clients, UModel must be installed on each client computer. Also, your custom integration code must be deployed to (or your application installed on) each client computer.

16.3.1 Accessing the API

To access the COM API, a new instance of the `Application` object must be created in your application (or script). Once you have created the application object, you can start using the functionality of UModel. You will generally either open an existing document, create a new one, or access the active document (`IDocument` 832) 832. `IDocument` 832 corresponds to a UModel project and can be used to include sub-projects, generate documentation, synchronize model and code, while also giving access to the main UMLData objects, see also Object Model 753.

Note: When implementing a UModel IDE plugin, there is no need to create an instance of the application object, because UModel is already running and the current instance of the application object is provided by `IApplication` 818 as parameter for all important methods of `IUModelPlugIn` 748.

Prerequisites

To make the UModel COM object available in your Visual Studio project, add a reference to the UModel type library (.tlb) file, see How to Reference the UModel Type Library 772. A sample UModel API client in C# is available at: **C:\Users\<username>\Documents\Altova\UModel2025\UModelExamples\API\C#**.

In Java, the UModel API is available through Java-COM bridge libraries. These libraries are available in the UModel installation folder: **C:\Program Files (x86)\Altova\UModel2025\JavaAPI** (note this path is valid when 32-bit UModel runs on 64-bit Windows, otherwise adjust the path accordingly).

- `AltovaAutomation.dll`: a JNI wrapper for Altova automation servers
- `AltovaAutomation.jar`: Java classes to access Altova automation servers
- `UModelAPI.jar`: Java classes that wrap the UModel automation interface
- `UModelAPI_JavaDoc.zip`: a `Javadoc` file containing help documentation for the Java API

Note: In order to use the Java API, the .dll and .jar files must be on the Java `classpath`.

A sample UModel API client in Java is available at: **C:\Users\<username>\Documents\Altova\UModel2025\UModelExamples\API\Java**.

© 2019-2025 Altova GmbH

In scripting languages such as JScript or VBScript, the UModel COM object is accessible through the Microsoft Windows Script Host (see https://msdn.microsoft.com/en-us/library/9bbdkx3k.aspx). Such scripts can be written with a text editor, and do not need compilation, since they are executed by the Windows Script Host packaged with Windows. (To check that the Windows Script Host is running, type `wscript.exe /?` at the command prompt). Several JScript example files that call the UModel API are available at: **C:\Users\<username>\Documents\Altova\UModel2025\UModelExamples\API\JScript**.

Note: For 32-bit UModel, the registered name, or programmatic identifier (ProgId) of the COM object is `UModel.Application`. For 64-bit UModel, the name is `UModel_64.Application`. Be aware, though, that the calling program will access the CLASSES registry entries in its own registry hive, or group (32-bit or 64-bit). Therefore, if you run scripts using the standard command prompt and Windows Explorer on 64-bit Windows, the 64-bit registry entries will be accessed, which point to the 64-bit UModel. For this reason, if both UModel 32-bit and 64-bit are installed, special handling is required in order to call the 32-bit UModel. For example, assuming that Windows Scripting Host is the calling program, do the following:

1. Change the current directory to **C:\Windows\SysWOW64**.
2. At the command line, type **wscript.exe** followed by the path to the script that you would like to run, for example:

```
C:\Users\...\Documents\Altova\UModel2025\UModelExamples\API\JScript\Start.js
```

Guidelines

The following guidelines should be considered in your client code:

- Do not hold references to objects in memory longer than you need them. If a user interacts between two calls of your client, then there is no guarantee that these references are still valid.
- Be aware that if your client code crashes, instances of UModel may still remain in the system. For details on how to handle error messages, see Error handling 769.
- Free references explicitly, if using languages such as C or C++. In C# and Visual Basic, `GC.Collect()` can be used to force garbage collection.
- UModel API collections are zero-based. For example, the statement `myPackage.InsertPackagedElementAt(0, "Interface");` will insert a new interface as first child of the package.

16.3.2 Object Model

The starting point for every application which uses the UModel API is the `IApplication` 818 interface. The application object consists of the following main parts (each indentation level indicates a child–parent relationship with the level directly above):

- `IApplication` 818
  - `IDocument` 832
    - `IDiagramWindows` 828
      - `IDiagramWindow` 825
    - `IFocusedUMLDataNotifier` 840
    - `ITransactionNotifier` 889
    - `IUMLData` 904 (and all other derived UML data interfaces)
      - `IUMLDataList` 906
  - `IDialogs` 830

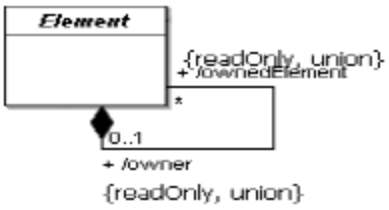
- IExportXMIFileDlg 839
- IGenerateDocumentationDlg 841
  - IKindSelectionList 865
    - IKindSelection 865
- IGenerateSequenceDiagramDlg 848
- IGenerateStateMachineCodeDlg 850
- IImportBinaryTypesDlg 851
  - IImportBinaryTypeEntries 821
    - IImportBinaryTypeEntry 822
- IImportDatabaseDlg 855
- IImportSourceDirectoryDlg 856
- IImportSourceProjectDlg 860
- IImportXMLSchemaDirectoryDlg 862
- IImportXMLSchemaFileDlg 863
- IIncludeSubprojectDlg 864
  - IModelTransformationDlg 881
    - IModelTransformationTypeMappings 884
      - IModelTransformationTypeMapping 883
- IProjectSettingsDlg 885
- ISaveAllDiagramsAsImagesDlg 888
- ISynchronizationSettingsDlg 888
  - IMatchRenamedDlg 878
    - IMatchRenamedEntries 879
      - IMatchRenamedEntry 880
  - IURLDlg 890
- ILocalOptions 866
  - ILocalOptionsCodeEngineering 868
  - ILocalOptionsDiagramEditing 870
    - ICollectionTemplates 824
      - ICollectionTemplate 823
  - ILocalOptionsEditing 873
  - ILocalOptionsFile 874
  - ILocalOptionsView 876

In addition, several Enumerations 896 and Events 891 are part of the model.

16.3.2.1 Object Model UMLData

The starting point to access UML elements is the root package (IUMLPackage 1134), which is a property of the IDocument 832 interface. All children of the root package are a subtype of IUMLElement 980 and are stored as defined by the OMG in the UML Superstructure Specification (http://www.uml.org). Specifically, the UML Superstructure Specification defines the following relationship for UML Element:

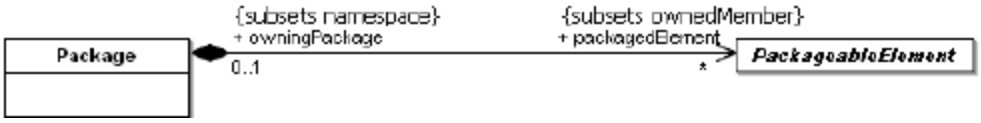
© 2019-2025 Altova GmbH



Which means that every UML element can have a list of owned elements, and every UML element (apart from the root-package) has an owner.

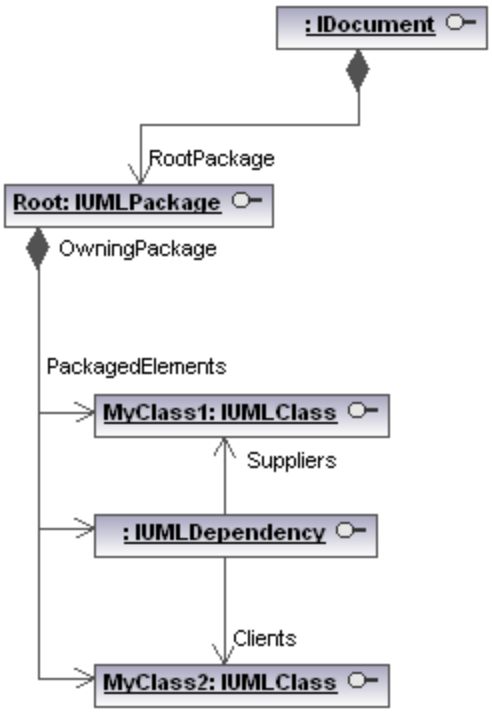
In the UModel API, an UML element is mapped to IUMLElement[1050] having the properties **`OwnedElement`** and **`Owner`**. Since these relationships are "read only" in the UML specification, both properties cannot be modified in the UModel API.

The UML Superstructure Specification also defines the following relationship between **`Package`** and **`PackageableElement`**:



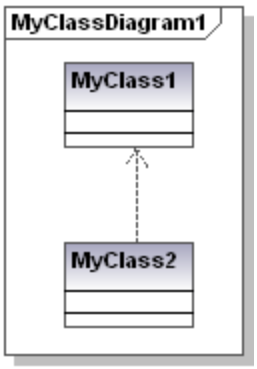
This is mapped to IUMLPackageableElement[1137] having a property **`OwningPackage`** and an IUMLPackage[1134], which not only has a property **`PackagedElements`**, but also a method **`InsertPackagedElementAt`** to insert new IUMLPackageableElement[1137]s (at the specified position). The method **`EraseFromModel`** deletes any IUMLElement[1050] (and all its children) from the model.

The sample below shows the mapping of a project which consists of two classes (IUMLClass[1015]) with a dependency (IUMLDependency[1041]) between them:



This structure is independent of whether these elements are shown on any diagram or not.

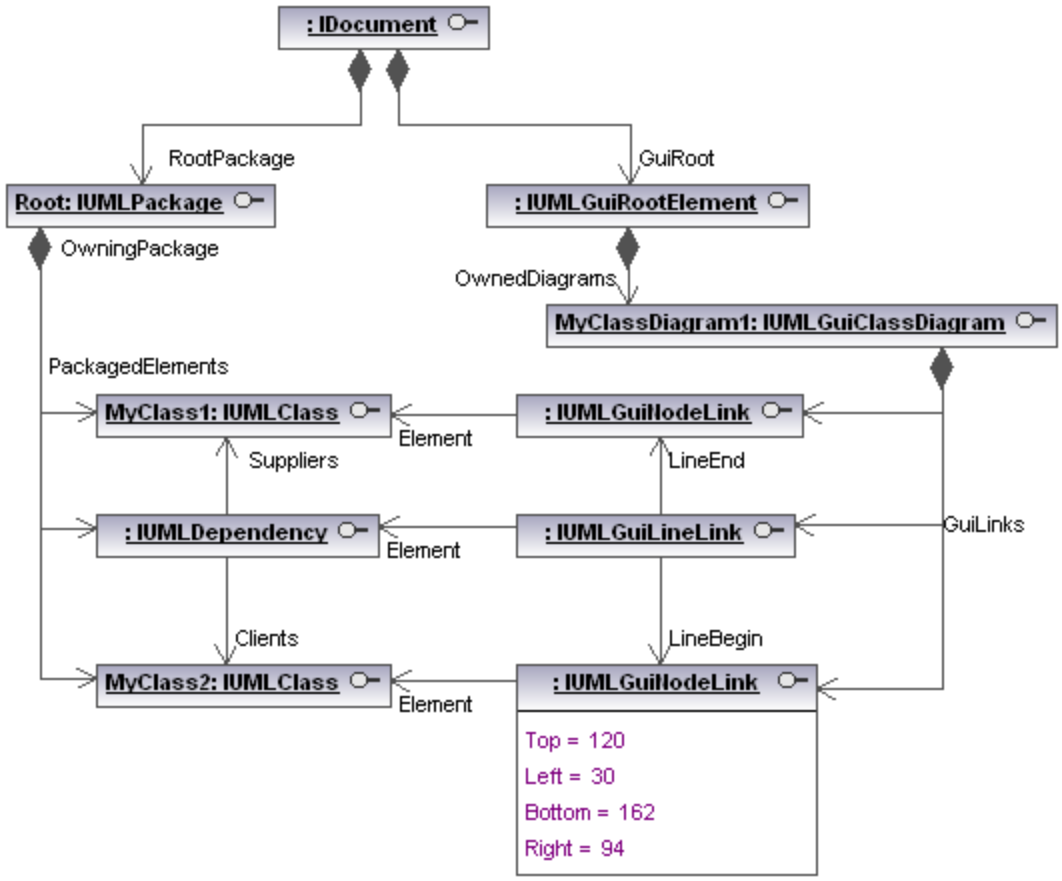
The representation of graphical objects on diagrams (as shown in the image below) is stored in a second structure with elements of kind IUMLGuiElement[1200] (also see Graphical Objects[758]).



The starting point to access UML GUI elements is the GuiRoot (IUMLGuiRootElement[1234]), which is a property of the IDocument[832] interface.

Lines are handled by IUMLGuiLineLink[1222]s, most other objects (like classes, interfaces, packages,...) by IUMLGuiNodeLink[1226]s.

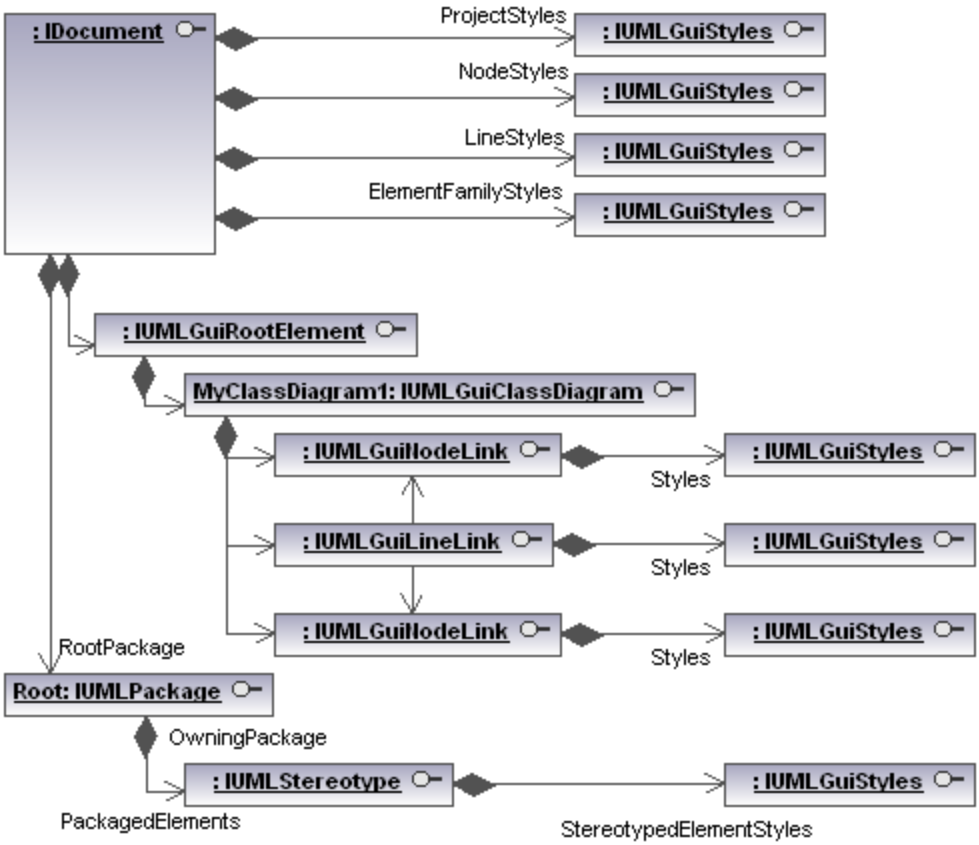
 © 2019-2025 Altova GmbH



16.3.2.2 Object Model UMLData Styles

UModel has various styles [90] allowing you to adapt the diagram appearance (i.e. font size, weight, color, visibility options,...).

The following picture shows how the different styles (IUMLGuiStyles [1241]) can be accessed using the UModel API [752]:



The different styles can be identified by ENUMUMLGuiStyleKind[1266].

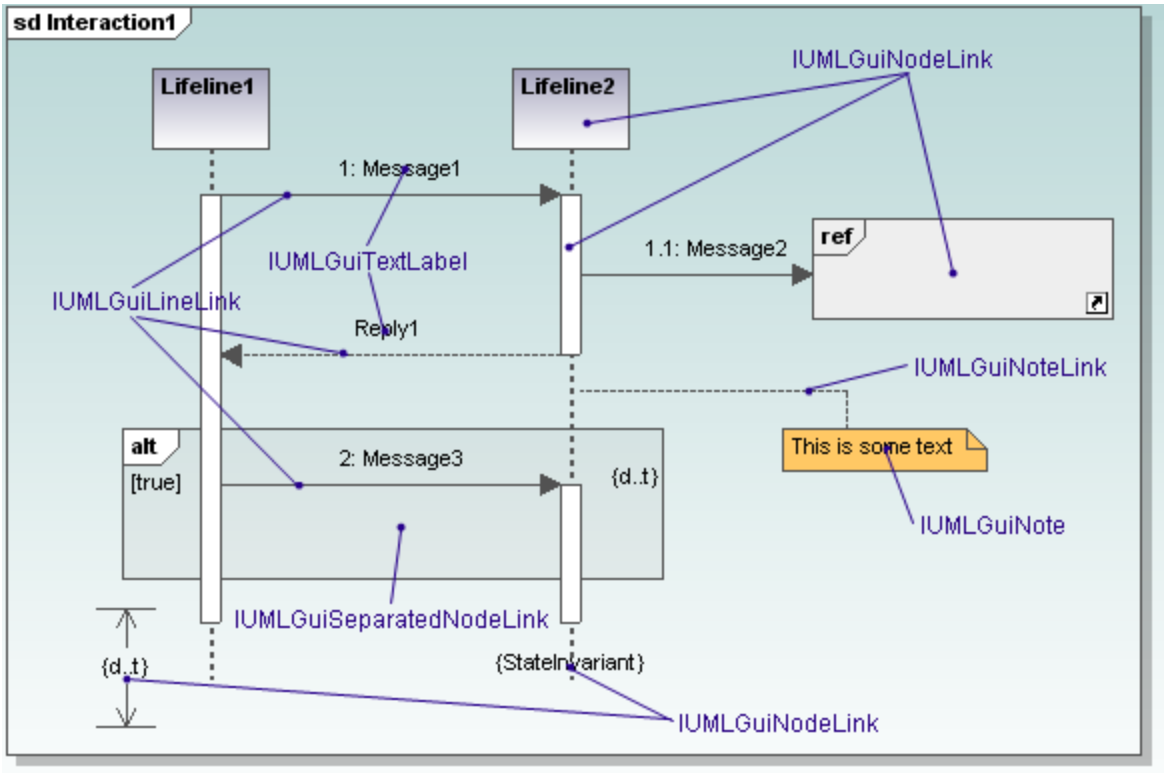
16.3.2.3 Graphical Objects

In the UModel API[752], graphical objects on diagrams are represented by objects derived from the IUMLGuiElement[1200] interface. Most of them can be accessed using the IUMLGuiDiagram[1211] property 'GuiLinks'.

For most diagrams, most objects which are lines are instances of IUMLGuiLineLink[1222] and most other, solid objects or 'nodes' are instances of IUMLGuiNodeLink[1226]. These interfaces have properties and methods for manipulating the basic properties of these graphical objects, such as position, color and style.

There are of course more specialized interfaces derived from these general interfaces which provide access to special properties. The following image shows a sequence diagram and the interface representing each graphical object on it:

© 2019-2025 Altova GmbH



16.3.3 How to...

16.3.3.1 How to Create Sequence Diagrams

There are two ways to create sequence diagrams programmatically using the UModel API:

- Generating a sequence diagram from existing source code[759] when there is code available that you want to be reverse engineered and displayed as UML diagram
- Manually create a sequence diagram[760] from scratch using `IUMLGuiElements` directly

16.3.3.1.1 How to Generate Sequence Diagrams from Code

Sequence diagrams in UModel can be generated programmatically from an IUMLOperation[1131] element. The operation needs to exist in the model and have some source code associated to it.

The operation could possibly have been previously "read" by the reverse engineering functionality of UModel. Creating new Sequence Diagrams programmatically by reverse engineering source code using the UModel API[752] involves two short steps:

- Setting up the options for diagram generation
- Invoking the diagram generation function

The following C# code shows how to set up the options and start the generation of the sequence diagram:

```
// starts the sequence diagram generation process based on an operation given as
parameter
public static void reverseEngineerAndCreateSequenceDiagram(IApplication application,
IUMLOperation operation)
{
    GenerateSequenceDiagramDlg dialog = application.Dialogs.GenerateSequenceDiagramDlg;

    // set some options
    dialog.ShowEmptyCombinedFragments = false;
    dialog.UseDedicatedLineForStaticCalls = true;
    dialog.ShowCodeOfMessagesDisplayedDirectlyBelow = true;
    dialog.ShowCodeInNotes = true;

    dialog.ShowDialog = true; // set this to true if you want the dialog to be displayed

    // generated the sequence diagram now
    application.ActiveDocument.GenerateSequenceDiagram(dialog, operation);
}
```

16.3.3.1.2 How to Create Sequence Diagrams Manually

Creating new Sequence Diagrams programmatically from scratch using the UModel API[752] is basically nothing more than placing interaction fragments, such as Lifelines on a diagram and connecting them with messages.

Messages can easily be created using the `AddUMLLineElement()` method of `IUMLGuiLineLink`[1224], which removes the necessity of creating multiple underlying UML Elements such as MessageEnds, ExecutionOccurrences and similar manually.

To make it simple to create Messages between two interaction fragments such as Lifelines, create a small helper function which calls `AddUMLLineElement()` and positions the created line:

```
// Creates a message between two interaction fragments (i.e. lifelines, interaction uses,
// combined fragments or gates) and attaches all necessary elements like events and
activation bars.
// Possible values for 'kind': "Message", "Reply", "Create", "Destruct"
protected static IUMLMessage addMessage(int ypos, string kind,
                                IUMLGuiNodeLink from, IUMLGuiNodeLink to,
                                DiagramWindow wnd)
{
    // add message
    IUMLGuiLineLink line = wnd.Diagram.AddUMLLineElement(kind, from, to);

    if (line == null)
        return null;

    // set position of the line where we want it to show up
    wnd.UpdateWindow();

    if (from == to && line.Waypoints.Count > 3)
    {
```


© 2019-2025 Altova GmbH

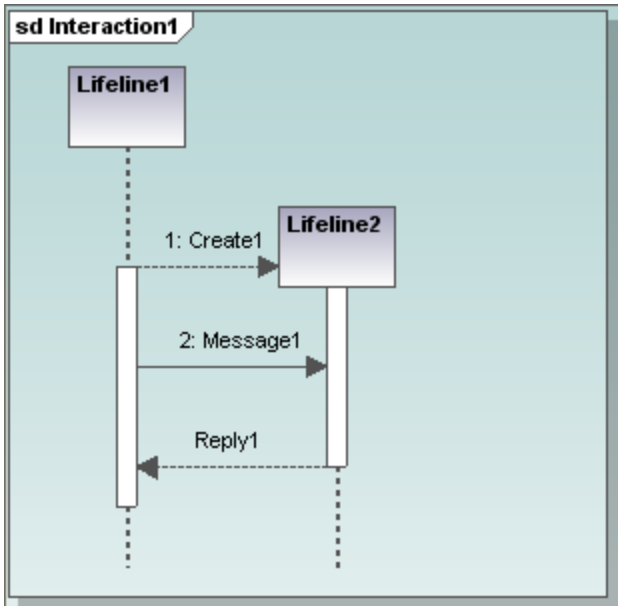
```
        // self-message
        ((IUMLGuiWaypoint)line.Waypoints[1]).SetPos(0, ypos);
        ((IUMLGuiWaypoint)line.Waypoints[4]).SetPos(0, ypos + 25);
    }
    else
    if (line.Waypoints.Count > 1)
    {
        // normal message
        ((IUMLGuiWaypoint)line.Waypoints[1]).SetPos(0, ypos);
        ((IUMLGuiWaypoint)line.Waypoints[2]).SetPos(0, ypos);
    }

    return (IUMLMessage)line.Element;
}
```

As you can see, IUMLDiagram.AddUMLLineElement() 1211 accepts as a parameter not only the string "Message", to create a Message Line; but also "Reply", "Create" and "Destruct", for Reply Messages, Creation Messages and Destruction Messages.

In order to create a simple diagram it is only necessary to create a Sequence Diagram in the GuiRoot object, open the diagram, add a handful of lifelines and connect them with messages using this helper function:

```
IDocument document = theapplication.ActiveDocument;

// create diagram and open it
IUMLGuiSequenceDiagram sequenceDiagram =
   (IUMLGuiSequenceDiagram)document.GuiRoot.InsertOwnedDiagramAt(0, document.RootPackage,
 "SequenceDiagram");

DiagramWindow wnd = document.OpenDiagram(sequenceDiagram);

// create two lifelines
IUMLGuiNodeLink lifeline1 = sequenceDiagram.AddUMLElement("Lifeline", 0, 0);
IUMLGuiNodeLink lifeline2 = sequenceDiagram.AddUMLElement("Lifeline", 100, 70);

// connect these lifelines using some messages
addMessage(100, "Create",  lifeline1, lifeline2, wnd);
addMessage(150, "Message", lifeline1, lifeline2, wnd);
addMessage(200, "Reply",   lifeline2, lifeline1, wnd);
```

The resulting created Diagram will look like this:



Setting the Type of a Lifeline

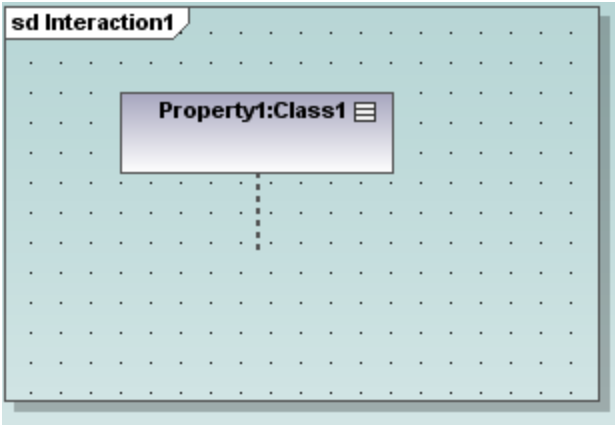
To display the Type represented by a Lifeline, be it be a Class, Interface, DataType or similar, use the IUMLLifeline.Represents[1104] property which references a IUMLProperty[1147]. If the Type of this property is set, the Type will show up on the diagram as well.

The following code creates a Lifeline which references a class:

```
// create a class to be referenced by the lifeline
IUMLClass someclass = (IUMLClass)document.RootPackage.InsertPackagedElementAt(0,
"Class");

// create a lifeline and a property with the class as type in the interaction
// of the sequence diagram to reference this class
IUMLInteraction interaction = (IUMLInteraction)sequenceDiagram.LinkedOwner;

IUMLProperty prop = interaction.InsertOwnedAttributeAt(0);
prop.Type = someclass;

UModelLib.IUMLLifeline lifeline = interaction.InsertLifelineAt(0);
lifeline.Represents = (IUMLConnectableElement)prop;

// show the lifeline on the diagram
sequenceDiagram.AddUMLGuiNodeLink(lifeline, 200, 0);
```

The resulting lifeline would then look like this:

© 2019-2025 Altova GmbH



Setting the Operation of a Message

Messages usually represent the invocation of an operation of an object. Note: based on the type of the Message (normal Message, Creation, Deletion or Reply) and the existence, or absence of underlying UML elements, such as MessageOccurenceSpecifications or CallEvents, it is not always possible for a Message to represent an Operation, and getting the cprrect UML element to point to the Operation is not that trivial.

This is why the `IUMLMessage`[1111] interface in the UModel API, offers the method `SetOperation()` with makes it possible to let a Message refer an Operation if it is able to do so:

```
// create a message, an operation in a class and let the message refer this operation
IUMLMessage msg = addMessage(250, "Message", lifeline1, lifeline2, wnd);

UModelLib.IUMLOperation someoperation = someclass.InsertOwnedOperationAt(0);
someoperation.Name = "SomeOperation";

msg.SetOperation(someoperation);
```

16.3.3.2 Undo / Redo and UMLData Transaction Handling

When modifying the UML data structure using the UModel API[752], there is no need to take care of Undo/Redo or transactions.

The following code makes three modifications:

```
        public void ChangeClass( IUMLClass iClass )
        {
            iClass.SetName("NewName");
            iClass.Visibility = ENUMUMLVisibilityKind.eVisibility_Public;
            iClass.IsAbstract = true;
        }
```

and for every modification, a new undo-step is created, in other words: the user will have to press the "Undo" button three times in UModel to undo these three changes.

This is not always the required behavior so the UModel API[752] supports "transaction-handling" making it possible to execute multiple modifications in one step.

IDocument[832] has the functionality to define when a group of modifications starts ("BeginModification") and when it ends ("EndModification"):

```
        public void ChangeClass(IUMLClass iClass, IDocument iDoc)
        {
            try
            {
                // make all modifications within one UndoStep; start modification here
                if (!iDoc.BeginModification())
                    return;

                iClass.SetName("NewName");
                iClass.Visibility = ENUMUMLVisibilityKind.eVisibility_Public;
                iClass.IsAbstract = true;

                // do not forget to end modification and finish UndoStep
                iDoc.EndModification();
            }
            catch (System.Exception)
            {
                // rollback made changes
                iDoc.AbortModification();

                // add error handling
            }
        }
```

This kind of transaction handling may only be used for UML data modifications. Other functions, such as e.g. 'synchronize model from code', will create one single Undo step anyway.

16.3.3.3 How to Use Predefined UModel Elements

UModel defines several important elements as "predefined". This includes several internal elements (Root, Component View and Unknown Externals package) as well as the elements of all profiles installed with UModel (e.g. the C#, VB and Java profile).

Predefined elements can be uniquely identified by using ENUMUMLPredefinedElement[1271], which allows direct and easy access to these elements for several functionalities, for example:

- Find a predefined element:

```
// get the CSharp profile
IUMLProfile iCSharpProfile = (IUMLProfile)
iDoc.RootPackage.FindPredefinedOwnedElement(ENUMUMLPredefinedElement.ePredefined_CSharp_P
rofile, false);
```

- Apply a predefined stereotype:

```
// set the CSharp 'delegate' stereotype
```


© 2019-2025 Altova GmbH

```
iClass.ApplyPredefinedStereotype(
 ENUMUMLPredefinedElement.ePredefined_CSharp_delegateStereotypeOfClass );
```

- Check if a predefined stereotype is applied:

```
// check if package is a CSharp - namespace (if 'namespace' stereotype is applied)
bool bIsCSharpNamespace =
iPackage.IsPredefinedStereotypeApplied(
 ENUMUMLPredefinedElement.ePredefined_CSharp_namespaceStereotypeOfPackage );
```

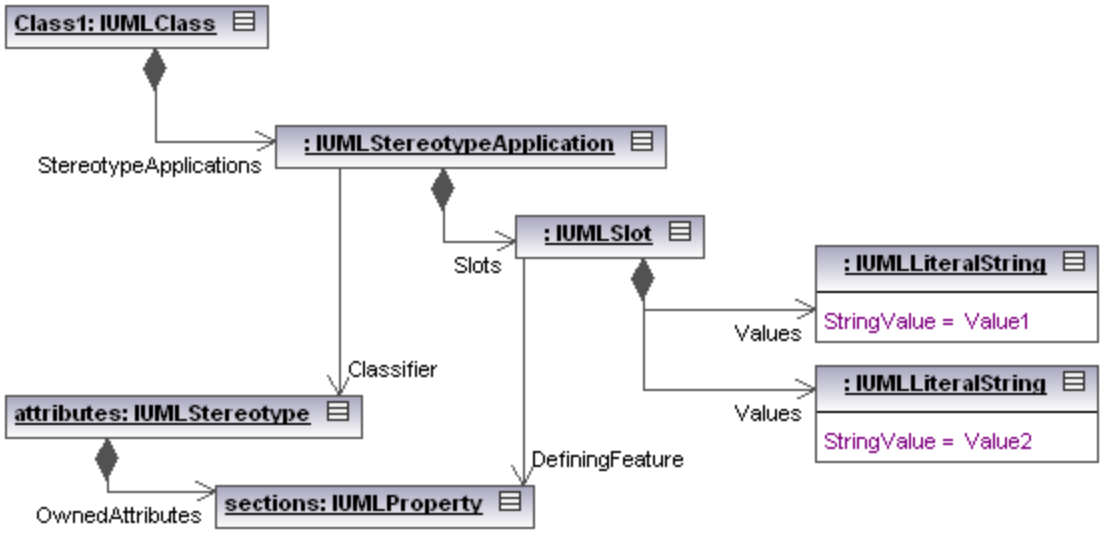
- Set the tagged value of a predefined tag definition:

```
// set attribute-section "STAThread"
// ...
iStereotypeApp.SetPredefinedTaggedValueAt(-1,
ENUMUMLPredefinedElement.ePredefined_CSharp_attributesStereotypeOfClass_sectionsProperty,
 iSTAThread.Name);
```

16.3.3.4 How to Work with Stereotypes and Tagged Values

Stereotypes and tagged values are quite complex as defined in the UML Superstructure Specification. UModel has simplified their handling and treats them similar to IUMLInstanceSpecification[1085]s and IUMLSlot[1162]s in UML. In the following sample, the stereotype "attributes" is applied to "Class1", and the tag definition "sections" has the tagged values "Value1" and "Value2":

UModel API introduces IUMLStereotypeApplication[1170]s and maps the sample above to the following UMLData structure:



Applying stereotypes and setting tagged values using the UModel API is quite simple:

```
IUMLStereotype iStereotypeAttributes = ...;
IUMLProperty iTagDefSections = ...;
IUMLClass iClass = ...;

IUMLStereotypeApplication iStereotypeApp = iClass.ApplyStereotype(iStereotypeAttributes);
iStereotypeApp.SetTaggedValueAt(-1, iTagDefSections, "Value1");
iStereotypeApp.SetTaggedValueAt(-1, iTagDefSections, "Value2");
```

See also the section Predefined UModel elements [764] for information about dealing with predefined stereotypes, tag definitions and tagged values.

16.3.3.5 How to Use UMLData Events and Event Filters

Event receivers must implement the IUMLDataEvents [1263] interface in order to receive one or more of following possible events from IUMLData [904] objects:

OnBeforeErase	Sent immediately before the UML data is erased from the model. If multiple data are erased, this event is sent for every IUMLData [904] (not only for the topmost one).
OnAfterAddChild	Sent when the UML data is added to the model tree. If multiple data are added in one step (e.g. a class with multiple attributes is added to a package) only the topmost IUMLData [904] event is sent.
OnChanged	Sent when the UML data has been modified (e.g. when a class name is changed).

© 2019-2025 Altova GmbH

OnMoveData	Sent when the UML data has been moved to a new parent (e.g. when a class is moved to another package in the ModelTree). This event always occurs twice: once when detaching from the old parent, and once when the UML data is attached to the new parent.

Eventfilter can be set with (combinations of) ENUMUMLDataEventFilter 1265 in order to specify which events should be sent by the UModel API 752. To keep performance high and the overhead as low as possible, event receivers should only register for events they need.

For example, the following code registers OnAfterAddChild events when specifically the root-package gets a new child (no event will arrive if a child of the root-package gets a new child):

```
// ensure we get informed when m_RootPackage (and only itself; we do not care about its
children) gets a new child
m_RootPackage.EventFilter = (int)ENUMUMLDataEventFilter.eUMLDataEvent_AddChild;
```

UMLData events work hierarchically, so the event filter can be set to receive events from the attached IUMLData 904 only, or from the attached IUMLData 904 and any of its children (grandchildren,...).

```
// ensure we get "OnBeforeErase" events also for *any* erased child (grandchild,...) of
the rootpackage
m_RootPackage.EventFilter |= (int)ENUMUMLDataEventFilter.eUMLDataEvent_EraseDataOrChild;
```

UMLData events are also sent when UML data is modified by Undo / Redo, but beware that no UML data modification may be made during Undo / Redo:

```
public void OnAfterAddChild(IUMLData ipUMLParent, IUMLData ipUMLChild)
{
   // check if child was added by undo/redo
      // (we are not allowed to modify anything during Undo/Redo !!)
      IDocument iDoc = (IDocument)ipUMLChild.Parent;
      if (!iDoc.IsInUndoRedo)
      {
         // ...
      }
}
```

16.3.3.6 How to Create and Use Hyperlinks

UModel allows hyperlinks between most modeling elements (except for lines) and:

- any diagram in the current ump project
- any element on a diagram
- any element in the Model Tree
- external documents, e.g. PDF, Excel or Word documents
- web pages

See also: Hyperlinking modeling elements 118.

Hyperlinks are not part of the UML specification and the UModel API introduces the following interfaces for hyperlinks on

IUMLNamedElement[1117]s:

- IUMLHyperlink[1075] is the common base interface and can be used to open links as well as to retrieve the default- and user-defined link name
- IUMLHyperlink2File[1076] to handle external documents and web pages
- IUMLHyperlink2GuiElement[1077] to handle any diagram in the current ump project or any element on a diagram
- IUMLHyperlink2Model[1078] for hyperlinks to model elements (in the Model Tree)

Examples

Insert a hyperlink to the Altova homepage:

```
IUMLHyperlink2File iHyperlink = iMyClass.InsertOwnedHyperlink2FileAt(-1,
"http://www.altova.com");
```

Insert a hyperlink to a diagram of the current ump project:

```
IUMLGuiDiagram iDiagram = ...;
IUMLHyperlink2GuiElement iHyperlink = iMyClass.InsertOwnedHyperlink2GuiElementAt(-1,
iDiagram, null);
```

Insert a hyperlink to the representation of a class on a diagram:

```
IUMLGuiNodeLink iNodeLink = ...;
IUMLHyperlink2GuiElement iHyperlink = iMyClass.InsertOwnedHyperlink2GuiElementAt(-1,
iNodeLink, null);
```

Insert a hyperlink to an attribute of a class on a diagram:

```
IUMLGuiNodeLink iNodeLink = ...;
IUMLProperty iAttribute = ...;
IUMLHyperlink2GuiElement iHyperlink = iMyClass.InsertOwnedHyperlink2GuiElementAt(-1,
iNodeLink, iAttribute);
```

Insert a hyperlink to the same attribute (from above) in the Model Tree:

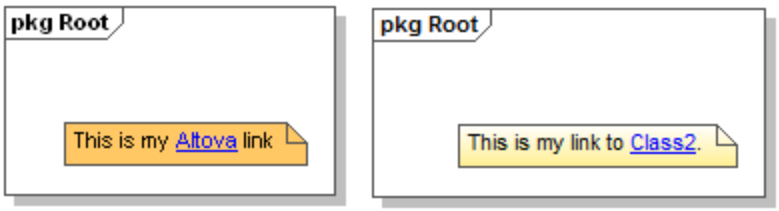
```
IUMLHyperlink2Model iHyperlink = iMyClass.InsertOwnedHyperlink2ModelAt(-1, iAttribute);
```

Open all hyperlinks of an IUMLNamedElement[1117]:

```
foreach (IUMLHyperlink iHyperlink in iMyClass.OwnedHyperlinks)
   iHyperlink.OpenLink();
```

UModel also allows hyperlinks in notes (IUMLGuiNote[1228]) and comments (IUMLComment[1025]):

© 2019-2025 Altova GmbH



These are handled by IUMLGuiTextHyperlink 1250s (respectively IUMLCommentTextHyperlinks 1026) and the start- and end-character position of the hyperlink must be specified, e.g:

```
IUMLGuiDiagram iDiagram = ...;
IUMLGuiNote iNote = iDiagram.AddUMLGuiNote(200, 100);

iNote.NoteText = "This is my Altova link";
int nStart = iNote.NoteText.IndexOf("Altova");
int nEnd = nStart + "Altova".Length;

IUMLGuiTextHyperlink iHyperlink = iNote.InsertOwnedGuiTextHyperlinkAt(nStart, nEnd,
"http://www.altova.com");
```

Similar for hyperlinks in comments:

```
IUMLComment iComment = ...;
IUMLClass iClass2 = ...;

iComment.Body = "This is my link to Class2";
int nStart = iComment.Body.IndexOf("Class2");
int nEnd = nStart + "Class2".Length;

IUMLCommentTextHyperlink iHyperlink = iComment.InsertOwnedCommentTextHyperlinkAt(nStart,
nEnd, "");
iHyperlink.SetHyperlinkModelElementAddress( iClass2 );
```

16.3.3.7 Handle Errors

The UModel API returns errors in two different ways. Every API method returns an `HRESULT`. This return value informs the caller about any errors during the execution of the method. If the call was successful, the return value is equal to `S_OK`. C/C++ programmers generally use `HRESULT` to detect errors.

VisualBasic, scripting languages, and other high-level development environments do not give the programmer access to the returning `HRESULT` of a COM call. They use the second error-raising mechanism supported by the UModel API, the `IErrorInfo` interface. If an error occurs, the API creates a new object that implements the `IErrorInfo` interface. The development environment takes this interface and fills its own error-handling mechanism with the provided information.

The example code listings below show how to deal with errors raised from the UModel API in different development environments.

Visual Basic

A common way to handle errors in VisualBasic is to define an error handler. This error handler can be set with the `On Error` statement. Usually the handler displays an error message and performs cleanup functions to avoid spare references and any kind of resource leaks.

VisualBasic fills its own `Err` object with the information from the `IErrorInfo` interface.

```
Sub Validate()
  'place variable declarations here

  'set error handler
  On Error GoTo ErrorHandler

  'if DoSomeWork fails, program execution continues at ErrorHandler:
  objUModel.ActiveDocument.DoSomeWork()

  'additional code comes here

  'exit
  Exit Sub

  ErrorHandler:
  MsgBox("Error: " & (Err.Number - vbObjectError) & Chr(13) &
     "Description: " & Err.Description)
End Sub
```

JavaScript

The Microsoft implementation of JavaScript (JScript) provides a try-catch mechanism to deal with errors raised from COM calls. It is very similar to the Visual Basic approach, in that you also declare an error object containing the necessary information.

```
   function Generate()
   {
      // please insert variable declarations here

      try
      {
         objUModel.ActiveDocument.DoSomeWork();
      }
      catch(Error)
      {
         sError = Error.description;
         nErrorCode = Error.number & 0xffff;
         return false;
      }

      return true;
   }
```

C/C++

C/C++ gives you easy access to the `HRESULT` of the COM call and to the `IErrorInterface`.

© 2019-2025 Altova GmbH

```
HRESULT hr;

// Call DoSomeWork() from the UModel API
if(FAILED(hr = ipDocument->DoSomeWork()))
{
   IErrorInfo *ipErrorInfo = Null;

   if(SUCCEEDED(::GetErrorInfo(0, &ipErrorInfo)))
   {
      BSTR   bstrDescr;
      ipErrorInfo->GetDescription(&bstrDescr);

      // handle Error information
      wprintf(L"Error message:\t%s\n",bstrDescr);
      ::SysFreeString(bstrDescr);

      // release Error info
      ipErrorInfo->Release();
   }
}
```

16.3.4 C# API Examples

To help you get started, your UModel package contains an example C# project, which is located at **C:\Users\<username>\Documents\Altova\UModel2025\UModelExamples\API**.

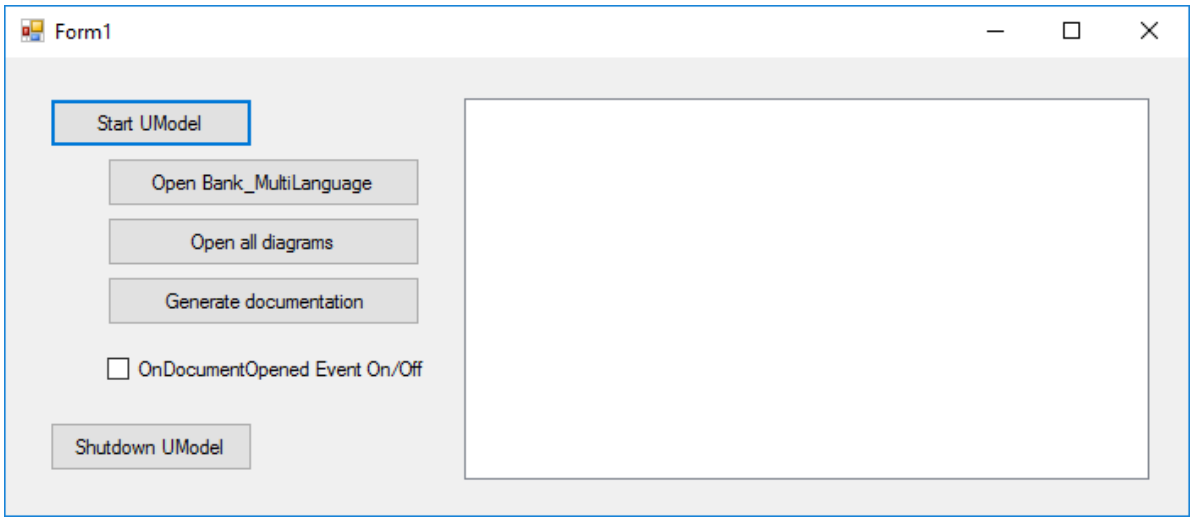
Importantly, this example project includes a reference to the UModel Type Library, see How to Reference the UModel Type Library 772. A reference to the UModel Type Library is required in each project where you need the UModel API. This makes it possible to instantiate the main application object from your code as follows:

```
UModelLib.Application um = new UModelLib.Application();
MessageBox.Show(String.Format("Hello from UModel API version {0}.{1}",
um.APIMajorVersion, um.APIMinorVersion));
```

If you have a 64-bit operating system and are using a 32-bit installation of UModel, add the x86 platform in the solution's Configuration Manager and build the sample using this configuration. To access Configuration Manager, run the menu command **Build | Configuration Manager**.

The example application displays a Windows form with buttons that invoke basic UModel operations:

- Start UModel
- Open **Bank_MultiLanguage.ump**
- Open All Diagrams
- Generate documentation for the currently active document
- Shows how to listen to UModel events (`OnDocumentOpened` Event On/Off)
- Shutdown UModel

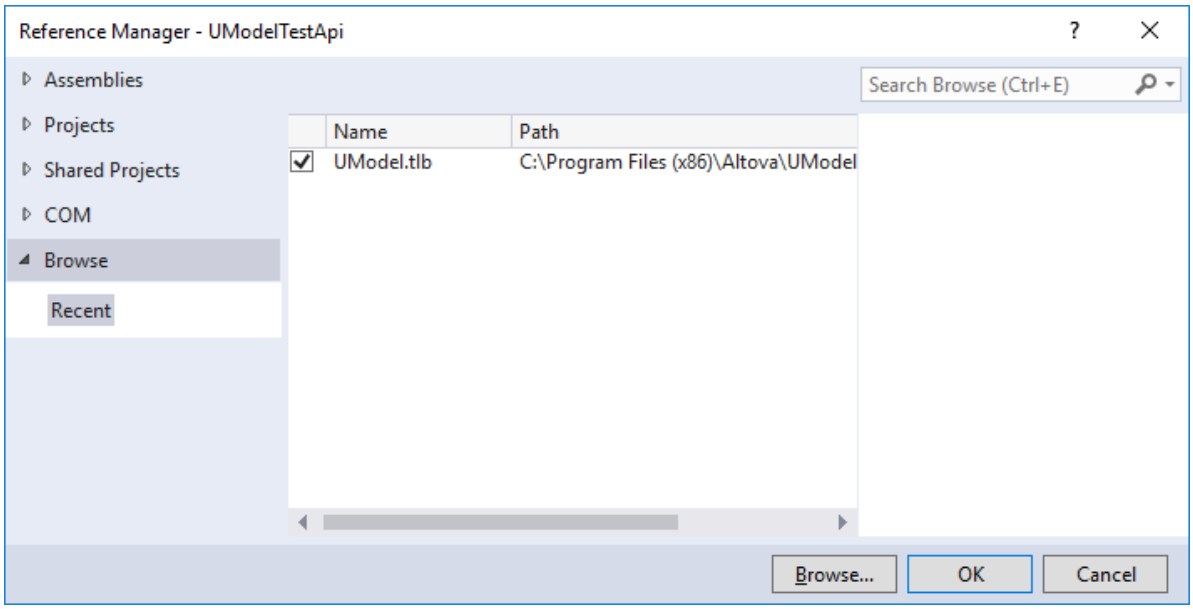


The code essentially consists of a series of handlers for the buttons in the user interface shown above. Note that you may need to adjust the path to the UModel examples folder which is referenced from the code.

16.3.4.1 How to Reference the UModel Type Library

To access the API functionality of UModel from your Visual Studio project, add a reference to the UModel Type Library in Visual Studio, as follows:

1. Create a new Visual Studio project, or open an existing one.
2. On the **Project** menu, click **Add Reference**.
3. In the COM section, select **UModel Type Library** from the list. If this entry is not available in the COM section, click **Browse** and select the file **UModel.tlb** from the UModel program application folder.

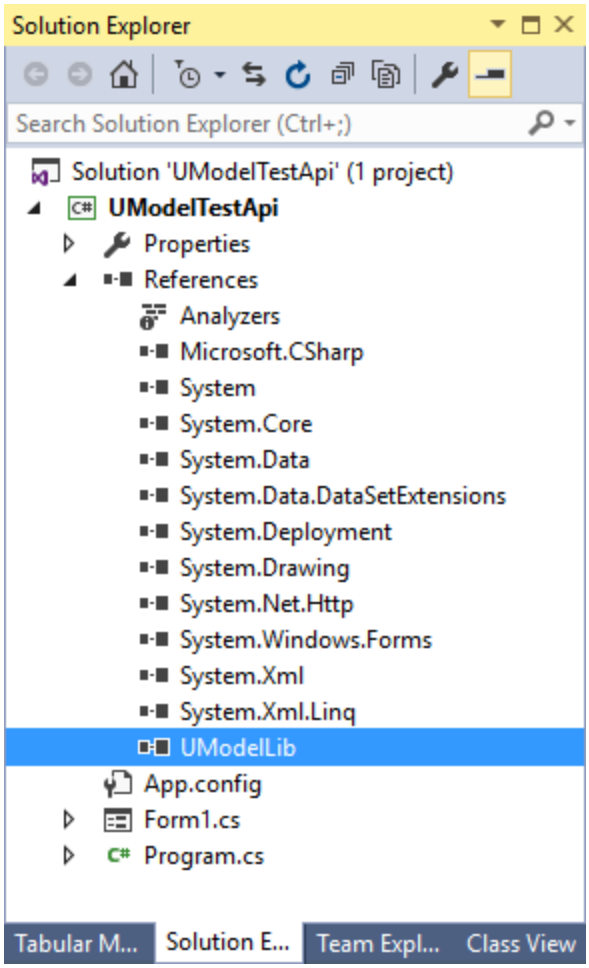


Note: Do not confuse the **UModel Type Library** with the **UModelPlugin Type Library**. The latter can be used to create your own plug-ins and integrate them into UModel, see Add Reference to UModel Plug-In

© 2019-2025 Altova GmbH

Library[734].

After you follow the steps above, the UModel Type Library should be available in the list of references of your Visual Studio solution, for example:



16.3.4.2 Importing Binary Types Programmatically

With UModel, you can import binary types from .NET .dll or Java .jar files, either from the graphical user interface, or programmatically using the UModel API. This example illustrates how to import binary types from a NET .dll file into UModel using the UModel API. For information about importing binary types from the graphical user interface, see Importing Java, C# and VB.NET Binaries[204].

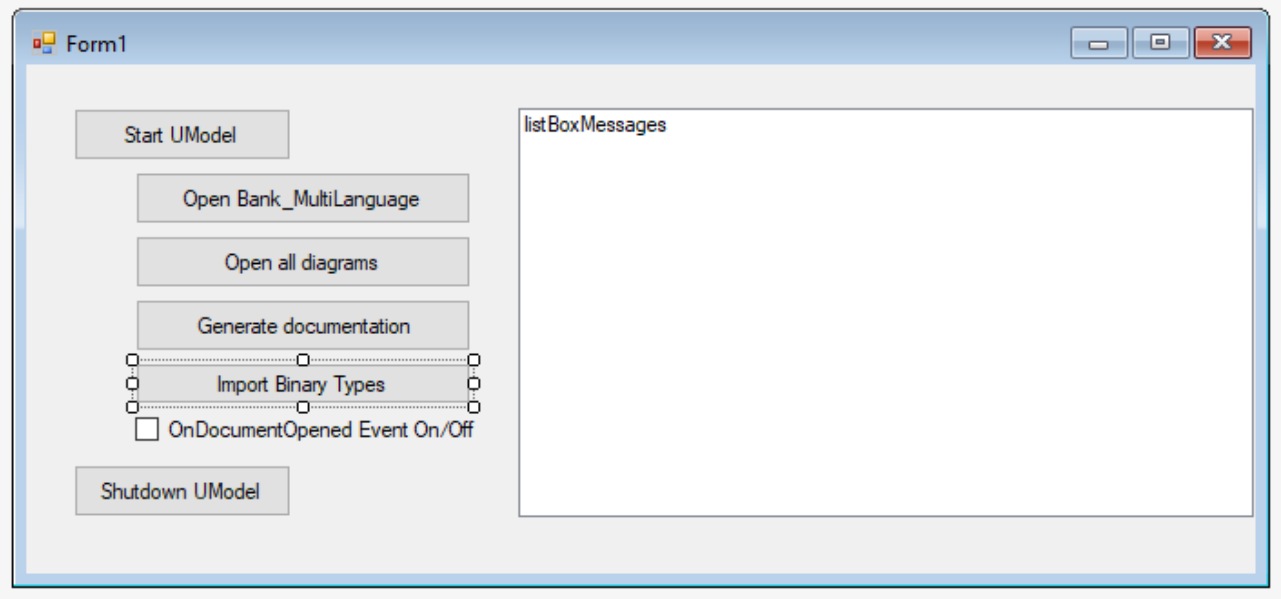
This example uses Microsoft Visual Studio 2019 and C#. The instructions below (except for the code listing) are similar for VB.NET. To complete this example, you also need a .dll that contains some types (such as classes or interfaces) that you would like to import into UModel.

To accomplish the task, we will use an existing C# demo application that already integrates into the UModel API, rather than creating a new project from scratch. Namely, we will add to this demo application a new button. When clicked, the button will create a new UModel project and import into it types from a .dll file. To begin, run Visual Studio and open the following solution: **C:**

\Users\<username>\Documents\Altova\UModel2025\UModelExamples\API\C#\AutomateUModel_VS2010.sln.

Note: The demo application already includes a reference to the UModel Type Library so it is not necessary to add a reference explicitly. However, if you are creating a new Visual Studio project, make sure to reference the UModel Type Library from your project, see How to Reference the UModel Type Library 772.

Next, open the `Form1.cs` in the Design Editor and add a new button. Let's call it **Import Binary Types**.



Double-click the new button and paste the following code into the body of the handler method. Make sure that the path to the .dll file is correct and that the .dll qualifies for import of binary types (that is, it must not be obfuscated).

```
try
{
    // Create a new document
    UModelDocument = UModel.NewDocument();
    // Instantiate the Import Binary Types dialog
    UModelLib.ImportBinaryTypesDlg dlg = UModel.Dialogs.ImportBinaryTypesDlg;
    // Set the .NET runtime version according to your environment (must be greater than
v2.0) or use "any"
    dlg.Runtime = "any";
    // Set the import language (C# 8.0, in this case)
    dlg.Language = UModelLib.ENUMCodeLangVersion.eCodeLang_CSharp_8_0;
    // No need to show the dialog since we want to do this programmatically
    dlg.ShowDialog = false;
    // Add a new binary type entry to be imported
    UModelLib.IBinaryTypeEntry entry = dlg.CSharp_BinaryTypes.AddItem();
    // Specify the .dll to import (make sure to adjust the path)
    entry.Entry = "C:\\Path\\To\\My.dll";
    // All types shall be imported from this .dll
    entry.ImportTypes = true;
```

 © 2019-2025 Altova GmbH

```
    // The .dll is an executable
    entry.Executeable = true;
    // Perform the actual import
    UModelDocument.ImportBinaryTypes(dlg);
}
catch (Exception ex)
{
    MessageBox.Show(ex.Message);
}
```

Importing all types

The code above essentially creates a new UModel project, sets the import options in the "Import Binary Types" dialog box, and performs the actual import of binary types.

To run the C# code and import binary types:

1. Press **F5** to build and run the Visual Studio solution.
2. On the Windows form that appears, click **Start UModel**, and be patient while the UModel application loads.
3. Only after UModel has finished loading, click **Import Binary Types**, and observe the outcome in the Messages window of UModel.

If you would like to import only specific types, set the `ImportTypes` property is **false**, and supply the types to be imported as arguments to the `TypesToImport` method. The list of distinct types can be separated by comma, semi-colon, or space characters, as illustrated in the code listing below.

```
try
{
    UModelDocument = UModel.NewDocument();
    UModelLib.ImportBinaryTypesDlg dlg = UModel.Dialogs.ImportBinaryTypesDlg;
    dlg.ShowDialog = false;
    dlg.CSharp_BinaryTypes.RemoveAllItems();
    UModelLib.IBinaryTypeEntry entry = dlg.CSharp_BinaryTypes.AddItem();
    entry.Entry = "C:\\Path\\To\\My.dll";
    entry.ImportTypes = false;
    entry.Executeable = true;
    // import only specific types:
    entry.TypesToImport = "MyNamespace.Class1; MyNamespace.Class2";
    UModelDocument.ImportBinaryTypes(dlg);
}
catch (Exception ex)
{
    MessageBox.Show(ex.Message);
}
```

Importing distinct types

16.3.4.3 "Set Styles" Sample

The following sample sets multiple styles for selected diagram elements (if style is available and not already set). The sample uses both the UModel API and the UModel IDE Plug-In library and is available in the following file: **..\UModelExamples\IDEPlugIn\Styles\Styles.cs**.

The solution also includes two setup projects (in .vdproj format, for 32-bit and 64-bit platforms). The setup installs all necessary files, and registers the IDE plug-in for COM and UModel on your target system, so that the plug-in is automatically loaded when UModel is started the next time.

Notes:

- To build and run the sample, the same requirements as for other UModel IDE Plug-ins apply, see Build and Run the Plug-In 741.
- Visual Studio setup projects are not supported starting with Visual Studio 2012 and require a separate extension to be opened. See the information messages displayed by the Visual Studio migration wizard for more details.

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Windows.Forms;
using UModelLib;
using UModelPlugInLib;

/*
 * Styles sample
 * set following styles for selected diagram elements
 *      Fill Color
 *      Header Gradient Begin Color
 *      Header Gradient End Color
 * if style is available and not already set
 */

namespace Styles
{
    public class UModelStyles : UModelPlugInLib.IUModelPlugIn
    {
        bool m_bPlugInVersionOK = true; // verify if UModel-API has been changed in a way
that a recompile of this plug-in is recommended

        #region helpers

        protected string GetPlugInPath()
        {
            string sDLLPath = System.Reflection.Assembly.GetExecutingAssembly().Location;
            return System.IO.Path.GetDirectoryName(sDLLPath);
        }

        #endregion

        #region IUModelPlugIn Members

        public string GetDescription()
        {
```


© 2019-2025 Altova GmbH

```
            return "Styles sample Plug-in for UModel;This Plug-in demonstrates how to
change several styles of the selected diagram elements.";
        }

        public string GetUIModifications()
        {
            try
            {
                string sPath = GetPlugInPath();

                System.IO.StreamReader myFile = new System.IO.StreamReader(sPath + "\
\config.xml");
                string sRet = myFile.ReadToEnd();
                myFile.Close();

                // this replaces the token "**path**" from the XML file with
                // the actual installation path of the plug-in to get the image file
                return sRet.Replace("**path**", sPath);
            }
            catch (System.Exception ex)
            {
                MessageBox.Show("Error in GetUIModifications:" + ex.Message);
                throw ex;
            }
        }

        public void OnInitialize(object pUModel)
        {
            // before processing DDE or batch commands
        }

        public void OnRunning(object pUModel)
        {
            // DDE or batch commands are processed; application is fully initialized

            // verify if UModel-API has been changed in a way that a recompile of this
plug-in is recommended:
            IApplication iApp = (IApplication)pUModel;
            if (iApp == null || iApp.APIMajorVersion != 5) // this plug-in was compiled
for API major version '5'!
            {
                MessageBox.Show("'Styles': This Plug-in has been made with a previous
version of the UModel-API and should be recompiled.\nDisabled Plug-in commands in the
meantime.");
                m_bPlugInVersionOK = false;
            }
        }

        public void OnShutdown(object pUModel)
        {
            // application will shutdown; release all unused objects
            GC.Collect();
        }

        public UModelUpdateAction OnUpdateCommand(int nID, object pUModel)
        {
            UModelUpdateAction action = UModelUpdateAction.UModelUpdateAction_Disable;
            if (!m_bPlugInVersionOK)
                return action;
```

```
            // check for "fill red"
            if (nID == 3 || nID == 6)
                action = OnUpdateSetStyles((IApplication)pUModel);

            // check for "fill green"
            if (nID == 4 || nID == 7)
                action = OnUpdateSetStyles((IApplication)pUModel);

            // release unused objects
            GC.Collect();

            return action;
        }

        public void OnCommand(int nID, object pUModel)
        {
            if (!m_bPlugInVersionOK)
                return;

            // fill red
            if (nID == 3 || nID == 6)
                OnSetStyles((IApplication)pUModel, "red");

            // fill green
            if (nID == 4 || nID == 7)
                OnSetStyles((IApplication)pUModel, "green");

            // release unused objects
            GC.Collect();
        }

        #endregion

        #region SetStyles // set styles of selected diagram elements

        UModelUpdateAction OnUpdateSetStyles(IApplication pUModel)
        {
            if (pUModel == null)
                return UModelUpdateAction.UModelUpdateAction_Disable;

            // get the active document of the application
            IDocument iDoc = pUModel.ActiveDocument;
            if (iDoc == null)
                return UModelUpdateAction.UModelUpdateAction_Disable;

            // get the active diagram window
            IDiagramWindow iActiveDiagram = iDoc.ActiveDiagramWindow;
            if ( iActiveDiagram == null )
                return UModelUpdateAction.UModelUpdateAction_Disable;

            // get the selected elements on the active diagram
            IUMLDataList iSelection = iActiveDiagram.SelectedGuiElements;
            if ( iSelection == null )
                return UModelUpdateAction.UModelUpdateAction_Disable;

            // search all selected elements, if at least one has one of the styles to
change
            foreach ( IUMLGuiElement iSelGuiElement in iSelection )
```


© 2019-2025 Altova GmbH

```
            {
                // verify if it is a GuiVisibleElement (with Styles) and if it may be
modified
                if ( iSelGuiElement is IUMLGuiVisibleElement &&
iSelGuiElement.IsEditable )
                {
                    IUMLGuiVisibleElement iVisGuiElement = (IUMLGuiVisibleElement)
iSelGuiElement;

                    if
( iVisGuiElement.Styles.GetStyle(ENUMUMLGuiStyleKind.eUMLGuiStyle_FillColor) != null ||
                        iVisGuiElement.Styles.GetStyle(ENUMUMLGuiStyleKind.eUMLGuiStyle_
HeaderGradientBeginColor) != null ||
                        iVisGuiElement.Styles.GetStyle(ENUMUMLGuiStyleKind.eUMLGuiStyle_
HeaderGradientEndColor) != null )
                    {
                        return UModelUpdateAction.UModelUpdateAction_Enable;
                    }
                }
            }

            // nothing found => disable command
            return UModelUpdateAction.UModelUpdateAction_Disable;
        }

        public void OnSetStyles(IApplication pUModel, string sColor)
        {
            if (pUModel == null)
                return;

            // get the active document of the application
            IDocument iDoc = pUModel.ActiveDocument;
            if (iDoc == null)
                return;

            // get the active diagram window
            IDiagramWindow iActiveDiagram = iDoc.ActiveDiagramWindow;
            if (iActiveDiagram == null)
                return;

            // get the selected elements on the active diagram
            IUMLDataList iSelection = iActiveDiagram.SelectedGuiElements;
            if (iSelection == null)
                return;

            try
            {
                // make all modifications within one UndoStep; start modification here
                if (!iDoc.BeginModification())
                    return;

                // search all selected elements, and change the style if the wanted value
is not already used (directly applied or through style-chain)
                foreach (IUMLGuiElement iSelGuiElement in iSelection)
                {
                    // verify if it is a GuiVisibleElement (with Styles) and if it may be
modified
                     if (iSelGuiElement is IUMLGuiVisibleElement &&
iSelGuiElement.IsEditable )
```

```
                    {
                        IUMLGuiVisibleElement iVisGuiElement = (IUMLGuiVisibleElement)
iSelGuiElement;

                        // set Fill Color if possible and not already set
                        IUMLGuiStyle iStyle =
iVisGuiElement.Styles.GetStyle(ENUMUMLGuiStyleKind.eUMLGuiStyle_FillColor);
                        if (iStyle != null && iStyle.UsedValue != sColor)
                            iStyle.Value = sColor;

                        // set Header Gradient Begin Color if possible and not already
set
                        iStyle =
iVisGuiElement.Styles.GetStyle(ENUMUMLGuiStyleKind.eUMLGuiStyle_HeaderGradientBeginColor)
;
                        if (iStyle != null && iStyle.UsedValue != sColor)
                            iStyle.Value = sColor;

                        // set Header Gradient End Color if possible and not already set
                        iStyle =
iVisGuiElement.Styles.GetStyle(ENUMUMLGuiStyleKind.eUMLGuiStyle_HeaderGradientEndColor);
                        if (iStyle != null && iStyle.UsedValue != sColor)
                            iStyle.Value = sColor;
                    }
                }

                // do not forget to end modification and finish UndoStep
                iDoc.EndModification();
            }
            catch( System.Exception )
            {
                // rollback made changes
                iDoc.AbortModification();

                // add error handling
            }
        }

        #endregion
    }
}
```

16.3.4.4 "C# Delegate" Sample

The following sample inserts a new C# delegate at the top/left corner of the active diagram window (if this diagram is inside a C# namespace root). The sample uses both the UModel API and the UModel IDE Plug-In library and is available in the following file: **..\UModelExamples\IDEPlugIn\CSharpDelegate\UModelCSharpDelegate.cs**.

To build and run the sample, the same requirements as for other UModel IDE Plug-ins apply, see Build and Run the Plug-In 741.

```
using System;
```

 © 2019-2025 Altova GmbH

```
using System.Collections.Generic;
using System.Text;
using UModelLib;
using UModelPlugInLib;

/*
 * CSharp delegate sample
 * add a new CSharp delegate on the top/left corner of the active class diagram if
possible
 * (i.e. when diagram is inside a C# root namespace)
 */

namespace CSharpDelegate
{
    public class UModelCSharpDelegate : UModelPlugInLib.IUModelPlugIn
    {
        #region helpers

        protected string GetPlugInPath()
        {
            string sDLLPath = System.Reflection.Assembly.GetExecutingAssembly().Location;
            return System.IO.Path.GetDirectoryName(sDLLPath);
        }

        #endregion

        #region IUModelPlugIn Members

        public string GetDescription()
        {
            return "CSharpDelegate sample Plug-in for UModel;This Plug-in demonstrates
how to create a new CSharp delegate on a class diagram.";
        }

        public string GetUIModifications()
        {
            try
            {
                string sPath = GetPlugInPath();

                System.IO.StreamReader myFile = new System.IO.StreamReader(sPath + "\
\config.xml");
                string sRet = myFile.ReadToEnd();
                myFile.Close();

                // this replaces the token "**path**" from the XML file with
                // the actual installation path of the plug-in to get the image file
                return sRet.Replace("**path**", sPath);
            }
            catch (System.Exception ex)
            {
                System.Windows.Forms.MessageBox.Show("Error in GetUIModifications:" +
ex.Message);
                throw ex;
            }
        }


        public void OnInitialize(object pUModel)
```

```
        {
            // before processing DDE or batch commands
        }

        public void OnRunning(object pUModel)
        {
            // DDE or batch commands are processed; application is fully initialized
        }

        public void OnShutdown(object pUModel)
        {
            // application will shutdown; release all unused objects
            GC.Collect();
        }

        public UModelUpdateAction OnUpdateCommand(int nID, object pUModel)
        {
            UModelUpdateAction action = UModelUpdateAction.UModelUpdateAction_Disable;

            // check if we can add a new CSharpDelegate on the active diagram
            if (nID == 3 || nID == 4)
                action = OnUpdateAddNewCSharpDelegate((IApplication)pUModel);

            // release unused objects
            GC.Collect();

            return action;
        }

        public void OnCommand(int nID, object pUModel)
        {
            // add a new CSharpDelegate on the active diagram
            if (nID == 3 || nID == 4)
                OnAddNewCSharpDelegate((IApplication)pUModel);

            // release unused objects
            GC.Collect();
        }

        #endregion

        #region AddNewCSharpDelegate // add new CSharp delegate on active diagram

        UModelUpdateAction OnUpdateAddNewCSharpDelegate(IApplication pUModel)
        {
            if (pUModel == null)
                return UModelUpdateAction.UModelUpdateAction_Disable;

            // get the active document of the application
            IDocument iDoc = pUModel.ActiveDocument;
            if (iDoc == null)
                return UModelUpdateAction.UModelUpdateAction_Disable;

            // get the active diagram window
            IDiagramWindow iActiveDiagram = iDoc.ActiveDiagramWindow;
            if ( iActiveDiagram == null )
                return UModelUpdateAction.UModelUpdateAction_Disable;

            // get the UML diagram of the diagram window
```


© 2019-2025 Altova GmbH

```
            IUMLGuiDiagram iUMLDiagram = iActiveDiagram.Diagram;

            // check if it is a class diagram
            if ( !( iUMLDiagram is IUMLGuiClassDiagram) )
                return UModelUpdateAction.UModelUpdateAction_Disable;

            // verify if the diagram may be modified
            if ( !iUMLDiagram.IsEditable )
                return UModelUpdateAction.UModelUpdateAction_Disable;

            // get the UML element, which "owns" the class diagram
            IUMLElement iDiagramOwner = iUMLDiagram.LinkedOwner;
            if (iDiagramOwner == null)
                return UModelUpdateAction.UModelUpdateAction_Disable;

            // verify if we are inside a CSharp namespace root (otherwise adding a CSharp
delegate makes no sense)
            IUMLElement iFindNamespaceRoot = iDiagramOwner;
            while( iFindNamespaceRoot != null)
            {
                if ( iFindNamespaceRoot is IUMLPackage)
                {
                    IUMLPackage iPackage = (IUMLPackage) iFindNamespaceRoot;
                    if
( iPackage.IsCodeLangNamespaceRoot( ENUMCodeLang.eCodeLang_CSharp ) )
                        return UModelUpdateAction.UModelUpdateAction_Enable;
                }

                iFindNamespaceRoot = iFindNamespaceRoot.Owner;
            }

            // nothing found => disable command
            return UModelUpdateAction.UModelUpdateAction_Disable;
        }

        public void OnAddNewCSharpDelegate(IApplication pUModel)
        {
            if (pUModel == null)
                return;

            // get the active document of the application
            IDocument iDoc = pUModel.ActiveDocument;
            if (iDoc == null)
                return;

            // get the active diagram window
            IDiagramWindow iActiveDiagram = iDoc.ActiveDiagramWindow;
            if (iActiveDiagram == null)
                return;

            // get the UML diagram of the diagram window
            IUMLGuiDiagram iUMLDiagram = iActiveDiagram.Diagram;

            // get the CSharp profile
            IUMLProfile iCSharpProfile = (IUMLProfile)
iDoc.RootPackage.FindPredefinedOwnedElement(ENUMUMLPredefinedElement.ePredefined_CSharp_P
rofile, false);
            if ( iCSharpProfile == null)
                return;
```

```
            try
            {
                // make all modifications within one UndoStep; start modification here
                if (!iDoc.BeginModification())
                    return;

                // get top left corner of the visible diagram area
                int nInsertPosX = iActiveDiagram.ScrollPosX;
                int nInsertPosY = iActiveDiagram.ScrollPosY;

                // add new class on diagram
                IUMLGuiNodeLink iClassNode = iUMLDiagram.AddUMLElement("Class",
nInsertPosX + 100, nInsertPosY + 100);

                IUMLClass iClass = (IUMLClass) iClassNode.Element;
                // use SetName (instead of Name) that UModel automatically generates a
valid, unique name starting with "NewDelegate"
                iClass.SetName("NewDelegate");

                // set the CSharp 'delegate' stereotype
                iClass.ApplyPredefinedStereotype(
 ENUMUMLPredefinedElement.ePredefined_CSharp_delegateStereotypeOfClass );

                // set attribute-section "STAThread"
                IUMLStereotypeApplication iStereotypeApp =
iClass.ApplyPredefinedStereotype(ENUMUMLPredefinedElement.ePredefined_CSharp_attributesSt
ereotypeOfClass);
                IUMLEnumerationLiteral iSTAThread = (IUMLEnumerationLiteral)
iCSharpProfile.FindPredefinedOwnedElement(ENUMUMLPredefinedElement.ePredefined_CSharp_Att
ributePresetsEnumeration_STAThreadEnumerationLiteral, true);
                iStereotypeApp.SetPredefinedTaggedValueAt(-1,
ENUMUMLPredefinedElement.ePredefined_CSharp_attributesStereotypeOfClass_sectionsProperty,
 iSTAThread.Name);

                // add delegate operation:
                IUMLOperation iOperation = iClass.InsertOwnedOperationAt(-1);
                iOperation.SetName( "delegate");

                // per default set operation-return type "void"
                IUMLPrimitiveType iTypeVoid = (IUMLPrimitiveType)
iCSharpProfile.FindPredefinedOwnedElement(ENUMUMLPredefinedElement.ePredefined_CSharp_voi
dPrimitiveType, true);
                iOperation.Type = iTypeVoid;

                // do not forget to end modification and finish UndoStep
                iDoc.EndModification();

                // at last focus newly inserted delegate on the diagram:
                iActiveDiagram.SelectGuiElement(iClassNode, true);
            }
            catch( System.Exception )
            {
                // rollback made changes
                iDoc.AbortModification();

                // add error handling
            }
        }
```


© 2019-2025 Altova GmbH

```
        #endregion
    }
}
```

16.3.4.5 "Set Prefix" Sample

The following sample automatically sets a prefix when new attributes or enumeration literals are added to your UModel project. The sample uses both the UModel API and the UModel IDE Plug-In library and is available in the following file: **..\UModelExamples\IDEPlugIn\DefaultPrefix\DefaultPrefix.cs**.

To build and run the sample, the same requirements as for other UModel IDE Plug-ins apply, see Build and Run the Plug-In 741.

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Diagnostics;
using System.Runtime.InteropServices.ComTypes;
using UModelLib;
using UModelPlugInLib;

/*
 * DefaultPrefix sample
 * listen for newly added UML data and
 * set the prefix of properties ('m_') and EnumerationLiterals ('k_')
 * if the corresponding option is turned on
 */

namespace DefaultPrefix
{
    /* UModelDefaultPrefix is the main class of this plugin and implements
UModelPlugInLib.IUModelPlugIn
     * it is also responsible for attaching/detaching UModelApplicationEvents to/from
UModels IApplication interface
     * and implements the handling of turning on/off the whole "SetPrefix" functionality
     */
    public class UModelDefaultPrefix : UModelPlugInLib.IUModelPlugIn
    {
        // variable which defines whether "SetPrefix" functionality is turned on or off
        bool m_bSetPrefix = true;

        // reference to UModelApplicationEvents; is only used when "SetPrefix"
functionality is turned on (to reduce overhead in the other case)
        UModelApplicationEvents m_AppEvents = null;

        #region helpers

        protected string GetPlugInPath()
        {
            string sDLLPath = System.Reflection.Assembly.GetExecutingAssembly().Location;
            return System.IO.Path.GetDirectoryName(sDLLPath);
        }
```

```
        #endregion

        // create UModelApplicationEvents and attach it to IApplication
        protected void AttachAppEvents( IApplication iUModelApp )
        {
            if (m_AppEvents == null && iUModelApp != null)
            {
                m_AppEvents = new UModelApplicationEvents();
                m_AppEvents.Attach(iUModelApp);
            }
        }

        // detach UModelApplicationEvents;
        protected void DetachAppEvents()
        {
            if (m_AppEvents != null)
            {
                m_AppEvents.Detach();
                m_AppEvents = null;
            }
        }

        #region IUModelPlugIn Members

        public string GetDescription()
        {
            return "DefaultPrefix sample Plug-in for UModel;This Plug-in demonstrates how
to attach to several callback interfaces and how to add a prefix to newly inserted
elements.";
        }

        public string GetUIModifications()
        {
            try
            {
                string sPath = GetPlugInPath();

                System.IO.StreamReader myFile = new System.IO.StreamReader(sPath + "\
\config.xml");
                string sRet = myFile.ReadToEnd();
                myFile.Close();

                // this replaces the token "**path**" from the XML file with
                // the actual installation path of the plug-in to get the image file
                return sRet.Replace("**path**", sPath);
            }
            catch (System.Exception ex)
            {
                System.Windows.Forms.MessageBox.Show("Error in GetUIModifications:" +
ex.Message);
                throw ex;
            }
        }

        public void OnInitialize(object pUModel)
        {
            // before processing DDE or batch commands
        }
```


© 2019-2025 Altova GmbH

```
        public void OnRunning(object pUModel)
        {
            // DDE or batch commands are processed; application is fully initialized
            // and we can attach UModelApplicationEvents
            AttachAppEvents( (IApplication)pUModel );
        }

        public void OnShutdown(object pUModel)
        {
            // detach UModelApplicationEvents; stop receiving events
            DetachAppEvents();

            // application will shutdown; release all unused objects
            GC.Collect();
        }

        public UModelUpdateAction OnUpdateCommand(int nID, object pUModel)
        {
            UModelUpdateAction action = UModelUpdateAction.UModelUpdateAction_Disable;

            // check if automatically setting the prefix is turned on:
            if (nID == 3 || nID == 4)
            {
                action = UModelUpdateAction.UModelUpdateAction_Enable;

                if (m_bSetPrefix)
                    action |= UModelUpdateAction.UModelUpdateAction_Check;
            }

            // release unused objects
            //GC.Collect(); not necessary since we do not access objects here

            return action;
        }

        public void OnCommand(int nID, object pUModel)
        {
            // toggle automatically setting the prefix:
            if (nID == 3 || nID == 4)
                m_bSetPrefix = !m_bSetPrefix;

            // attach UModelApplicationEvents when "SetPrefix" functionality is turned
on; detach otherwise
            if (m_bSetPrefix)
                AttachAppEvents((IApplication)pUModel);
            else
                DetachAppEvents();

            // release unused objects
            GC.Collect();
        }

        #endregion
    }


    /* UModelApplicationEvents is an eventhandler to receive _IApplicationEvents
     * that we know when UModel documents are opened or closed
```

```
     * and that we can Attach/Detach UModelDataEvents
     * We are interested in all _IApplicationEvents and use a connectionpoint to connect
to all these events
     */
    public class UModelApplicationEvents : UModelLib._IApplicationEvents
    {
        // connection point to _IApplicationEvents
        System.Runtime.InteropServices.ComTypes.IConnectionPoint m_cpApplicationEvents =
null;
        // connection cookie
        int m_nApplicationtEventsCookie = 0;
        // we always hold a reference to UModelDataEvents
        UModelDataEvents m_UMLDataEvents = new UModelDataEvents();

        public void Attach(IApplication iApp)
        {
            if (m_cpApplicationEvents == null && iApp != null)
            {
                // find connection point of _IApplicationEvents
                IConnectionPointContainer icpc = (IConnectionPointContainer)iApp;
                Guid IID = typeof(UModelLib._IApplicationEvents).GUID;
                icpc.FindConnectionPoint(ref IID, out m_cpApplicationEvents);

                // advise UModelApplicationEvents as sink for _IApplicationEvents
                m_cpApplicationEvents.Advise(this, out m_nApplicationtEventsCookie);

                // also attach UModelDataEvents to the current document and start
receiving events there
                m_UMLDataEvents.Attach(iApp.ActiveDocument);
            }
        }
        public void Detach()
        {
            if (m_cpApplicationEvents != null)
            {
                // also detach UModelDataEvents and stop receiving events there
                m_UMLDataEvents.Detach();

                // terminate established connection to _IApplicationEvents
                m_cpApplicationEvents.Unadvise(m_nApplicationtEventsCookie);
                m_cpApplicationEvents = null;
            }
        }

        #region _IApplicationEvents Members
        public void OnNewDocument(Document ipDocument)
        {
            Debug.WriteLine("UModelApplicationEvents.OnNewDocument " + ipDocument.Name);
            // a new document has been created in UModel => (re-)connect UModelDataEvents
            m_UMLDataEvents.Attach(ipDocument);
        }

        public void OnDocumentOpened(Document ipDocument)
        {
            Debug.WriteLine("UModelApplicationEvents.OnDocumentOpened " +
ipDocument.Name);
            // a document has been opened in UModel => (re-)connect UModelDataEvents
            m_UMLDataEvents.Attach(ipDocument);
        }
```


© 2019-2025 Altova GmbH

```
        public void OnDocumentClosed(Document ipDocument)
        {
            Debug.WriteLine("UModelApplicationEvents.OnDocumentClosed " +
ipDocument.Name);
            // document has been closed in UModel => disconnect UModelDataEvents
            m_UMLDataEvents.Detach();
        }

        public void OnShutdown()
        {
            Debug.WriteLine("UModelApplicationEvents.OnShutdown");
        }

        #endregion
    }


    /* UModelDataEvents is an eventhandler to receive _IUMLDataEvents
     * from the root-package and all its children.
     * We are only interested in 'OnAfterAddChild' events, so we use a delegate to
connect to this event.
     */
    public class UModelDataEvents : UModelLib._IUMLDataEvents
    {
        // hold a reference to the current UML Root package; this is safe as long as we
listen to when it is deleted
        protected UMLData m_RootPackage = null;

        // attach this eventhandler to the root-package of the (current) document
        public void Attach(IDocument iDoc)
        {
            if (m_RootPackage == null && iDoc != null && iDoc.RootPackage != null)
            {
                // hold a reference to the current UML Root package
                m_RootPackage = (UMLData)iDoc.RootPackage;

                // ensure we get 'OnAfterAddChild' events for *any* added child of the
rootpackage
                // (added to the root-package or one of its children)
                m_RootPackage.EventFilter = (int)
ENUMUMLDataEventFilter.eUMLDataEvent_AddChildOrGrandChild;
                // ensure we get informed when m_RootPackage (and only itself; we do not
care about its chidlren) is deleted
                m_RootPackage.EventFilter |= (int)
ENUMUMLDataEventFilter.eUMLDataEvent_EraseData;

                // we are only interested in 'OnAfterAddChild' and 'OnBeforeErase' events
so use and connect the delegates
                m_RootPackage.OnAfterAddChild += new
_IUMLDataEvents_OnAfterAddChildEventHandler(OnAfterAddChild);
                m_RootPackage.OnBeforeErase += new
_IUMLDataEvents_OnBeforeEraseEventHandler(OnBeforeErase);
            }
        }

        // detach eventhandler from the current UML Root package
        public void Detach()
        {
```

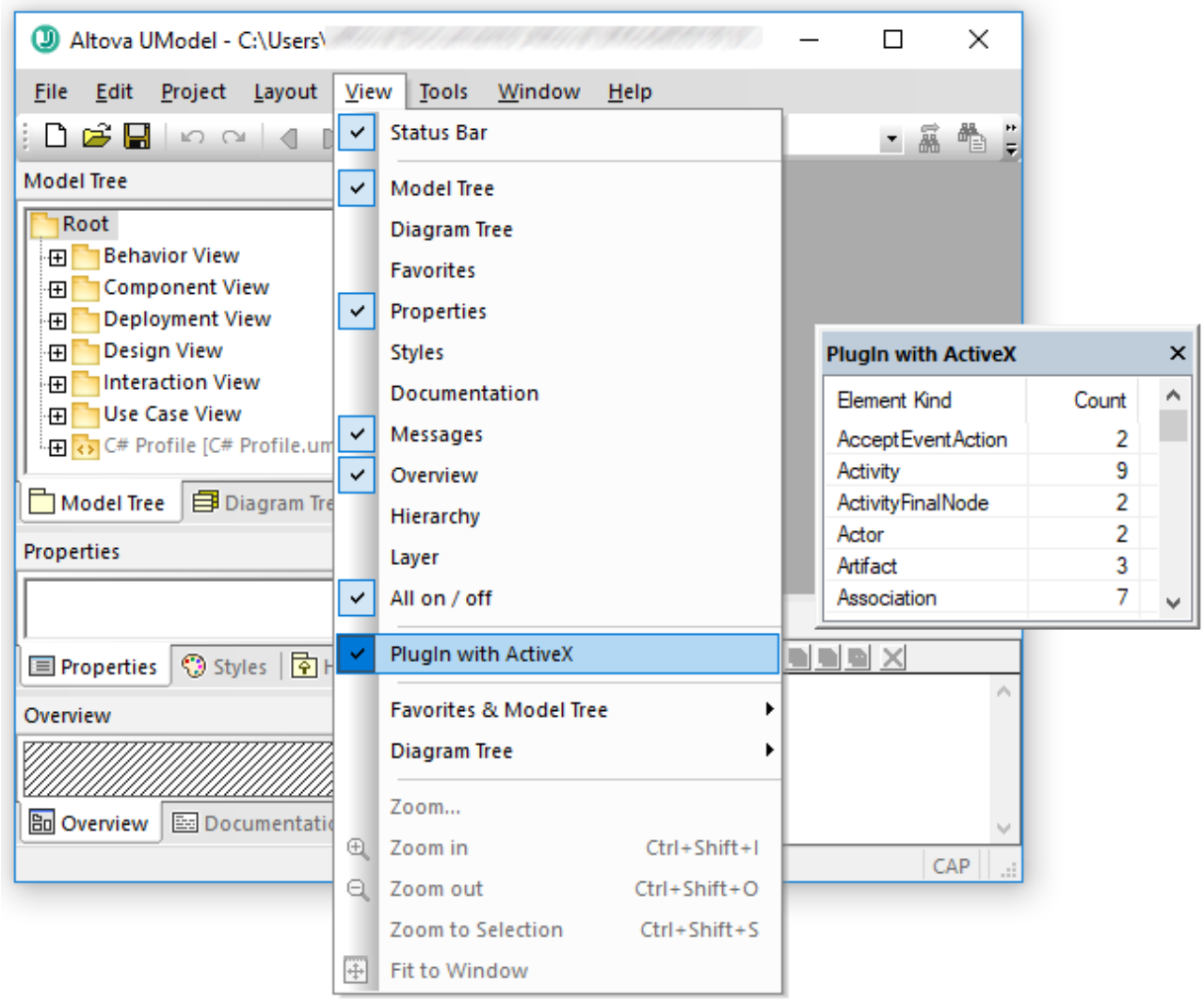
```
            if (m_RootPackage != null)
            {
                m_RootPackage.OnAfterAddChild -= OnAfterAddChild;
                m_RootPackage.OnBeforeErase -= OnBeforeErase;
                m_RootPackage = null;

                // release unused objects
                GC.Collect();
            }
        }

        #region _IUMLDataEvents Members

        public void OnAfterAddChild(IUMLData ipUMLParent, IUMLData ipUMLChild)
        {
            if (ipUMLParent == null || ipUMLChild == null)
                return;

            Debug.WriteLine("UModelDataEvents.OnAfterAddChild " + GetName(ipUMLChild) + "
to " + GetName(ipUMLParent));

           // verify if newly added child is of interesting kind:
            bool bIsEnumerationLiteral = (ipUMLChild is IUMLEnumerationLiteral);
            bool bIsProperty = (ipUMLChild is IUMLProperty);

            if (bIsProperty || bIsEnumerationLiteral)
            {
                try
                {
                    // check if child was added by undo/redo
                    // (we are not allowed to modify anything during Undo/Redo !!)
                    IDocument iDoc = (IDocument)ipUMLChild.Parent;
                    if (!iDoc.IsInUndoRedo)
                    {
                        // we only make one single modification here
                        // no need to use iDoc.BeginModification / iDoc.EndModification
in this case

                        // get the wanted prefix for the element kind
                        string sPrefix = null;

                        if (bIsProperty)
                            sPrefix = "m_";
                        if (bIsEnumerationLiteral)
                            sPrefix = "k_";

                        IUMLNamedElement iNamedChild = (IUMLNamedElement)ipUMLChild;

                        // set prefix only if not already set:
                        if (sPrefix != null && !iNamedChild.Name.StartsWith(sPrefix))
                        {
                            // use SetName (instead of Name) that UModel automatically
generates a valid, unique name starting with 'sPrefix + iNamedChild.Name'
                            iNamedChild.SetName(sPrefix + iNamedChild.Name);
                        }
                    }
                }
                catch (System.Exception e)
                {
```


© 2019-2025 Altova GmbH

```
                    Debug.WriteLine("EXCEPTION: " + e.Message);
                }
            }

            // release unused objects
            GC.Collect();
        }

        public void OnBeforeErase(IUMLData ipUMLData)
        {
            if (ipUMLData != null && m_RootPackage != null &&
ipUMLData.IsSameUMLData((IUMLData)m_RootPackage)) // should always be
            {
                // Detach ourself, since the UML data of m_RootPackage has been deleted
in UModel and we may not access it anymore
                Detach();
            }
        }

        public void OnChanged(IUMLData ipUMLData, string strHint)
        {
            // unused
        }

        public void OnMoveData(IUMLData ipUMLParent, IUMLData ipUMLChild, bool bAttach)
        {
            // unused
        }

        #endregion

        protected string GetName(IUMLData iUMLData)
        {
            if (iUMLData is IUMLNamedElement)
                return ((IUMLNamedElement)iUMLData).Name;

            return "";
        }
    }
}
```

16.3.4.6 "Statistics" Sample

The "Statistics" sample listens for data modifications and counts elements of different element kinds. The sample uses both the UModel API and the UModel IDE Plug-In library. Since the plug-in derives from `System.Windows.Forms.UserControl`, it also acts as an ActiveX control and the results can be shown in a custom window inside UModel:



This code is available in the following file: **..\UModelExamples\IDEPlugIn\StatisticsActiveX\StatisticsActiveX.cs**.

To build and run the sample, the same requirements as for other UModel IDE Plug-ins apply, see Build and Run the Plug-In 741.

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Text;
using System.Drawing;
using System.Runtime.InteropServices.ComTypes;
using System.Windows.Forms;
using UModelLib;
using UModelPlugInLib;
```


© 2019-2025 Altova GmbH

```
/*
 * StatisticsActiveX sample
 * listen for data modifications and count the elements of the different element kinds
 * show the result in a listview of an ActiveX control
 */
namespace StatisticsActiveX
{
    /* StatisticsActiveX is the main class of this plugin and implements
UModelPlugInLib.IUModelPlugIn
     * it is also responsible for attaching/detaching _IApplicationEvents and
_ITransactionEvents
     */
    public partial class StatisticsActiveX : UserControl,
                                             IUModelPlugIn,
                                            _IApplicationEvents,
                                            _ITransactionEvents
    {
        // a sorted dictionary to count the different element kinds
        private Statistics m_Statistics;
        // reference to the transaction notifier of a UModel document
        private TransactionNotifier m_TransactionNotifier;
        // connection point to _IApplicationEvents
        private IConnectionPoint m_cpApplicationEvents = null;
        // connection cookie
        int m_nApplicationtEventsCookie = 0;

        public StatisticsActiveX()
        {
            InitializeComponent();
        }

        #region IUModelPlugIn Members

        public string GetDescription()
        {
            return "PlugIn with ActiveX;This Plug-in demonstrates how to show an ActiveX
control inside UModel.";
        }

        public string GetUIModifications()
        {
            // We don't add any menu or toolbar modifications.
            return "<ConfigurationData><Modifications/></ConfigurationData>";
        }

        public void OnInitialize(object pUModel)
        {
            // before processing DDE or batch commands
        }

        public void OnRunning(object pUModel)
        {
            // DDE or batch commands are processed; application is fully initialized
            // and we can attach to get _IApplicationEvents

            IApplication iApp = (IApplication)pUModel;

            if (m_cpApplicationEvents == null && iApp != null)
```

```
            {
                // find connection point of _IApplicationEvents
                IConnectionPointContainer icpc = (IConnectionPointContainer)iApp;
                Guid IID = typeof(UModelLib._IApplicationEvents).GUID;
                icpc.FindConnectionPoint(ref IID, out m_cpApplicationEvents);

                // advise UModelApplicationEvents as sink for _IApplicationEvents
                m_cpApplicationEvents.Advise(this, out m_nApplicationtEventsCookie);
            }

            AttachTransactionEvents(iApp.ActiveDocument);
        }

        public void OnShutdown(object pUModel)
        {
            // detach application events; stop receiving events
            DetachTransactionEvents();

            if (m_cpApplicationEvents != null)
            {
                // terminate established connection to _IApplicationEvents
                m_cpApplicationEvents.Unadvise(m_nApplicationtEventsCookie);
                m_cpApplicationEvents = null;
            }

            // application will shutdown; release all unused objects
            GC.Collect();
        }

        public void OnCommand(int nID, object pUModel)
        {
            // unused; we did not add any menu- or toolbar-commands
        }

        public UModelUpdateAction OnUpdateCommand(int nID, object pUModel)
        {
            // unused; we did not add any menu- or toolbar-commands
            return UModelUpdateAction.UModelUpdateAction_Disable;
        }

        #endregion

        private void AttachTransactionEvents(IDocument iDoc)
        {
            if (iDoc != null)
            {
                m_TransactionNotifier = iDoc.TransactionNotifier;
                if (m_TransactionNotifier != null)
                {
                    // we are only interested in 'OnEndDataModification' events so use
and connect the delegate
                    m_TransactionNotifier.OnEndDataModification += new
_ITransactionEvents_OnEndDataModificationEventHandler(OnEndDataModification);
                }
            }

            UpdateStatistics(iDoc);
        }
```


© 2019-2025 Altova GmbH

```
        // detach eventhandler from the transaction notifier
        private void DetachTransactionEvents()
        {
            if (m_TransactionNotifier != null)
            {
                m_TransactionNotifier.OnEndDataModification -= OnEndDataModification;
                m_TransactionNotifier = null;
            }
            UpdateStatistics(null);
        }

        void UpdateStatistics(IDocument iDoc)
        {
            // count current elements
            Statistics statistics = new Statistics();

            if (iDoc != null && iDoc.RootPackage != null)
                CountElements(iDoc.RootPackage, ref statistics);

            // anything changed to last update ?
            if (!statistics.IsEqual(m_Statistics))
            {
                m_Statistics = statistics;
                PopulateListView(m_Statistics);
            }

            // release unused objects
            GC.Collect();
        }

        private void CountElements(IUMLElement iElem, ref Statistics statistics)
        {
            // we only count editable elements
            if (iElem == null || iElem.IsEditable == false)
                return;

            string sKindName = iElem.KindName;

            if (!statistics.ContainsKey(sKindName))
                statistics[sKindName] = 1;
            else
                statistics[sKindName]++;

            foreach (IUMLElement iChild in iElem.OwnedElements)
                CountElements(iChild, ref statistics);
        }

        private void PopulateListView(Statistics statistics)
        {
            listView1.BeginUpdate();

            listView1.Items.Clear();
            foreach (KeyValuePair<string, int> kvp in statistics)
            {
                ListViewItem item = new ListViewItem(kvp.Key);
                item.SubItems.Add(Convert.ToString(kvp.Value));

                listView1.Items.Add(item);
            }
```

```
            listView1.EndUpdate();
        }

        #region _ITransactionEvents Members

        public void OnBeginDataModification(Document ipDocument)
        {
            // begin of transaction
        }

        public void OnEndDataModification(Document ipDocument)
        {
            // end of transaction - update statistics
            if (ipDocument != null && ipDocument.TransactionNotifier ==
m_TransactionNotifier)
                UpdateStatistics(ipDocument);
        }

        #endregion

        #region _IApplicationEvents Members

        public void OnNewDocument(Document ipDocument)
        {
            // a new document has been created in UModel => (re-)connect transaction
events
            AttachTransactionEvents(ipDocument);
        }

        public void OnDocumentOpened(Document ipDocument)
        {
            // a document has been opened in UModel => (re-)connect transaction events
            AttachTransactionEvents(ipDocument);
        }

        public void OnDocumentClosed(Document ipDocument)
        {
            // document has been closed in UModel => disconnect transaction events
            if (ipDocument != null && ipDocument.TransactionNotifier ==
m_TransactionNotifier)
                DetachTransactionEvents();
        }

        public void OnShutdown()
        {
        }

        #endregion


        #region Statistics dictionary

        private class Statistics : SortedDictionary<string, int>
        {
            public bool IsEqual(Statistics other)
            {
                if (other == null)
                    return false;
```


© 2019-2025 Altova GmbH

```
                if (Count != other.Count)
                    return false;

                Enumerator e1 = GetEnumerator();
                Enumerator e2 = other.GetEnumerator();
                while (e1.MoveNext() && e2.MoveNext())
                {
                    if ((e1.Current.Key != e2.Current.Key) ||
                        (e1.Current.Value != e2.Current.Value))
                        return false;
                }

                return true;
            }
        }

        #endregion
    }
}
```

16.3.5 Java API Example

The UModel installation package contains an example Java project, located at **C:\Users\<username>\Documents\Altova\UModel2025\UModelExamples\API**. This folder contains Java examples for the UModel API. You can test it directly from the command line using the batch file `BuildAndRun.bat`, or you can compile and run the example project from within Eclipse. See below for instructions on how to use these procedures.

The Java examples folder contains all the files required to run the example project. These files are listed below:

AltovaAutomation.dll	Java-COM bridge: DLL part
AltovaAutomation.jar	Java-COM bridge: Java library part
UModelAPI.jar	Java classes of the UModel API
RunUModel.java	Java example source code
BuildAndRun.bat	Batch file to compile and run example code from the command line prompt. Expects folder where Java Virtual Machine resides as parameter.
.classpath	Eclipse project helper file
.project	Eclipse project file
UModelAPI_JavaDoc.zip	`Javadoc` file containing help documentation for the Java API
Readme.txt	This file

The example starts up UModel and performs a few operations, including opening and closing documents. When done, UModel stays open. You must close it manually.

Running the example from the command line

To run the example from the command line, open a command prompt window, go to the Java folder of the API Examples folder (*see above for location*), and then type:

```
buildAndRun.bat "<Path-to-the-Java-bin-folder>"
```

The Java binary folder must be that of a JDK 1.7 or later installation on your computer.

Press the **Return** key. The Java source in `RunUModel.java` will be compiled and then executed.

Loading the example in Eclipse

Open Eclipse and use the **File | Import... | General | Existing Projects into Workspace** command to add the Eclipse project file (`.project`) located in the Java folder of the API Examples folder (*see above for location*). The project `RunUModel` will then appear in your Package Explorer or Navigator.

Select the project and then the command **Run as | Java Application** to execute the example.

Note: You can select a class name or method of the Java API and press F1 to get help for that class or method.

Potential issues with instanceof and cast operators

There could be some issues with the `instanceof` and cast operators. For example, the `instanceof` and cast operators do not work when you receive instances of a base class type (e.g., `UMLElement.getOwnedElements()`). Note the following recommendations:

- You should not use the `instanceof` operator and class casts when you receive instances of base classes directly from the API.
- Use a member function instead to determine the type of the element. Then create a new instance of the derived class.

An extract of the code listing from **`RunUModel.java`** will give you an idea about how to work around potential issues with the `instanceof` and cast operators:

```
private static void printUMLTree(UMLData i_data, String tab) throws AutomationException
	{
		// Java's 'instanceof' operator does not work where we receive instances of
a base class type like with 'UMLElement.getOwnedElements()'.
		if (i_data.isKindOf("Package"))
		{
			// the Java cast operator does not work for these objects either.
Instead, create a new instance with the appropriate class type.
			UMLPackage umlPackage = new UMLPackage(i_data);  // (UMLPackage)
i_data

			if (umlPackage.getIsShared())
				System.out.println(tab + "Shared Package " +
umlPackage.getName());
			else
```


© 2019-2025 Altova GmbH

```
                        System.out.println(tab + "Package " + umlPackage.getName());
            }
            else if (i_data.isKindOf("Class"))
            {
                  System.out.println(tab + "Class " + new UMLClass(i_data).getName());
            }

            // recurse
            tab += "    ";
            if (i_data.isKindOf("Element"))
                  for (UMLData elem : new UMLElement(i_data).getOwnedElements())
                        printUMLTree(elem, tab);
      }
```

16.3.6 JScript Examples

This section contains listings of JScript code that demonstrate the following basic functionality:

- Start application 799
- Document Access 800
- Generate documentation 801
- Generate code 802
- Update Documentation 807

Example files

The code listings in this section are available in example files that you can test as is or modify to suit your needs. The JScript example files are located at **C:\Users\<username>\Documents\Altova\UModel2025\UModelExamples\API**.

The example files can be run in one of two ways:

- *From the command line:* Open a command prompt window, change the directory to the path above, and type the name of one of the example scripts (for example, `Start.js`).
- *From Windows Explorer:* In Windows Explorer, browse for the JScript file and double-click it.

The script is executed by Windows Script Host that is packaged with Windows operating system. For more information about Windows Script Host, refer to MSDN documentation (https://msdn.microsoft.com).

16.3.6.1 Start application

The JScript below starts the application and shuts it down. If an instance of the application is already running, the running instance will be returned.

Note: For 32-bit UModel, the registered name, or programmatic identifier (ProgId) of the COM object is `UModel.Application`. For 64-bit UModel, the name is `UModel_x64.Application`.

This code is available in the sample file ..\UModelExamples\API\JScript\Start.js (see also Example Files[799]).

```
// Initialize application's COM object. This will start a new instance of the application
and
// return its main COM object. Depending on COM settings, a the main COM object of an
already
// running application might be returned.
try {   objUModel = WScript.GetObject("", "UModel.Application");   }
catch(err) {}

if( typeof( objUModel ) == "undefined" )
{
   try   {   objUModel = WScript.GetObject("", "UModel_x64.Application")   }
   catch(err)
   {
      WScript.Echo( "Can't access or create UModel.Application" );
      WScript.Quit();
   }
}

// if newly started, the application will start without its UI visible. Set it to
visible.
objUModel.Visible = true;

WScript.Echo(objUModel.Edition + " has successfully started. ");

objUModel.Visible = false; // will shutdown application if it has no more COM connections
//objUModel.Visible = true;   // will keep application running with UI visible
```

16.3.6.2 Document Access

The JScript listing below shows how to open documents, set a document as the active document, iterate through the open documents, and close documents.

This code is available in the sample file ..\UModelExamples\API\JScript\DocumentAccess.js (see also Example Files[799]).

```
// Initialize application's COM object. This will start a new instance of the application
and
// return its main COM object. Depending on COM settings, a the main COM object of an
already
// running application might be returned.
try {   objUModel = WScript.GetObject("", "UModel.Application");   }
catch(err) {}

if( typeof( objUModel ) == "undefined" )
{
   try   {   objUModel = WScript.GetObject("", "UModel_x64.Application")   }
   catch(err)
   {
      WScript.Echo( "Can't access or create UModel.Application" );
      WScript.Quit();
   }
```


© 2019-2025 Altova GmbH

```
}

// if newly started, the application will start without its UI visible. Set it to
visible.
objUModel.Visible = true;

// ************************** code snippet for "Simple Document Access"
**********************

// Locate examples using property PersonalDataDirectory
objDoc = objUModel.OpenDocument(objUModel.PersonalDataDirectory + "\\UModelExamples\
\Bank_MultiLanguage.ump");
// open all diagrams
objDoc.OpenAllDiagrams();

// ************************** code snippet for "Simple Document Access"
**********************

// ************************** code snippet for "Iteration"
***********************************

objName = "";
count = 0;
// go through all open diagrams using a JScript Enumerator
for (var iterDiagrams = new Enumerator(objDoc.DiagramWindows); !iterDiagrams.atEnd();
iterDiagrams.moveNext())
{
   objName += "\t" + ++count + " " + iterDiagrams.item().Name + "\n";
}

WScript.Echo("Opened diagrams: \n" + objName);

// go through all open diagrams using index-based access to the document collection
for (i = objDoc.DiagramWindows.Count; i > 0; i--)
   objDoc.DiagramWindows.Item(i).Close();


// ************************** code snippet for "Iteration"
***********************************

//objUModel.Visible = false;      // will shutdown application if it has no more COM
connections
objUModel.Visible = true;   // will keep application running with UI visible
```

16.3.6.3 Generate Documentation

The JScript listing below shows how to generate documentation for the **Bank_MultiLanguage.ump** file in the UModelExamples folder.

This code is available in the sample file `..\UModelExamples\API\JScript\GenerateDoc.js` (see also Example Files 799).

```
// Initialize application's COM object. This will start a new instance of the application
```

```
and
// return its main COM object. Depending on COM settings, a the main COM object of an
already
// running application might be returned.
try {   objUModel = WScript.GetObject("", "UModel.Application");   }
catch(err) {}

if( typeof( objUModel ) == "undefined" )
{
   try   {   objUModel = WScript.GetObject("", "UModel_x64.Application")   }
   catch(err)
   {
      WScript.Echo( "Can't access or create UModel.Application" );
      WScript.Quit();
   }
}

// if newly started, the application will start without its UI visible. Set it to
visible.
objUModel.Visible = true;

// Locate examples via USERPROFILE shell variable.
objWshShell = WScript.CreateObject("WScript.Shell");
majorVersionYear = objUModel.MajorVersion + 1998
strExamplesFolder = objWshShell.ExpandEnvironmentStrings("%USERPROFILE%") + "\\Documents\
\Altova\\UModel" + majorVersionYear + "\\UModelExamples\\";

objDoc = objUModel.OpenDocument(strExamplesFolder + "Bank_MultiLanguage.ump");

// generate documentation
dlgs = objUModel.Dialogs;
docDlg = dlgs.GenerateDocumentationDlg;
docDlg.OutputFormat = 0; // ENUMDocumentationOutputFormat.eDocumentationOutputFormat_HTML

var myObject = new ActiveXObject("Scripting.FileSystemObject");
strDocOutputFolder = strExamplesFolder + "GeneraredDocFromJScriptExample\\";

if (!myObject.FolderExists(strDocOutputFolder))
   myObject.CreateFolder(strDocOutputFolder);

strResultFile = strDocOutputFolder + "Bank_MultiLanguage.html";
objDoc.generateDocumentation(docDlg, strResultFile);

//objUModel.Visible = false;      // will shutdown application if it has no more COM
connections
objUModel.Visible = true;   // will keep application running with UI visible
```

16.3.6.4 Generate Code

The following JScript sample creates a new UModel project, creates some classes and generates code.

This code is available in the sample file `..\UModelExamples\API\JScript\UModelCreateCode.js` (see Example Files 799).

```
// ########################################
// access runing UModel.Application or
```

 © 2019-2025 Altova GmbH

```
// launch new one and access it
// ##########################################


// ##########################################
// CreateCode sample
// shows forward engineering from scratch
// it creates some coding elements in a new UModel project and generates code (saving the
project afterwards)
// ##########################################



// ############ global variables #################
var objUModel = null;
var objWshShell = null;
var objFSO = null;

// ###################### Helpers ##############################

function Exit(strErrorText)
{
   WScript.Echo(strErrorText);

   if (objUModel != null)
      objUModel.Quit();

   WScript.Quit(-1);
}

function CreateGlobalObjects ()
{
   // the Shell and FileSystemObject of the windows scripting host often always useful
   try
   {
      objWshShell = WScript.CreateObject("WScript.Shell");
      objFSO = WScript.CreateObject("Scripting.FileSystemObject");
   }
   catch(err)
   {
      Exit("Can't create WScript.Shell object");
   }

   // create the UModel connection
   // if there is a running instance of UModel (that never had a connection) - use it
   // otherwise, we automatically create a new instance
   try {   objUModel = WScript.GetObject("", "UModel.Application");   }
   catch(err) {}

   if( typeof( objUModel ) == "undefined" )
   {
      try   {   objUModel = WScript.GetObject("", "UModel_x64.Application")   }
      catch(err)
      {
         objUModel = null;
         Exit( "Can't access or create UModel.Application" );
      }
   }
}
```

```
function GetSourceCodeDirectory()
{
   // get directory for source code
   var path =  objUModel.PersonalDataDirectory + "\\UModelExamples\\API\\JScript\
\CreateCode";
   var codeDirectory = objFSO.BuildPath( path, "SampleCode" );
   return codeDirectory;
}

function GetUMPFilePath()
{
   // get file path to save UModel projectfile
   return objUModel.PersonalDataDirectory + "\\UModelExamples\\API\\JScript\\CreateCode\
\CreateCode.ump";
}

function IncludeCSharpProfile( objDocument )
{
   try
   {
      // get dialog for including subprojects:
      var objIncludeSubProjectDialog = objUModel.Dialogs.IncludeSubprojectDlg;

      objIncludeSubProjectDialog.ProjectFile = objUModel.InstallationDirectory + "\
\UModelInclude\\c# Profile.ump";

      return objDocument.IncludeSubproject( objIncludeSubProjectDialog );
   }
   catch(err)
   {
      Exit("Can't include CSharp profile");
   }
}

// ////////////////////// MAIN ////////////////////////////

CreateGlobalObjects();

objUModel.Visible = true;

// open a new, empty document
var objDocument = objUModel.NewDocument();
// get the root-package
var objRootPackage = objDocument.RootPackage;

if ( objDocument    != null &&
     objRootPackage != null &&
    IncludeCSharpProfile( objDocument ) )
{
   // create coding elements
    try
    {
        // make all modifications within one UndoStep; start modification here
      if ( !objDocument.BeginModification() )
         Exit("No modifications allowed");

      // create a namespace root package
      var objCSharpRootNamespace = objRootPackage.InsertPackagedElementAt( -1,
```


© 2019-2025 Altova GmbH

```
"Package" );
      objCSharpRootNamespace.SetName( "CSharp" );

      // find C# Profile...
      var objCSharpProfile = objRootPackage.FindPredefinedOwnedElement( 159, false );//
ePredefined_CSharp_Profile = 159,
      // ...and apply it to the package, which is now a CSharp namespace root
      objCSharpRootNamespace.InsertProfileApplicationAt( -1, objCSharpProfile );


      // create a C# namespace package...
      var objCSharpNamespace = objCSharpRootNamespace.InsertPackagedElementAt( -1,
"Package" );
      objCSharpNamespace.SetName( "Namespace1" );
      // ... and apply the predefined C# namespace stereotype
      objCSharpNamespace.ApplyPredefinedStereotype( 223 ); //
ePredefined_CSharp_namespaceStereotypeOfPackage = 223,

      // create new class within the C# namespace
      var objClass     = objCSharpNamespace.InsertPackagedElementAt( -1, "Class" );
      var objClass2    = objCSharpNamespace.InsertPackagedElementAt( -1, "Class" );
      var objBaseClass = objCSharpNamespace.InsertPackagedElementAt( -1, "Class" );
      objClass    .SetName( "MyClass"     );
      objClass2   .SetName( "MyClass2"    );
      objBaseClass.SetName( "MyBaseClass" );

      // set attribute-section "STAThread"
      var objAttributesStereotypeApplication =
objClass.ApplyPredefinedStereotype( 191 );//
ePredefined_CSharp_attributesStereotypeOfClass = 191
      var objSTAThread = objCSharpProfile.FindPredefinedOwnedElement( 185, true ); //
ePredefined_CSharp_AttributePresetsEnumeration_STAThreadEnumerationLiteral = 185
      objAttributesStereotypeApplication.SetPredefinedTaggedValueAt(-1, 192,
objSTAThread.Name); // ePredefined_CSharp_attributesStereotypeOfClass_sectionsProperty =
192

      // insert new attribute
      var objProperty = objClass.InsertOwnedAttributeAt( -1 );
      objProperty.SetName( "m_Att" );
      objProperty.Visibility = 2;      // eVisibility_Private = 2
      objProperty.Type = objClass2;

      // insert new operation
      var objOperation = objClass.InsertOwnedOperationAt( -1 );
      objOperation.SetName( "GetAtt" );
      objOperation.Type = objClass2;

      // derive MyClass from MyBaseClass
      objClass.InsertGeneralizationAt( -1, objBaseClass );


      // find the component view package
      var objComponentView = objRootPackage.FindPredefinedOwnedElement( 1, false );//
ePredefined_ComponentViewPackage = 1

      // create a new component for C# 3.0 and set the source code directory, where we
want to generate the source code
      var objComponent = objComponentView.InsertPackagedElementAt( -1, "Component" );
      objComponent.CodeLangVersion= 5; //    eCodeLang_CSharp_3_0   = 5,
```

```
      objComponent.CodeProjectFileOrDirectory = GetSourceCodeDirectory();
      objComponent.IsCodeProjectFile = false;

      // this component should realize our classes:
      objComponent.InsertRealizationAt( -1, objClass     );
      objComponent.InsertRealizationAt( -1, objClass2    );
      objComponent.InsertRealizationAt( -1, objBaseClass );

        // do not forget to end modification and finish UndoStep
      objDocument.EndModification();
    }
   catch( err )
   {
        // rollback made changes
      objDocument.AbortModification();
      Exit("Error when creating UML model elements");
   }

   // update code from model
   try
   {
      // explicitely run a syntax check
      if ( objDocument.CheckProjectSyntax() )
      {
         // get dialog for code <=> model synchonizations and set the wanted options:
         var objSynchronizationSettingsDlg =
objUModel.Dialogs.SynchronizationSettingsDlg;

         objSynchronizationSettingsDlg.CodeFromModel_Synchronization = 0; //
eSynchronization_Merge = 0
         objSynchronizationSettingsDlg.CodeFromModel_UserDefinedSPLTemplatesOverrideDefau
lt = true;

         // update code from model
         if ( !objDocument.SynchronizeCodeFromModel( objSynchronizationSettingsDlg ) )
            Exit("Update code from model failed");
      }
      else
         Exit("Syntax check failed");
   }
   catch( err )
   {
      Exit("Error when updating code from model");
   }

   // save project
   objDocument.SaveAs( GetUMPFilePath() );

   WScript.Echo("Finished successfully");
}

// if something went wrong (and we did not save the project),
// we also do not want get asked for saving => set ModifiedFlag to false
if ( objDocument != null )
   objDocument.ModifiedFlag = false;

objUModel.Visible = false;   // will shutdown application if it was started by this
script
```


© 2019-2025 Altova GmbH

16.3.6.5 Update Documentation

The following JScript sample, when running for the first time, reverse engineers all UModel API C# samples found in the **..\UModelExamples\IDEPlugIn** directory and creates HTML and RTF documentation as well as an XMI export of the UModel project. The resulting UMP files, as well as the generated documentation output, are saved to the **..\UModelExamples\API\JScript\UpdateDocumentation** directory. On subsequent runs, it opens the previously generated UModel project files, and creates HTML and RTF documentation, as well as XMI export, provided that something has changed in the UML model.

This code is available in the sample file `..\UModelExamples\API\JScript\UModelUpdateDocumentation.js` (see Example Files 799).

```
// ########################################
// access runing UModel.Application or
// launch new one and access it
// ########################################


// ########################################
// UpdateDocumentation sample
// *) When running the first time (= when no UMP file exists), reverse engineer all C#
UModelAPI samples
//    and create HTML and RTF documentation, make XMI export and save UMP file
// *) when UMP file already exists, open it and synchronize model from code
//    create HTML and RTF documentation and XMI export only if something has been changed
(listen to all different UML data events)
// ########################################

var bRunVisible = true;
var bShowDialogs = bRunVisible && false;

// ////////////// global variables /////////////////
var objUModel   = null;
var objWshShell = null;
var objFSO      = null;

var bChangedAnything = false;
var nAddedClasses   = 0;
var nAddedInterfaces= 0;
var nAddedProperties= 0;
var nAddedOperations= 0;

// ////////////////////// Helpers ///////////////////////////////

function Exit(strErrorText)
{
   WScript.Echo(strErrorText);

   if (objUModel != null)
      objUModel.Quit();

   WScript.Quit(-1);
```

```
}

function CreateGlobalObjects ()
{
   // the Shell and FileSystemObject of the windows scripting host often always useful
   try
   {
      objWshShell = WScript.CreateObject("WScript.Shell");
      objFSO = WScript.CreateObject("Scripting.FileSystemObject");
   }
   catch(err)
   {
      Exit("Can't create WScript.Shell object");
   }

   // create the UModel connection
   // if there is a running instance of UModel (that never had a connection) - use it
   // otherwise, we automatically create a new instance
   try {   objUModel = WScript.GetObject("", "UModel.Application");   }
   catch(err) {}

   if( typeof( objUModel ) == "undefined" )
   {
      try   {   objUModel = WScript.GetObject("", "UModel_x64.Application")   }
      catch(err)
      {
         objUModel = null;
         Exit( "Can't access or create UModel.Application" );
      }
   }
}

// ////////////////////// get different filepathes / ensure folders are
created  ////////////////////////////
function GetScriptPath()
{
   var path = objUModel.PersonalDataDirectory + "\\UModelExamples\\API\\JScript\
\UpdateDocumentation";

   if ( !objFSO.FolderExists( path ) )
      objFSO.CreateFolder( path );

    return path;
}

function GetFilePath( subdir, filename )
{
   var path = objFSO.BuildPath( GetScriptPath(), subdir );

   if ( !objFSO.FolderExists( path ) )
      objFSO.CreateFolder( path );

   return path + "\\" + filename;
}

function GetUMPFilePath ()   { return GetFilePath( "UMP",      "UModelAPI.ump" ); }
function GetXMIFilePath ()   { return GetFilePath( "Output_XMI", "UModelAPI.xmi" ); }
function GetHTMLFilePath()   { return GetFilePath( "Output_HTML","UModelAPI.html"); }
function GetRTFFilePath ()   { return GetFilePath( "Output_RTF", "UModelAPI.rtf" ); }
```


© 2019-2025 Altova GmbH

```
// ////////////////////// UML data event handlers ////////////////////////////
function objRootPackage_OnChanged( objData, strHint )
{
   bChangedAnything = true;
}

// recursively count newly added classes, interfaces, properties and operations
function CountAddedElements( objNewChild )
{
   if ( objNewChild != null )
   {
      if ( objNewChild.KindName == "Class"    ) ++nAddedClasses;
      if ( objNewChild.KindName == "Interface" ) ++nAddedInterfaces;
      if ( objNewChild.KindName == "Property"  ) ++nAddedProperties;
      if ( objNewChild.KindName == "Operation" ) ++nAddedOperations;

      var ownedElements = objNewChild.OwnedElements;
      var itr = new Enumerator( ownedElements );
      for ( ; !itr.atEnd(); itr.moveNext() )
         CountAddedElements( itr.item() );
   }
}

function objRootPackage_OnAfterAddChild( objParent, objNewChild )
{
    bChangedAnything = true;

   // recursively count newly added classes, interfaces, properties and operations
   CountAddedElements( objNewChild );
}

function objRootPackage_OnBeforeErase( objData )
{
    bChangedAnything = true;
}

function objRootPackage_OnMoveData( objParent, objChild, bAttach )
{
    bChangedAnything = true;
}

// ////////////////////// MAIN ////////////////////////////

CreateGlobalObjects();

if ( bRunVisible )
   objUModel.Visible = true;

var objDocument = null;

try
{
   // open document if it exists; create new one otherwise
   var bDocumentExisted = false;

   if ( objFSO.FileExists( GetUMPFilePath() ) )
   {
      objDocument = objUModel.OpenDocument( GetUMPFilePath() );
```

```
      bDocumentExisted = true;
   }
   else
   {
      objDocument = objUModel.NewDocument();
      objDocument.SaveAs( GetUMPFilePath() );
   }

   if ( objDocument == null )
      Exit( "Cannot create or open UModel projectfile" );

   // connect to receive _IUMLDataEvents from the root-package and all its children:
   var objRootPackage = objDocument.RootPackage;
    WScript.ConnectObject (objRootPackage, "objRootPackage_" );

   // ensure we get *all* events from root-package and *all* children:
   objRootPackage.EventFilter = 2 + // eUMLDataEvent_EraseDataOrChild      = 2,
                        8 + // eUMLDataEvent_AddChildOrGrandChild   = 8,
                       32 + // eUMLDataEvent_ChangeDataOrChild      = 32,
                       128; // eUMLDataEvent_MoveChildOrGrandChild   = 128
   if ( bDocumentExisted )
   {
      // UModel projectfile already exists => update model from code

      // get dialog for code <=> model synchonizations and set the wanted options:
      var objSynchronizationSettingsDlg = objUModel.Dialogs.SynchronizationSettingsDlg;
      objSynchronizationSettingsDlg.ShowDialog = bShowDialogs;

      objSynchronizationSettingsDlg.ModelFromCode_Synchronization = 0; //
eSynchronization_Merge = 0

      // update model from code
      if ( !objDocument.SynchronizeModelFromCode( objSynchronizationSettingsDlg ) )
         Exit("Update model from code failed");
   }
   else
   {
      // UModel projectfile did not exist => newly import code into model

      var objImportSourceDirectoryDlg = objUModel.Dialogs.ImportSourceDirectoryDlg;
      objImportSourceDirectoryDlg.ShowDialog = bShowDialogs;

      // set source code directory to import
      objImportSourceDirectoryDlg.Directory = objUModel.PersonalDataDirectory + "\
\UModelExamples\\IDEPlugIn";
      objImportSourceDirectoryDlg.ProcessSubdirectories = true;
      // set source code language to import (C# 3.0)
      objImportSourceDirectoryDlg.Language      = 5; // eCodeLang_CSharp_3_0      = 5
      objImportSourceDirectoryDlg.Synchronization = 0; // eSynchronization_Merge = 0
      // import in a new package
      objImportSourceDirectoryDlg.ImportInNewPackage = true;

      objImportSourceDirectoryDlg.DiagramGeneration = true;

      // content diagram generation settings
      objImportSourceDirectoryDlg.Content_GenerateSingleDiagram            = true;
      objImportSourceDirectoryDlg.Content_GenerateDiagramPerPackage         = true;
      objImportSourceDirectoryDlg.Content_ShowNestedClassifiersSeparately      = false;
      objImportSourceDirectoryDlg.Content_ShowAnonymousBoundElements         = false;
```


© 2019-2025 Altova GmbH

```
      objImportSourceDirectoryDlg.Content_HyperlinkPackagesToDiagrams         = true;
      objImportSourceDirectoryDlg.Content_ShowAttributesCompartment          = true;
      objImportSourceDirectoryDlg.Content_ShowOperationsCompartment          = true;
      objImportSourceDirectoryDlg.Content_ShowNestedClassifiersCompartment   = false;
      objImportSourceDirectoryDlg.Content_ShowEnumerationLiteralsCompartment   = true;
      objImportSourceDirectoryDlg.Content_ShowTaggedValues                = true;
      objImportSourceDirectoryDlg.Content_Autolayout                      = 1; //
eDiagramLayout_Hierarchic = 1
      // open diagrams that autolayout is done:
      objImportSourceDirectoryDlg.Content_OpenDiagrams                   = true;

      // package dependency diagram generation settings (disabled)
      objImportSourceDirectoryDlg.PackageDependency_GenerateDiagram = false;

      // import source directory
      if ( !objDocument.ImportSourceDirectory( objImportSourceDirectoryDlg) )
      {
         // also delete newly created (empty) UMP file that source code directory import
is retried the next time
         objFSO.DeleteFile( GetUMPFilePath() );
         Exit( "Error on importing source directory" );
      }
   }

   // disconnect from getting root-package events
   WScript.DisconnectObject( objRootPackage );
}
catch( err )
{
   // also delete newly created (empty) UMP file that source code directory import is
retried the next time
   objFSO.DeleteFile( GetUMPFilePath() );
   Exit( "Error on importing source directory" );
}

//if something has changed, update the outputs:
if ( bChangedAnything )
{
   try
   {
      // make XMI export for UML2.1.2
      var objIExportXMIFileDlg = objUModel.Dialogs.ExportXMIFileDlg;
      objIExportXMIFileDlg.ShowDialog = bShowDialogs;
      objIExportXMIFileDlg.XMIFile            = GetXMIFilePath();
      objIExportXMIFileDlg.PrettyPrintXMIOutput   = true;
      objIExportXMIFileDlg.ExportUUIDs         = true;
      objIExportXMIFileDlg.ExportExtensions      = true;
      objIExportXMIFileDlg.ExportDiagrams         = true;
      objIExportXMIFileDlg.XMIType            = 1; // eXMI21ForUML212 = 1

      // export to XMI file:
      if ( !objDocument.ExportToXMIFile( objIExportXMIFileDlg ) )
      {
         // error on XMI generation
      }
   }
   catch( err )
   {
      // error on XMI generation
```

```
   }

   try
   {
      var objIDocumentationGenerationDlg = objUModel.Dialogs.GenerateDocumentationDlg;
      objIDocumentationGenerationDlg.ShowDialog = bShowDialogs;

      objIDocumentationGenerationDlg.GenerateLinksToLocalFiles = 1; //
eDocumentationFilePath_RelativeToResultFile   = 1
      objIDocumentationGenerationDlg.SplitOutputToMultipleFiles = true;
      objIDocumentationGenerationDlg.ShowResultFileAfterGeneration = true;

      objIDocumentationGenerationDlg.Details_SelectAll();
      // show up to 10 base class/interface hierarchies
      objIDocumentationGenerationDlg.Details_HierarchyDiagramNestingDepthUp = 10;
      // only show directly derived classes/interfaces
      objIDocumentationGenerationDlg.Details_HierarchyDiagramNestingDepthDown = 1;
      // keep hierarchy diagram as small as possible => expand each element only once
      objIDocumentationGenerationDlg.Details_HierarchyDiagramExpandItemsOnlyOnce = true;

      objIDocumentationGenerationDlg.Include_SelectAllDiagrams();
      objIDocumentationGenerationDlg.Include_SelectNoElements();
      objIDocumentationGenerationDlg.Include_Index = true;
      objIDocumentationGenerationDlg.Include_IncludedSubprojects = false;
      objIDocumentationGenerationDlg.Include_NamedElementsOnly = true;
      objIDocumentationGenerationDlg.Include_UnknownExternals = false;

      var objIncludeElements = objIDocumentationGenerationDlg.Include_Elements;
      var itrIncludeElements = new Enumerator( objIncludeElements );
      for ( ; !itrIncludeElements.atEnd(); itrIncludeElements.moveNext() )
      {
         var objElemSel = itrIncludeElements.item();

         if ( objElemSel.KindName == "Class"         ||
              objElemSel.KindName == "Interface"     ||
              objElemSel.KindName == "Enumeration"   ||
              objElemSel.KindName == "Operation"     ||
              objElemSel.KindName == "Package"      )
         {
            objElemSel.Selection = true;
         }
      }

      // generate HTML documentation (with PNG pictures)
      objIDocumentationGenerationDlg.OutputFormat = 0; // eDocumentationOutputFormat_HTML
  = 0
      objIDocumentationGenerationDlg.DiagramImageFormat = 0; // eOutputImageFormat_PNG =
0
      objIDocumentationGenerationDlg.EmbedDiagrams = false;
      if ( !objDocument.GenerateDocumentation( objIDocumentationGenerationDlg,
GetHTMLFilePath() ) )
      {
         // error on HTML documentation generation
      }

      // generate RTF documentation (with embeded EMF pictures)
      objIDocumentationGenerationDlg.ShowDialog = false; // don't show dialog again
      objIDocumentationGenerationDlg.OutputFormat = 2; // eDocumentationOutputFormat_RTF
  = 2
```


© 2019-2025 Altova GmbH

```
      objIDocumentationGenerationDlg.DiagramImageFormat = 1; // eOutputImageFormat_EMF =
1
      objIDocumentationGenerationDlg.EmbedDiagrams = true;
      if ( !objDocument.GenerateDocumentation( objIDocumentationGenerationDlg,
GetRTFFilePath() ) )
      {
         // error on RTF documentation generation
      }
   }
   catch( err )
   {
      // error on documentation generation
   }

   // show the number of newly added classes, interfaces, properties and operations
   if ( bRunVisible )
   {
      WScript.Echo( "Added classes: "       + nAddedClasses   +
                    "\nAdded interfaces: " + nAddedInterfaces +
                    "\nAdded properties: " + nAddedProperties +
                    "\nAdded operations: " + nAddedOperations );
   }
}
else
{
   if ( bRunVisible )
      WScript.Echo( "Nothing has changed" );
}

// always save document (although it's not really necessary when nothing has been
changed)
objDocument.Save();

if ( bRunVisible )
   objUModel.Visible = false;   // will shutdown application if it was started by this
script
```

16.4 UModel API Reference

This documentation section describes the interfaces, operations, enumerations and events of the UModel API[816]. The content is organized into the following sub-sections:

- UModel Plug-ins[814] - Provides reference to interfaces required for integrating your own plug-ins into UModel
- UModel API Interfaces[816] - Provides reference to all interfaces of the UModel API except for UML data interfaces (see the next bullet)
- UMLData Interfaces[903] - Provides references to interfaces at the UML data level. These interfaces are also part of the UModel API but are described separately. They specifically provide access to UML elements in a UModel document.

16.4.1 UModel Plug-Ins

This section provides reference to the API interfaces required for integrating your own plug-ins into UModel. For conceptual information and instructions about creating UModel IDE plug-ins, see UModel IDE Plug-Ins[733].

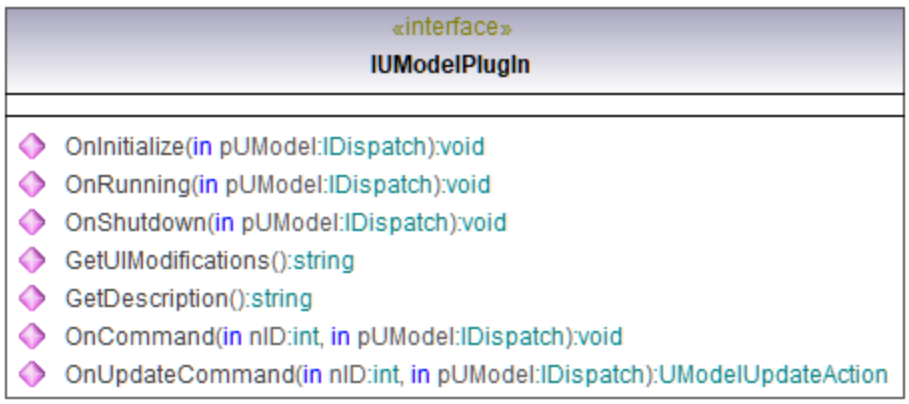
For C# code samples illustrating plug-ins integrated into UModel, see the following topics:

- "Set Styles" Sample[776]
- "C# Delegate" Sample[780]
- "Set Prefix" Sample[785]
- "Statistics" Sample[791]

16.4.1.1 UModelAPI - IUModelPlugIn

Interface **IUModelPlugIn**

diagram



Operation **IUModelPlugIn::GetDescription**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**string**			

Operation **IUModelPlugIn::GetUIModifications**

© 2019-2025 Altova GmbH

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**string**			

Operation **IUModelPlugIn::OnCommand**

parameter	name	direction	type	type modifier	multiplicity	default
	nID	**in**	**int**			
	pUModel	**in**	**IDispatch**			
	return	**return**	**void**			

Operation **IUModelPlugIn::OnInitialize**

parameter	name	direction	type	type modifier	multiplicity	default
	pUModel	**in**	**IDispatch**			
	return	**return**	**void**			

Operation **IUModelPlugIn::OnRunning**

parameter	name	direction	type	type modifier	multiplicity	default
	pUModel	**in**	**IDispatch**			
	return	**return**	**void**			

Operation **IUModelPlugIn::OnShutdown**

parameter	name	direction	type	type modifier	multiplicity	default
	pUModel	**in**	**IDispatch**			
	return	**return**	**void**			

Operation **IUModelPlugIn::OnUpdateCommand**

parameter	name	direction	type	type modifier	multiplicity	default
	nID	**in**	**int**			
	pUModel	**in**	**IDispatch**			
	return	**return**	**UModelUpdateAction** 815			

16.4.1.2 UModelAPI - UModelUpdateAction

Enumeration **UModelUpdateAction**

diagram	
	«enumeration» **UModelUpdateAction** UModelUpdateAction_Enable = 1 UModelUpdateAction_Disable = 2 UModelUpdateAction_Check = 4 UModelUpdateAction_Uncheck = 8

typedElements	Interface IUModelPlugIn 814	Operation OnUpdateCommand 815

16.4.2 UModel API Interfaces

This section provides reference to the objects of the UModel COM API. The objects are described in a generic manner, since the API may be used with virtually any language that supports calling a COM object. For language-specific examples, see:

- Example C# Project 771
- Example Java Project 797
- JScript Examples 799

The API reference contains two main sections, each describing the interfaces and the enumeration types used in the API, respectively. The enumeration values contain both the string name and a numeric value. If your scripting environment does not support enumerations, use the number-values instead.

In .NET, for every interface of the UModel COM automation interface, a .NET class exists with the same name. Also, COM types will be converted to the appropriate .NET type. For example, a type such as `Long` in the COM API would appear as `System.Int32` in .NET.

In Java, note the following syntax variations:

- **Classes and class names**. For every interface of the COM automation interface, a Java class exists with the name of the interface.
- **Method names**. Method names on the Java interface are the same as used on the COM interfaces, but start with a small letter to conform to Java naming conventions. To access COM properties, Java methods that prefix the property name with get and set can be used. If a property does not support write-access, no setter method is available. For example, for the `Name` property of the `Document` interface, the Java methods `getName` and `setName` are available.
- **Enumerations**. For every enumeration defined in the automation interface, a Java enumeration is defined with the same name and values.
- **Events and event handlers**. For every interface in the automation interface that supports events, a Java interface with the same name plus 'Event' is available. To simplify the overloading of single events, a Java class with default implementations for all events is provided. The name of this Java class is the name of the event interface plus 'DefaultHandler'. For example:

```
Application // Java class to access the application
ApplicationEvents // Events interface for the application
ApplicationEventsDefaultHandler // Default handler for "ApplicationEvents"
```

UModel API Errors

The UModel API may return the error codes listed below.

© 2019-2025 Altova GmbH

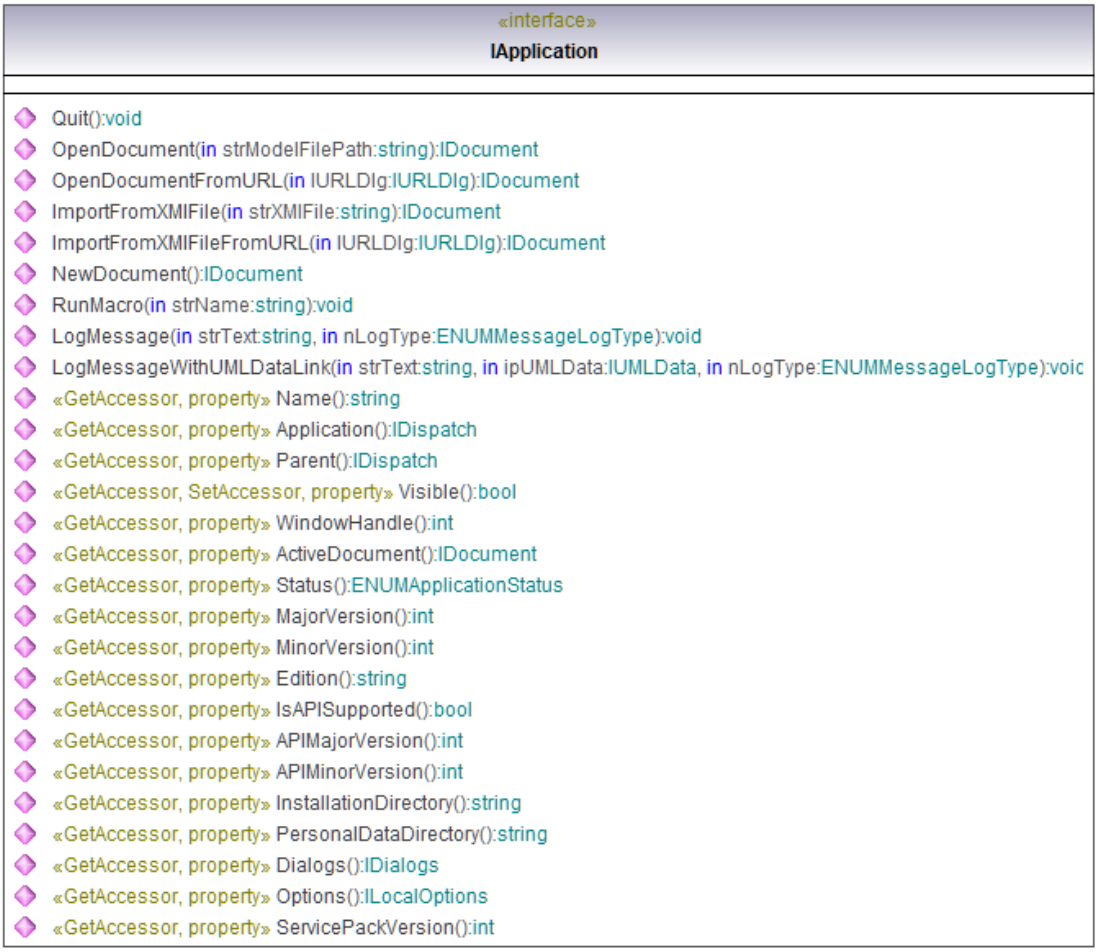
1000	The application object is no longer valid.
1001	Invalid parameter or invalid address for the return parameter was specified.
1002	UModel API is not available in the current edition.
1003	Model Transformations are not supported in the current edition.
1050	Macro not found
1051	Invalid (nested) macro execution
1100	Error when saving file, probably the file name is invalid.
1101	Invalid (duplicate) call to BeginModification.
1102	EndModification called without BeginModification
1200	Error deleting file at URL.
1201	Error creating directory at URL.

The `UMLData` interfaces have specific errors, see UMLData Interfaces 903.

16.4.2.1 UModelAPI - IApplication

Interface **IApplication**

diagram



Operation **IApplication::ActiveDocument**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IDocument** 832			

Operation **IApplication::APIMajorVersion**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**int**			
documentation	A change in the **APIMajorVersion** of the type library (e.g. 1.0 => 2.0) means that non-scripting clients (e.g. IDE PlugIns written in C#, VB.NET, C++,...) should be recompiled.					

Operation **IApplication::APIMinorVersion**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**int**			

© 2019-2025 Altova GmbH

Operation **IApplication::Application**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IDispatch**			

Operation **IApplication::Dialogs**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IDialogs** 830			

Operation **IApplication::Edition**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**string**			

Operation **IApplication::ImportFromXMIFile**

parameter	name	direction	type	type modifier	multiplicity	default
	strXMIFile	**in**	**string**			
	return	**return**	**IDocument** 832			

Operation **IApplication::ImportFromXMIFileFromURL**

parameter	name	direction	type	type modifier	multiplicity	default
	IURLDlg	**in**	**IURLDlg** 890			
	return	**return**	**IDocument** 832			

Operation **IApplication::InstallationDirectory**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**string**			

Operation **IApplication::IsAPISupported**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IApplication::LogMessage**

parameter	name	direction	type	type modifier	multiplicity	default
	strText	**in**	**string**			
	nLogType	**in**	**ENUMMessageLogType**			
	return	**return**	**void**			

Operation **IApplication::LogMessageWithUMLDataLink**

parameter	name	direction	type	type modifier	multiplicity	default
	strText	**in**	**string**			
	ipUMLData	**in**	**IUMLData** 904			
	nLogType	**in**	**ENUMMessageLogType**			
	return	**return**	**void**			

Operation **IApplication::MajorVersion**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**int**			

Operation **IApplication::MinorVersion**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**int**			

Operation **IApplication::Name**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**string**			

Operation **IApplication::NewDocument**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IDocument** 832			

Operation **IApplication::OpenDocument**

parameter	name	direction	type	type modifier	multiplicity	default
	strModelFilePath	**in**	**string**			
	return	**return**	**IDocument** 832			

Operation **IApplication::OpenDocumentFromURL**

parameter	name	direction	type	type modifier	multiplicity	default
	lURLDlg	**in**	**IURLDlg** 890			
	return	**return**	**IDocument** 832			

Operation **IApplication::Options**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**ILocalOptions** 866			

Operation **IApplication::Parent**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IDispatch**			

Operation **IApplication::PersonalDataDirectory**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**string**			

Operation **IApplication::Quit**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**void**			

Operation **IApplication::RunMacro**

parameter	name	direction	type	type modifier	multiplicity	default
	strName	**in**	**string**			
	return	**return**	**void**			

Operation **IApplication::ServicePackVersion**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**int**			

© 2019-2025 Altova GmbH

Operation **IApplication::Status**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	ENUMApplicationStatus 896			

Operation **IApplication::Visible**

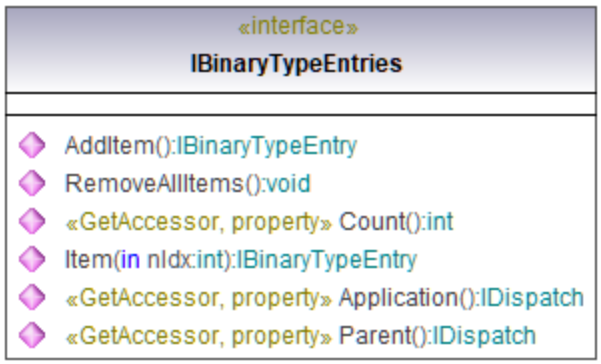
parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IApplication::WindowHandle**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**int**			

16.4.2.2 UModelAPI - IBinaryTypeEntries

Interface **IBinaryTypeEntries**

diagram	
typedElements	Interface IImportBinaryTypesDlg 851 Operation CSharp_BinaryTypes 852, Java_BinaryTypes 853, VBasic_BinaryTypes 854

Operation **IBinaryTypeEntries::AddItem**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IBinaryTypeEntry 822			

Operation **IBinaryTypeEntries::Application**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IDispatch**			

Operation **IBinaryTypeEntries::Count**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**int**			

Operation **IBinaryTypeEntries::Item**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	**IBinaryTypeEntry** 822			

Operation **IBinaryTypeEntries::Parent**

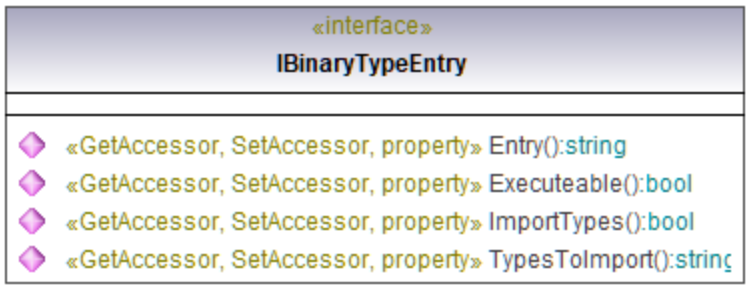
parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IDispatch**			

Operation **IBinaryTypeEntries::RemoveAllItems**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**void**			

16.4.2.3 UModelAPI - IBinaryTypeEntry

Interface **IBinaryTypeEntry**

diagram	
typedElements	Interface **IBinaryTypeEntries** 821 Operation **AddItem** 821 **Item** 822

Operation **IBinaryTypeEntry::Entry**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**string**			

Operation **IBinaryTypeEntry::Executeable**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IBinaryTypeEntry::ImportTypes**

parameter	name	direction	type	type modifier	multiplicity	default

 © 2019-2025 Altova GmbH

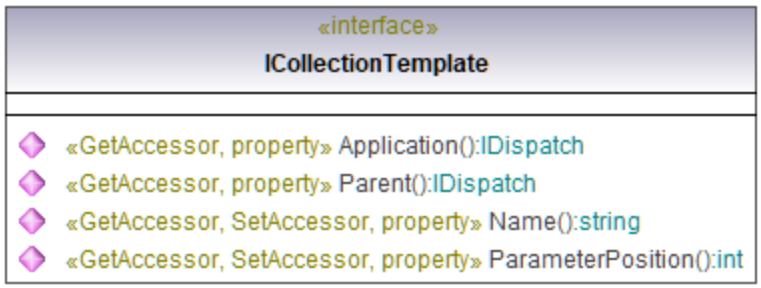
	return	**return**	**bool**

Operation **IBinaryTypeEntry::TypesToImport**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**string**			

16.4.2.4 UModelAPI - ICollectionTemplate

Interface **ICollectionTemplate**

diagram	
typedElements	Interface **ICollectionTemplates** 824 Operation **InsertItemAt** 824 **Item** 824	

Operation **ICollectionTemplate::Application**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IDispatch**			

Operation **ICollectionTemplate::Name**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**string**			

Operation **ICollectionTemplate::ParameterPosition**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**int**			

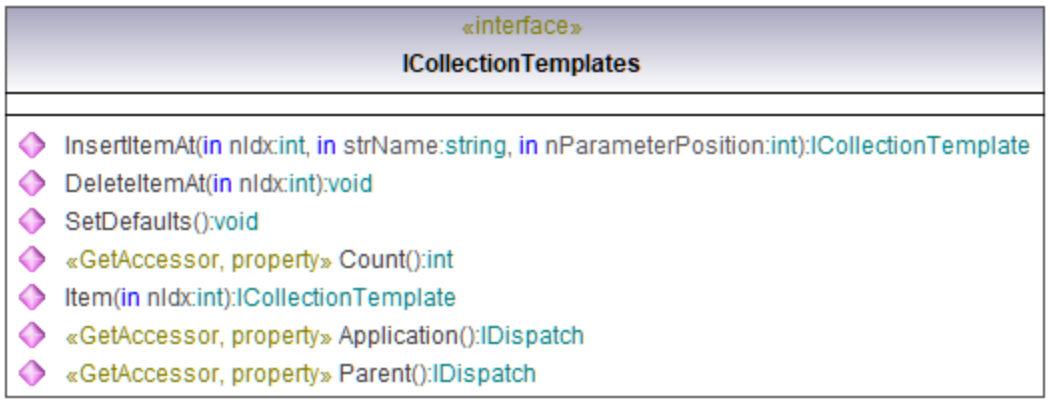
Operation **ICollectionTemplate::Parent**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IDispatch**			

© 2019-2025 Altova GmbH

16.4.2.5 UModelAPI - ICollectionTemplates

Interface **ICollectionTemplates**

diagram	
typedElements	Interface ILocalOptionsDiagramEditing 870 Operation CollectionTemplates_CSharp 871 CollectionTemplates_Java 871 CollectionTemplates_UML 871 CollectionTemplates_VBasic 871

Operation **ICollectionTemplates::Application**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IDispatch**			

Operation **ICollectionTemplates::Count**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**int**			

Operation **ICollectionTemplates::DeleteItemAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	**void**			

Operation **ICollectionTemplates::InsertItemAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	strName	**in**	**string**			
	nParameterPosition	**in**	**int**			
	return	**return**	**ICollectionTemplate** 823			

Operation **ICollectionTemplates::Item**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	**ICollectionTemplate** 823			

© 2019-2025 Altova GmbH

Operation **ICollectionTemplates::Parent**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IDispatch**			

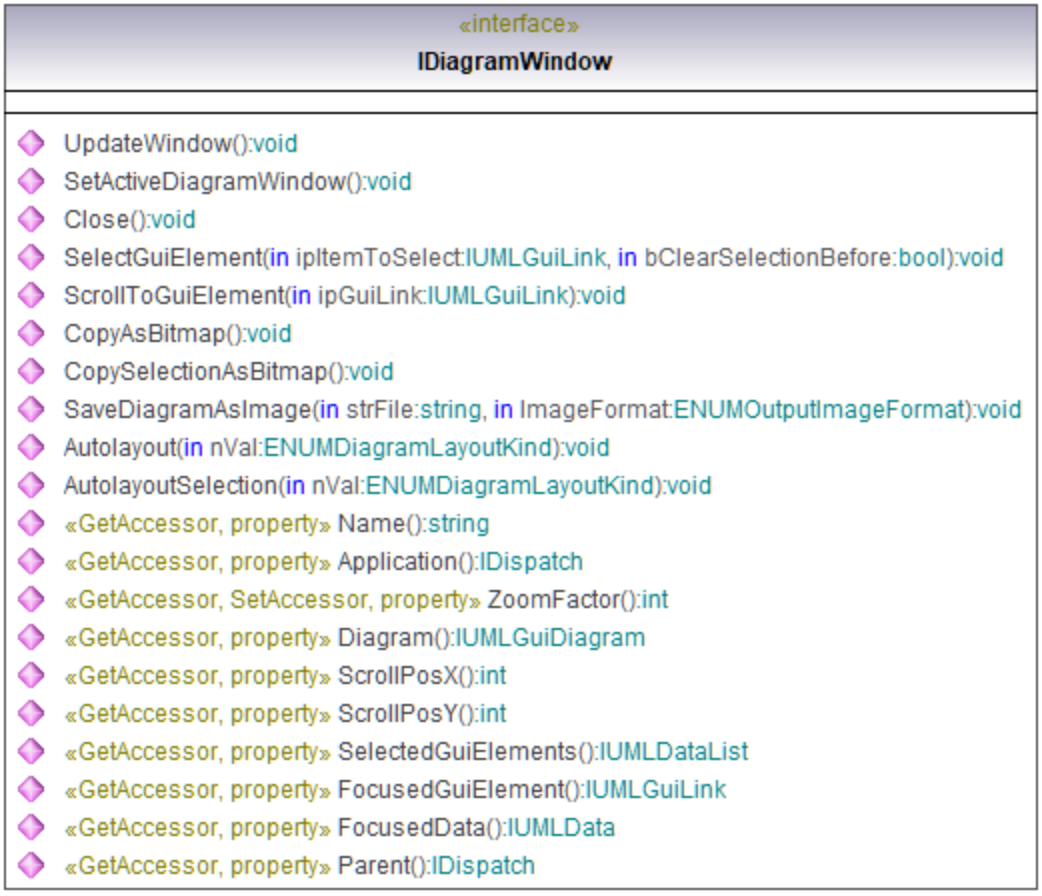
Operation **ICollectionTemplates::SetDefaults**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**void**			

16.4.2.6 UModelAPI - IDiagramWindow

Interface **IDiagramWindow**

diagram



typedElements

Interface **_IDiagramWindowEvents** 892
Interface **_IDocumentEvents** 893

Operation **OnDiagramWindowClosed** 892
Operation **OnActivateDiagramWindow** 893
OnDiagramWindowClosed 893
OnDiagramWindowOpened 893

	Interface IDiagramWindows 828 Interface IDocument 832	Operation Item 828 Operation ActiveDiagramWindow 833 OpenDiagram 837

Operation **IDiagramWindow::Application**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IDispatch**			

Operation **IDiagramWindow::Autolayout**

parameter	name	direction	type	type modifier	multiplicity	default
	nVal	**in**	**ENUMDiagramLayoutKind** 898			
	return	**return**	**void**			

Operation **IDiagramWindow::AutolayoutSelection**

parameter	name	direction	type	type modifier	multiplicity	default
	nVal	**in**	**ENUMDiagramLayoutKind** 898			
	return	**return**	**void**			

Operation **IDiagramWindow::Close**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**void**			

Operation **IDiagramWindow::CopyAsBitmap**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**void**			

Operation **IDiagramWindow::CopySelectionAsBitmap**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**void**			

Operation **IDiagramWindow::Diagram**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLGuiDiagram** 1211			

Operation **IDiagramWindow::FocusedData**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLData** 904			

Operation **IDiagramWindow::FocusedGuiElement**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLGuiLink** 1224			

Operation **IDiagramWindow::Name**

parameter	name	direction	type	type modifier	multiplicity	default

© 2019-2025 Altova GmbH

	return	**return**	**string**

Operation **IDiagramWindow::Parent**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IDispatch**			

Operation **IDiagramWindow::SaveDiagramAsImage**

parameter	name	direction	type	type modifier	multiplicity	default
	strFile	**in**	**string**			
	ImageFormat	**in**	**ENUMOutputImageFormat** 901			
	return	**return**	**void**			

Operation **IDiagramWindow::ScrollPosX**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**int**			

Operation **IDiagramWindow::ScrollPosY**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**int**			

Operation **IDiagramWindow::ScrollToGuiElement**

parameter	name	direction	type	type modifier	multiplicity	default
	ipGuiLink	**in**	**IUMLGuiLink** 1224			
	return	**return**	**void**			

Operation **IDiagramWindow::SelectedGuiElements**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			
documentation	A list of elements of type IUMLGuiElement 1216.					

Operation **IDiagramWindow::SelectGuiElement**

parameter	name	direction	type	type modifier	multiplicity	default
	ipItemToSelect	**in**	**IUMLGuiLink** 1224			
	bClearSelectionBefore	**in**	**bool**			
	return	**return**	**void**			

Operation **IDiagramWindow::SetActiveDiagramWindow**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**void**			

Operation **IDiagramWindow::UpdateWindow**

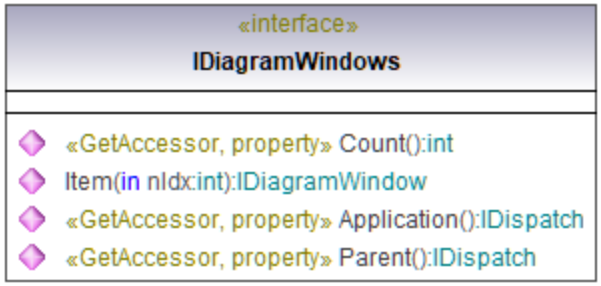
parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**void**			

Operation **IDiagramWindow::ZoomFactor**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**int**			

16.4.2.7 UModelAPI - IDiagramWindows

Interface **IDiagramWindows**

diagram	
typedElements	Interface **IDocument** 832 Operation **DiagramWindows** 833

Operation **IDiagramWindows::Application**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IDispatch**			

Operation **IDiagramWindows::Count**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**int**			

Operation **IDiagramWindows::Item**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	**IDiagramWindow** 825			

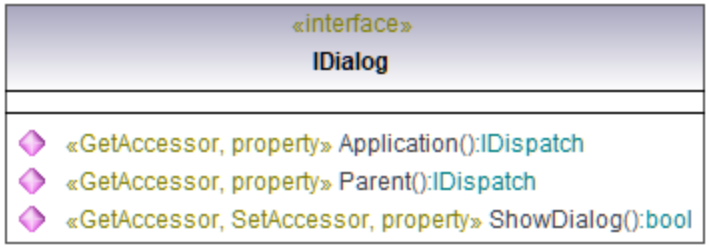
Operation **IDiagramWindows::Parent**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IDispatch**			

 © 2019-2025 Altova GmbH

16.4.2.8 UModelAPI - IDialog

Interface **IDialog**

diagram	
hierarchy	

Operation **IDialog::Application**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IDispatch**			

Operation **IDialog::Parent**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IDispatch**			

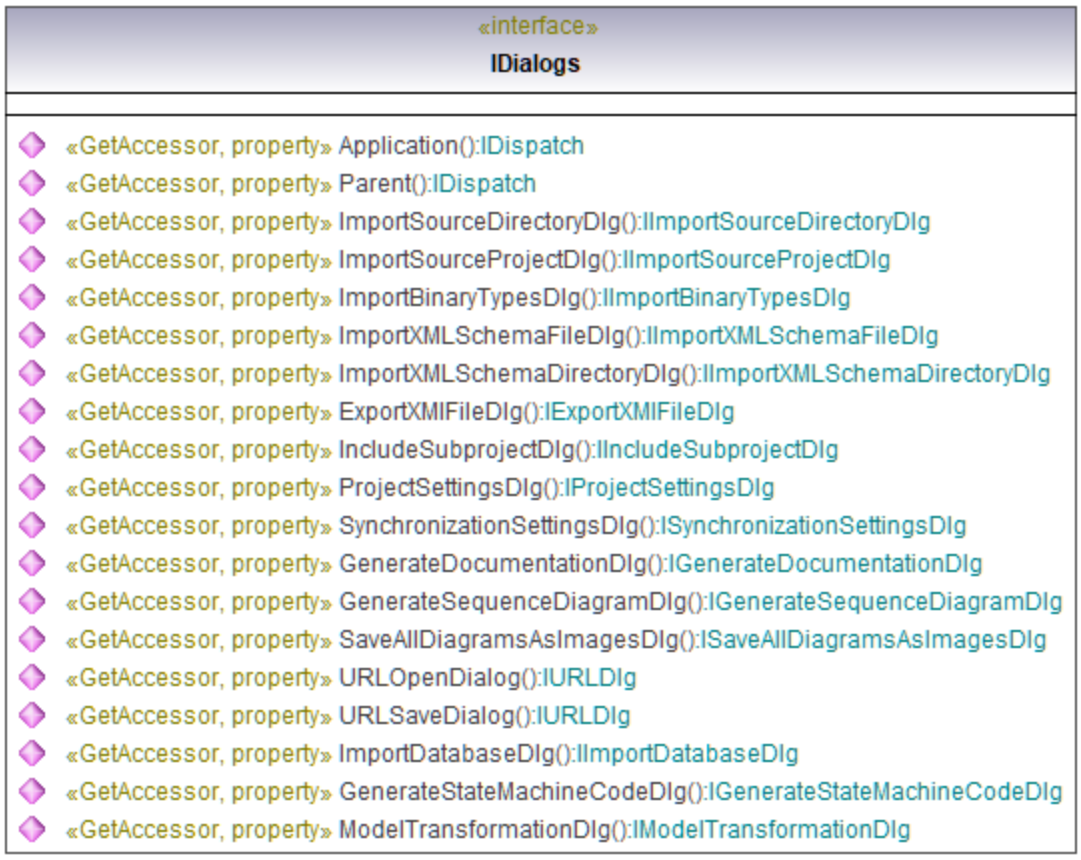
Operation **IDialog::ShowDialog**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

© 2019-2025 Altova GmbH

16.4.2.9 UModelAPI - IDialogs

Interface **IDialogs**

diagram	
typedElements	Interface **IApplication** 818 Operation **Dialogs** 819

Operation **IDialogs::Application**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IDispatch**			

Operation **IDialogs::ExportXMIFileDlg**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IExportXMIFileDlg** 839			

Operation **IDialogs::GenerateDocumentationDlg**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IGenerateDocumentationDlg** 841			

Operation **IDialogs::GenerateSequenceDiagramDlg**

© 2019-2025 Altova GmbH

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IGenerateSequenceDiagramDlg 848			

Operation **IDialogs::GenerateStateMachineCodeDlg**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IGenerateStateMachineCodeDlg 850			

Operation **IDialogs::ImportBinaryTypesDlg**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IImportBinaryTypesDlg 851			

Operation **IDialogs::ImportDatabaseDlg**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IImportDatabaseDlg 855			

Operation **IDialogs::ImportSourceDirectoryDlg**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IImportSourceDirectoryDlg 856			

Operation **IDialogs::ImportSourceProjectDlg**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IImportSourceProjectDlg 860			

Operation **IDialogs::ImportXMLSchemaDirectoryDlg**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IImportXMLSchemaDirectoryDlg 862			

Operation **IDialogs::ImportXMLSchemaFileDlg**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IImportXMLSchemaFileDlg 863			

Operation **IDialogs::IncludeSubprojectDlg**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IIncludeSubprojectDlg 864			

Operation **IDialogs::ModelTransformationDlg**

parameter	name	direction	type	type modifier	multiplicity	default

	return	**return**	IModelTransformationDlg 881

Operation **IDialogs::Parent**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IDispatch**			

Operation **IDialogs::ProjectSettingsDlg**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IProjectSettingsDlg 885			

Operation **IDialogs::SaveAllDiagramsAsImagesDlg**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	ISaveAllDiagramsAsImagesDlg 888			

Operation **IDialogs::SynchronizationSettingsDlg**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	ISynchronizationSettingsDlg 888			

Operation **IDialogs::URLOpenDialog**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IURLDlg 890			

Operation **IDialogs::URLSaveDialog**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IURLDlg 890			

16.4.2.10 UModelAPI - IDocument

Interface **IDocument**

diagram	*The diagram is not included because of page size constraints; however, it is available in the HTML version of the manual (https://www.altova.com/manual/en/umodelprofessional/2025.2/).*
typedElements	Interface _IApplicationEvents 891 Interface _IDocumentEvents 893 Operation OnDocumentClosed 891 OnDocumentOpened 891 OnNewDocument 892 Operation OnAfterReloadDocument 893 OnBeforeReloadDocument 893 OnDocumentClosed 893 OnDocumentSaved 894

© 2019-2025 Altova GmbH

	Interface _ITransactionEvents 895 Interface IApplication 818	Operation OnDocumentSavedAs 894 OnModifiedFlagChanged 894 OnBeginDataModification 895 OnEndDataModification 895 Operation ActiveDocument 818 ImportFromXMIFile 819 ImportFromXMIFileFromURL 819 NewDocument 820 OpenDocument 820 OpenDocumentFromURL 820

Operation **IDocument::AbortModification**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**void**			

Operation **IDocument::ActiveDiagramWindow**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IDiagramWindow 825			

Operation **IDocument::Application**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IDispatch**			

Operation **IDocument::BeginModification**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IDocument::CanFocusUMLDataInModelTree**

parameter	name	direction	type	type modifier	multiplicity	default
	ipUMLData	**in**	IUMLData 904			
	return	**return**	**bool**			

Operation **IDocument::CheckProjectSyntax**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IDocument::DiagramWindows**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IDiagramWindows 828			

Operation **IDocument::ElementFamilyStyles**

parameter	name	direction	type	type modifier	multiplicity	default
	strKind	**in**	**string**			
	return	**return**	IUMLGuiStyles 1241			

Operation **IDocument::EndModification**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**void**			

Operation **IDocument::ExportToXMIFile**

parameter	name	direction	type	type modifier	multiplicity	default
	ipDlg	**in**	**IExportXMIFileDlg** 839			
	return	**return**	**bool**			

Operation **IDocument::FocusedUMLData**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLData** 904			
documentation	Get the focused UML data of the docoument. Normally this is the one which is shown in the "Properties" window.					

Operation **IDocument::FocusedUMLDataNotifier**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IFocusedUMLDataNotifier** 840			

Operation **IDocument::FocusUMLDataInModelTree**

parameter	name	direction	type	type modifier	multiplicity	default
	ipUMLData	**in**	**IUMLData** 904			
	bFocusModelTree	**in**	**bool**			
	bEnsureModelTreeVisible	**in**	**bool**			
	return	**return**	**void**			

Operation **IDocument::FullName**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**string**			

Operation **IDocument::GenerateDocumentation**

parameter	name	direction	type	type modifier	multiplicity	default
	ipDlg	**in**	**IGenerateDocumentationDlg** 841			
	strResultFile	**in**	**string**			
	return	**return**	**bool**			

Operation **IDocument::GenerateSequenceDiagram**

parameter	name	direction	type	type modifier	multiplicity	default
	ipDlg	**in**	**IGenerateSequenceDiagramDlg** 848			
	ipOp	**in**	**IUMLOperation** 1131			
	return	**return**	**bool**			

Operation **IDocument::GenerateSequenceDiagramsForAllOperations**

parameter	name	direction	type	type modifier	multiplicity	default

© 2019-2025 Altova GmbH

	bAllPublicOnly	**in**	**bool**
	bIncludeGettersAndSetters	**in**	**bool**
	return	**return**	**void**

Operation **IDocument::GenerateStateMachineCode**

parameter	name	direction	type	type modifier	multiplicity	default
	ipDlg	**in**	**IGenerateStateMachineCodeDlg** 850			
	ipStateMachine	**in**	**IUMLStateMachine** 1167			
	return	**return**	**bool**			

Operation **IDocument::GuiRoot**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLGuiRootElement** 1234			

Operation **IDocument::ImportBinaryTypes**

parameter	name	direction	type	type modifier	multiplicity	default
	ipDlg	**in**	**IImportBinaryTypesDlg** 851			
	return	**return**	**bool**			

Operation **IDocument::ImportDatabase**

parameter	name	direction	type	type modifier	multiplicity	default
	ipDlg	**in**	**IImportDatabaseDlg** 855			
	return	**return**	**bool**			

Operation **IDocument::ImportSourceDirectory**

parameter	name	direction	type	type modifier	multiplicity	default
	ipDlg	**in**	**IImportSourceDirectoryDlg** 856			
	return	**return**	**bool**			

Operation **IDocument::ImportSourceProject**

parameter	name	direction	type	type modifier	multiplicity	default
	ipDlg	**in**	**IImportSourceProjectDlg** 860			
	return	**return**	**bool**			

Operation **IDocument::ImportXMLSchemaDirectory**

parameter	name	direction	type	type modifier	multiplicity	default
	ipDlg	**in**	**IImportXMLSchemaDirectoryDlg** 862			
	return	**return**	**bool**			

Operation **IDocument::ImportXMLSchemaFile**

parameter	name	direction	type	type modifier	multiplicity	default
	ipDlg	**in**	**IImportXMLSchemaFileDlg** 863			
	return	**return**	**bool**			

Operation **IDocument::IncludeSubproject**

parameter	name	direction	type	type modifier	multiplicity	default
	ipDlg	**in**	**IIncludeSubprojectDlg** 864			
	return	**return**	**bool**			

Operation **IDocument::IsInUndoRedo**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IDocument::IsLoadedFromPreviousFileFormat**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			
documentation	True, when the document has been loaded from a project file with a previous file format version. When saving the document, previous versions of UModel will not be able to load this file anymore.					

Operation **IDocument::LineStyles**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLGuiStyles** 1241			

Operation **IDocument::MergeProject**

parameter	name	direction	type	type modifier	multiplicity	default
	strProjectFile	**in**	**string**			
	return	**return**	**bool**			

Operation **IDocument::MergeProject3Way**

parameter	name	direction	type	type modifier	multiplicity	default
	strProjectFile	**in**	**string**			
	strCommonAncestorFile	**in**	**string**			
	return	**return**	**bool**			

Operation **IDocument::MergeProjectFromURL**

parameter	name	direction	type	type modifier	multiplicity	default
	lURLDlg	**in**	**IURLDlg** 890			
	return	**return**	**bool**			

Operation **IDocument::ModelTransformation**

parameter	name	direction	type	type modifier	multiplicity	default
	ipDlg	**in**	**IModelTransformationDlg** 881			
	return	**return**	**bool**			

© 2019-2025 Altova GmbH

Operation **IDocument::ModifiedFlag**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IDocument::Name**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**string**			

Operation **IDocument::NodeStyles**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLGuiStyles** 1241			

Operation **IDocument::OpenAllDiagrams**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**void**			

Operation **IDocument::OpenDiagram**

parameter	name	direction	type	type modifier	multiplicity	default
	ipUMLDiagram	**in**	**IUMLGuiDiagram** 1211			
	return	**return**	**IDiagramWindow** 825			

Operation **IDocument::Parent**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IDispatch**			

Operation **IDocument::Path**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**string**			

Operation **IDocument::ProjectSettings**

parameter	name	direction	type	type modifier	multiplicity	default
	ipDlg	**in**	**IProjectSettingsDlg** 885			
	return	**return**	**void**			

Operation **IDocument::ProjectStyles**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLGuiStyles** 1241			

Operation **IDocument::Reload**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IDocument::RootPackage**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLPackage** 1134			

Operation **IDocument::Save**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**void**			

Operation **IDocument::SaveAllDiagramsAsImages**

parameter	name	direction	type	type modifier	multiplicity	default
	ipDlg	**in**	**ISaveAllDiagramsAsImagesDlg** 888			
	return	**return**	**bool**			

Operation **IDocument::SaveAs**

parameter	name	direction	type	type modifier	multiplicity	default
	strFileName	**in**	**string**			
	return	**return**	**void**			

Operation **IDocument::SaveCopyAs**

parameter	name	direction	type	type modifier	multiplicity	default
	strFileName	**in**	**string**			
	return	**return**	**void**			

Operation **IDocument::Saved**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IDocument::SaveToURL**

parameter	name	direction	type	type modifier	multiplicity	default
	lURLDlg	**in**	**IURLDlg** 890			
	return	**return**	**void**			

Operation **IDocument::SynchronizationSettings**

parameter	name	direction	type	type modifier	multiplicity	default
	ipDlg	**in**	**ISynchronizationSettingsDlg** 888			
	return	**return**	**void**			

Operation **IDocument::SynchronizeCodeFromModel**

parameter	name	direction	type	type modifier	multiplicity	default
	ipDlg	**in**	**ISynchronizationSettingsDlg** 888			
	return	**return**	**bool**			

Operation **IDocument::SynchronizeModelFromCode**

parameter	name	direction	type	type modifier	multiplicity	default
	ipDlg	**in**	**ISynchronizationSettingsDlg** 888			
	return	**return**	**bool**			

© 2019-2025 Altova GmbH

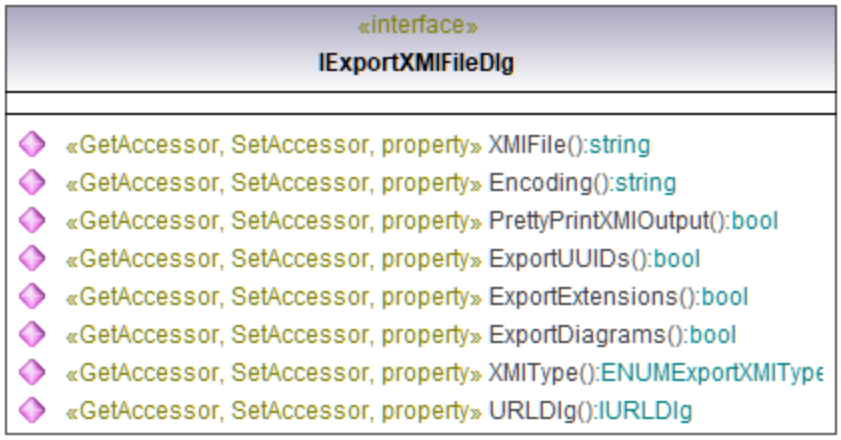
Operation **IDocument::TransactionNotifier**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	ITransactionNotifier 889			

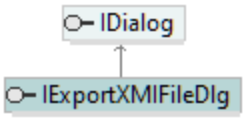
16.4.2.11 UModelAPI - IExportXMIFileDlg

Interface **IExportXMIFileDlg**

diagram



hierarchy



typedElements

Interface IDialogs 830
Interface IDocument 832
Operation ExportXMIFileDlg 830
Operation ExportToXMIFile 834

Operation **IExportXMIFileDlg::Encoding**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**string**			

Operation **IExportXMIFileDlg::ExportDiagrams**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IExportXMIFileDlg::ExportExtensions**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IExportXMIFileDlg::ExportUUIDs**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IExportXMIFileDlg::PrettyPrintXMIOutput**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IExportXMIFileDlg::URLDlg**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IURLDlg** 890			

Operation **IExportXMIFileDlg::XMIFile**

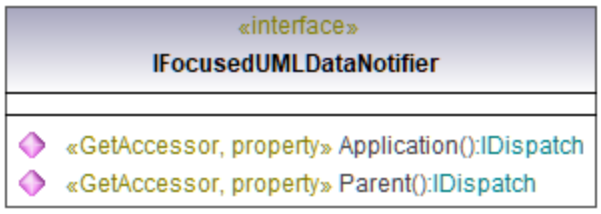
parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**string**			

Operation **IExportXMIFileDlg::XMIType**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**ENUMExportXMIType** 900			

16.4.2.12 UModelAPI - IFocusedUMLDataNotifier

Interface **IFocusedUMLDataNotifier**

diagram	
typedElements	Interface **IDocument** 832 Operation **FocusedUMLDataNotifier** 834

Operation **IFocusedUMLDataNotifier::Application**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IDispatch**			

Operation **IFocusedUMLDataNotifier::Parent**

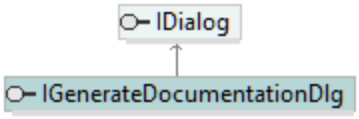
parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IDispatch**			

 © 2019-2025 Altova GmbH

UML documentation generated by UModel UML Editor http://www.altova.com/umodel Wed Sep 25 10:33:04 2024

16.4.2.13 UModelAPI - IGenerateDocumentationDlg

Interface **IGenerateDocumentationDlg**

diagram	*The diagram is not included because of page size constraints; however, it is available in the HTML version of the manual (https://www.altova.com/manual/en/umodelprofessional/2025.2/).*
hierarchy	
typedElements	Interface IDialogs 830 Interface IDocument 832 Operation GenerateDocumentationDlg 830 Operation GenerateDocumentation 834

Operation **IGenerateDocumentationDlg::CreateFolderForDiagrams**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IGenerateDocumentationDlg::Details_Associations**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IGenerateDocumentationDlg::Details_BoundElements**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IGenerateDocumentationDlg::Details_Constraints**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IGenerateDocumentationDlg::Details_Diagram**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IGenerateDocumentationDlg::Details_Documentation**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IGenerateDocumentationDlg::Details_DocumentationAsHTML**

parameter	name	direction	type	type modifier	multiplicity	default

	return	**return**	**bool**			

Operation **IGenerateDocumentationDlg::Details_Generals**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IGenerateDocumentationDlg::Details_HierarchyDiagram**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IGenerateDocumentationDlg::Details_HierarchyDiagramExpandItemsOnlyOnce**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IGenerateDocumentationDlg::Details_HierarchyDiagramNestingDepthDown**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**int**			

Operation **IGenerateDocumentationDlg::Details_HierarchyDiagramNestingDepthUp**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**int**			

Operation **IGenerateDocumentationDlg::Details_Hyperlinks**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IGenerateDocumentationDlg::Details_ImplementedInterfaces**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IGenerateDocumentationDlg::Details_OperationParameters**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IGenerateDocumentationDlg::Details_OwnedDiagrams**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IGenerateDocumentationDlg::Details_OwnedMembers**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IGenerateDocumentationDlg::Details_Owner**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

© 2019-2025 Altova GmbH

Operation **IGenerateDocumentationDlg::Details_Properties**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IGenerateDocumentationDlg::Details_SelectAll**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**void**			

Operation **IGenerateDocumentationDlg::Details_SelectNone**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**void**			

Operation **IGenerateDocumentationDlg::Details_ShownOnDiagrams**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IGenerateDocumentationDlg::Details_SourceTargetOfRelations**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IGenerateDocumentationDlg::Details_Specifics**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IGenerateDocumentationDlg::Details_TemplateParameters**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IGenerateDocumentationDlg::Details_TemplateParameterSubstitutions**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IGenerateDocumentationDlg::Details_TypedElements**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IGenerateDocumentationDlg::DiagramImageFormat**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**ENUMOutputImageFormat** 901			

Operation **IGenerateDocumentationDlg::EmbedCSSinHTML**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IGenerateDocumentationDlg::EmbedDiagrams**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IGenerateDocumentationDlg::Fonts_GetFace**

parameter	name	direction	type	type modifier	multiplicity	default
	nSetting	**in**	**ENUMDocumentationFontSetting** 899			
	return	**return**	**string**			

Operation **IGenerateDocumentationDlg::Fonts_GetSize**

parameter	name	direction	type	type modifier	multiplicity	default
	nSetting	**in**	**ENUMDocumentationFontSetting** 899			
	return	**return**	**int**			

Operation **IGenerateDocumentationDlg::Fonts_GetTextColor**

parameter	name	direction	type	type modifier	multiplicity	default
	nSetting	**in**	**ENUMDocumentationFontSetting** 899			
	return	**return**	**int**			

Operation **IGenerateDocumentationDlg::Fonts_IsBold**

parameter	name	direction	type	type modifier	multiplicity	default
	nSetting	**in**	**ENUMDocumentationFontSetting** 899			
	return	**return**	**bool**			

Operation **IGenerateDocumentationDlg::Fonts_IsItalic**

parameter	name	direction	type	type modifier	multiplicity	default
	nSetting	**in**	**ENUMDocumentationFontSetting** 899			
	return	**return**	**bool**			

Operation **IGenerateDocumentationDlg::Fonts_IsUnderline**

parameter	name	direction	type	type modifier	multiplicity	default
	nSetting	**in**	**ENUMDocumentationFontSetting** 899			
	return	**return**	**bool**			

Operation **IGenerateDocumentationDlg::Fonts_SetBold**

parameter	name	direction	type	type modifier	multiplicity	default
	nSetting	**in**	**ENUMDocumentationFontSetting** 899			
	bVal	**in**	**bool**			
	return	**return**	**void**			

© 2019-2025 Altova GmbH

Operation **IGenerateDocumentationDlg::Fonts_SetDefaults**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**void**			

Operation **IGenerateDocumentationDlg::Fonts_SetFace**

parameter	name	direction	type	type modifier	multiplicity	default
	nSetting	**in**	**ENUMDocumentationFontSetting** 899			
	strNewVal	**in**	**string**			
	return	**return**	**void**			

Operation **IGenerateDocumentationDlg::Fonts_SetItalic**

parameter	name	direction	type	type modifier	multiplicity	default
	nSetting	**in**	**ENUMDocumentationFontSetting** 899			
	bVal	**in**	**bool**			
	return	**return**	**void**			

Operation **IGenerateDocumentationDlg::Fonts_SetSize**

parameter	name	direction	type	type modifier	multiplicity	default
	nSetting	**in**	**ENUMDocumentationFontSetting** 899			
	nNewVal	**in**	**int**			
	return	**return**	**void**			

Operation **IGenerateDocumentationDlg::Fonts_SetTextColor**

parameter	name	direction	type	type modifier	multiplicity	default
	nSetting	**in**	**ENUMDocumentationFontSetting** 899			
	nNewVal	**in**	**int**			
	return	**return**	**void**			

Operation **IGenerateDocumentationDlg::Fonts_SetUnderline**

parameter	name	direction	type	type modifier	multiplicity	default
	nSetting	**in**	**ENUMDocumentationFontSetting** 899			
	bVal	**in**	**bool**			
	return	**return**	**void**			

Operation **IGenerateDocumentationDlg::GenerateLinksToLocalFiles**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**ENUMDocumentationFilePathKind** 899			

Operation **IGenerateDocumentationDlg::Include_Diagrams**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IKindSelectionList** 865			

Operation **IGenerateDocumentationDlg::Include_Elements**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IKindSelectionList** 865			

Operation **IGenerateDocumentationDlg::Include_IncludedPredefinedSubprojects**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IGenerateDocumentationDlg::Include_IncludedSubprojects**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IGenerateDocumentationDlg::Include_Index**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IGenerateDocumentationDlg::Include_NamedElementsOnly**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IGenerateDocumentationDlg::Include_SelectAllDiagrams**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**void**			

Operation **IGenerateDocumentationDlg::Include_SelectAllElements**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**void**			

Operation **IGenerateDocumentationDlg::Include_SelectAllKindsOf**

parameter	name	direction	type	type modifier	multiplicity	default
	strKindName	**in**	**string**			
	bVal	**in**	**bool**			
	return	**return**	**void**			

Operation **IGenerateDocumentationDlg::Include_SelectDefaults**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**void**			

Operation **IGenerateDocumentationDlg::Include_SelectKind**

parameter	name	direction	type	type modifier	multiplicity	default
	strKindName	**in**	**string**			
	bVal	**in**	**bool**			

© 2019-2025 Altova GmbH

	return	**return**	**void**

Operation **IGenerateDocumentationDlg::Include_SelectNoDiagrams**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**void**			

Operation **IGenerateDocumentationDlg::Include_SelectNoElements**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**void**			

Operation **IGenerateDocumentationDlg::Include_UnknownExternals**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IGenerateDocumentationDlg::OutputFormat**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**ENUMDocumentationOutputFormat** 900			

Operation **IGenerateDocumentationDlg::ShowResultFileAfterGeneration**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IGenerateDocumentationDlg::SplitOutputToMultipleFiles**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IGenerateDocumentationDlg::SPSFile**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**string**			

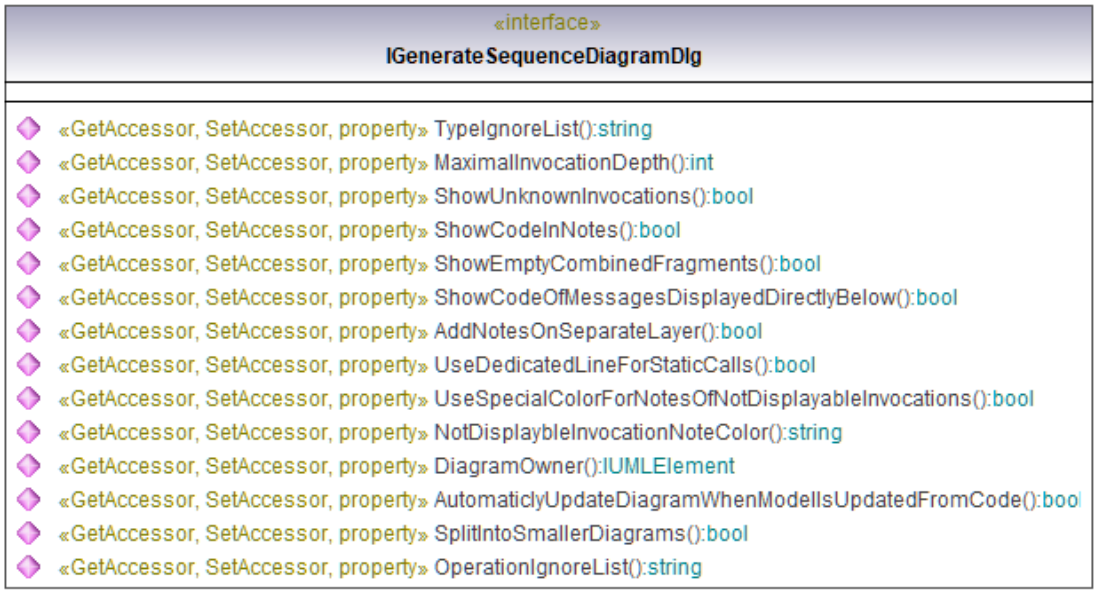
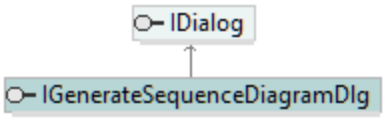
Operation **IGenerateDocumentationDlg::UseFixedDesign**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

© 2019-2025 Altova GmbH

16.4.2.14 UModelAPI - IGenerateSequenceDiagramDlg

Interface **IGenerateSequenceDiagramDlg**

diagram	
hierarchy	
typedElements	Interface IDialogs 830 Interface IDocument 832 Operation GenerateSequenceDiagramDlg 830 Operation GenerateSequenceDiagram 834

Operation **IGenerateSequenceDiagramDlg::AddNotesOnSeparateLayer**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IGenerateSequenceDiagramDlg::AutomaticlyUpdateDiagramWhenModelIsUpdatedFromCode**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IGenerateSequenceDiagramDlg::DiagramOwner**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLElement 1050			

Operation **IGenerateSequenceDiagramDlg::MaximalInvocationDepth**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**int**			

© 2019-2025 Altova GmbH

Operation **IGenerateSequenceDiagramDlg::NotDisplaybleInvocationNoteColor**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**string**			

Operation **IGenerateSequenceDiagramDlg::OperationIgnoreList**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**string**			

Operation **IGenerateSequenceDiagramDlg::ShowCodeInNotes**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IGenerateSequenceDiagramDlg::ShowCodeOfMessagesDisplayedDirectlyBelow**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IGenerateSequenceDiagramDlg::ShowEmptyCombinedFragments**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IGenerateSequenceDiagramDlg::ShowUnknownInvocations**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IGenerateSequenceDiagramDlg::SplitIntoSmallerDiagrams**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IGenerateSequenceDiagramDlg::TypeIgnoreList**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**string**			

Operation **IGenerateSequenceDiagramDlg::UseDedicatedLineForStaticCalls**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

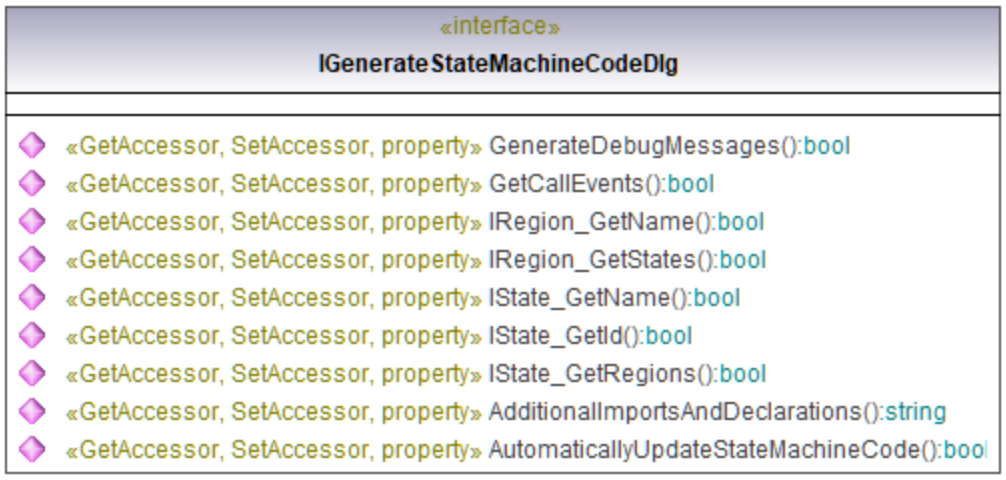
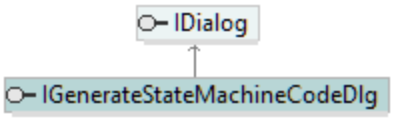
Operation **IGenerateSequenceDiagramDlg::UseSpecialColorForNotesOfNotDisplayableInvocations**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

© 2019-2025 Altova GmbH

16.4.2.15 UModelAPI - IGenerateStateMachineCodeDlg

Interface **IGenerateStateMachineCodeDlg**

diagram	
hierarchy	
typedElements	Interface IDialogs 830 Interface IDocument 832 Operation GenerateStateMachineCodeDlg 831 Operation GenerateStateMachineCode 835

Operation **IGenerateStateMachineCodeDlg::AdditionalImportsAndDeclarations**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**string**			

Operation **IGenerateStateMachineCodeDlg::AutomaticallyUpdateStateMachineCode**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IGenerateStateMachineCodeDlg::GenerateDebugMessages**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IGenerateStateMachineCodeDlg::GetCallEvents**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IGenerateStateMachineCodeDlg::IRegion_GetName**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

© 2019-2025 Altova GmbH

Operation **IGenerateStateMachineCodeDlg::IRegion_GetStates**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IGenerateStateMachineCodeDlg::IState_GetId**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IGenerateStateMachineCodeDlg::IState_GetName**

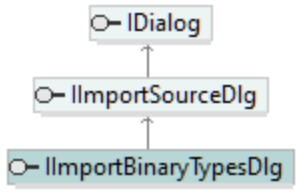
parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IGenerateStateMachineCodeDlg::IState_GetRegions**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

16.4.2.16 UModelAPI - IImportBinaryTypesDlg

Interface **IImportBinaryTypesDlg**

diagram	*The diagram is not included because of page size constraints; however, it is available in the HTML version of the manual (https://www.altova.com/manual/en/umodelprofessional/2025.2/).*
hierarchy	
typedElements	Interface **IDialogs** 830 Interface **IDocument** 832 Operation **ImportBinaryTypesDlg** 831 Operation **ImportBinaryTypes** 835

Operation **IImportBinaryTypesDlg::Content_GenerateDiagramPerPackage**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IImportBinaryTypesDlg::Content_GenerateSingleDiagram**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IImportBinaryTypesDlg::Content_HyperlinkPackagesToDiagrams**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IImportBinaryTypesDlg::Content_ShowAnonymousBoundElements**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IImportBinaryTypesDlg::Content_ShowNestedClassifiersSeparately**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IImportBinaryTypesDlg::CSharp_BinaryTypes**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IBinaryTypeEntries** 821			

Operation **IImportBinaryTypesDlg::CSharp_ImportReferencedTypes**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IImportBinaryTypesDlg::CSharp_ImportReferencedTypesRestriction**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**string**			

Operation **IImportBinaryTypesDlg::CSharp_ImportTypesOnly**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IImportBinaryTypesDlg::CSharp_ImportVisibility**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IImportBinaryTypesDlg::CSharp_ImportVisibilityRestriction**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**string**			

Operation **IImportBinaryTypesDlg::CSharp_OneAttributePerAttributeSection**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IImportBinaryTypesDlg::CSharp_OverridePathForNativeLibraries**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**string**			

Operation **IImportBinaryTypesDlg::CSharp_ReflectionOnly**

© 2019-2025 Altova GmbH

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IImportBinaryTypesDlg::CSharp_SuppressAttributeSections**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IImportBinaryTypesDlg::CSharp_SuppressAttributeSuffix**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IImportBinaryTypesDlg::Java_BinaryTypes**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IBinaryTypeEntries** 821			

Operation **IImportBinaryTypesDlg::Java_ImportReferencedTypes**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IImportBinaryTypesDlg::Java_ImportReferencedTypesRestriction**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**string**			

Operation **IImportBinaryTypesDlg::Java_ImportTypesOnly**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IImportBinaryTypesDlg::Java_ImportVisibility**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IImportBinaryTypesDlg::Java_ImportVisibilityRestriction**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**string**			

Operation **IImportBinaryTypesDlg::Java_OverridePathForNativeLibraries**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**string**			

Operation **IImportBinaryTypesDlg::Java_SuppressAnnotationModifiers**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IImportBinaryTypesDlg::Runtime**

parameter	name	direction	type	type modifier	multiplicity	default

	return	**return**	**string**

Operation **IImportBinaryTypesDlg::VBasic_BinaryTypes**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IBinaryTypeEntries** 821			

Operation **IImportBinaryTypesDlg::VBasic_ImportReferencedTypes**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IImportBinaryTypesDlg::VBasic_ImportReferencedTypesRestriction**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**string**			

Operation **IImportBinaryTypesDlg::VBasic_ImportTypesOnly**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IImportBinaryTypesDlg::VBasic_ImportVisibility**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IImportBinaryTypesDlg::VBasic_ImportVisibilityRestriction**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**string**			

Operation **IImportBinaryTypesDlg::VBasic_OneAttributePerAttributeSection**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IImportBinaryTypesDlg::VBasic_OverridePathForNativeLibraries**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**string**			

Operation **IImportBinaryTypesDlg::VBasic_ReflectionOnly**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IImportBinaryTypesDlg::VBasic_SuppressAttributeSections**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IImportBinaryTypesDlg::VBasic_SuppressAttributeSuffix**

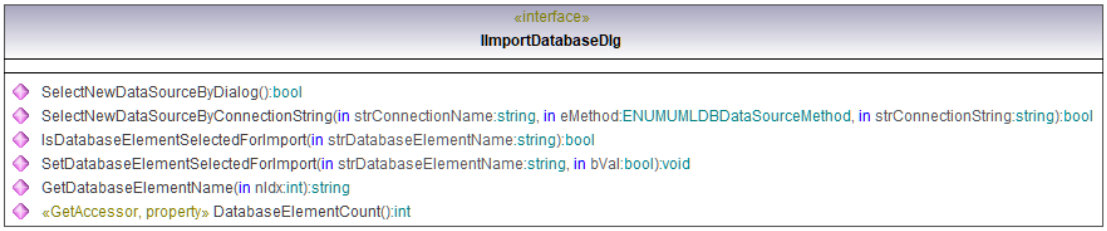
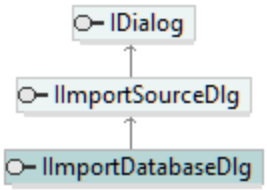
parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

© 2019-2025 Altova GmbH

UML documentation generated by UModel UML Editor http://www.altova.com/umodel Wed Sep 25 10:33:04 2024

16.4.2.17 UModelAPI - IImportDatabaseDlg

Interface **IImportDatabaseDlg**

diagram	
hierarchy	
typedElements	Interface IDialogs 830 Interface IDocument 832 Operation ImportDatabaseDlg 831 Operation ImportDatabase 835

Operation **IImportDatabaseDlg::DatabaseElementCount**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**int**			

Operation **IImportDatabaseDlg::GetDatabaseElementName**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	**string**			

Operation **IImportDatabaseDlg::IsDatabaseElementSelectedForImport**

parameter	name	direction	type	type modifier	multiplicity	default
	strDatabaseElementName	**in**	**string**			
	return	**return**	**bool**			

Operation **IImportDatabaseDlg::SelectNewDataSourceByConnectionString**

parameter	name	direction	type	type modifier	multiplicity	default
	strConnectionName	**in**	**string**			
	eMethod	**in**	**ENUMUMLDBDataSourceMethod**			

			1266			
	strConnectionStri ng	**in**	**string**			
	return	**return**	**bool**			

Operation **IImportDatabaseDlg::SelectNewDataSourceByDialog**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

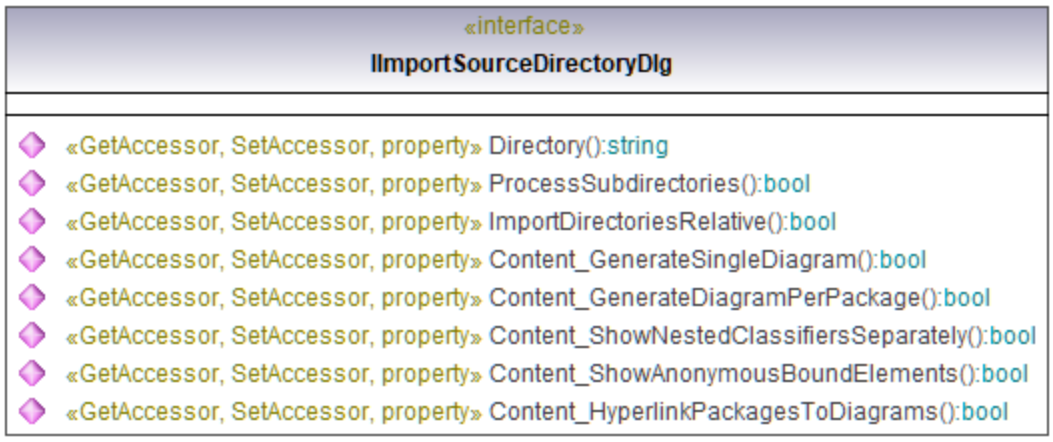
Operation **IImportDatabaseDlg::SetDatabaseElementSelectedForImport**

parameter	name	direction	type	type modifier	multiplicity	default
	strDatabaseEleme ntName	**in**	**string**			
	bVal	**in**	**bool**			
	return	**return**	**void**			

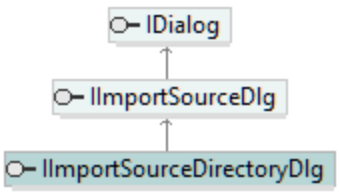
16.4.2.18 UModelAPI - IImportSourceDirectoryDlg

Interface **IImportSourceDirectoryDlg**

diagram



hierarchy



typedElements

Interface **IDialogs** 830
Interface **IDocument** 832
Operation **ImportSourceDirectoryDlg** 831
Operation **ImportSourceDirectory** 835

 © 2019-2025 Altova GmbH

Operation **IImportSourceDirectoryDlg::Content_GenerateDiagramPerPackage**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IImportSourceDirectoryDlg::Content_GenerateSingleDiagram**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IImportSourceDirectoryDlg::Content_HyperlinkPackagesToDiagrams**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IImportSourceDirectoryDlg::Content_ShowAnonymousBoundElements**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IImportSourceDirectoryDlg::Content_ShowNestedClassifiersSeparately**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IImportSourceDirectoryDlg::Directory**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**string**			

Operation **IImportSourceDirectoryDlg::ImportDirectoriesRelative**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IImportSourceDirectoryDlg::ProcessSubdirectories**

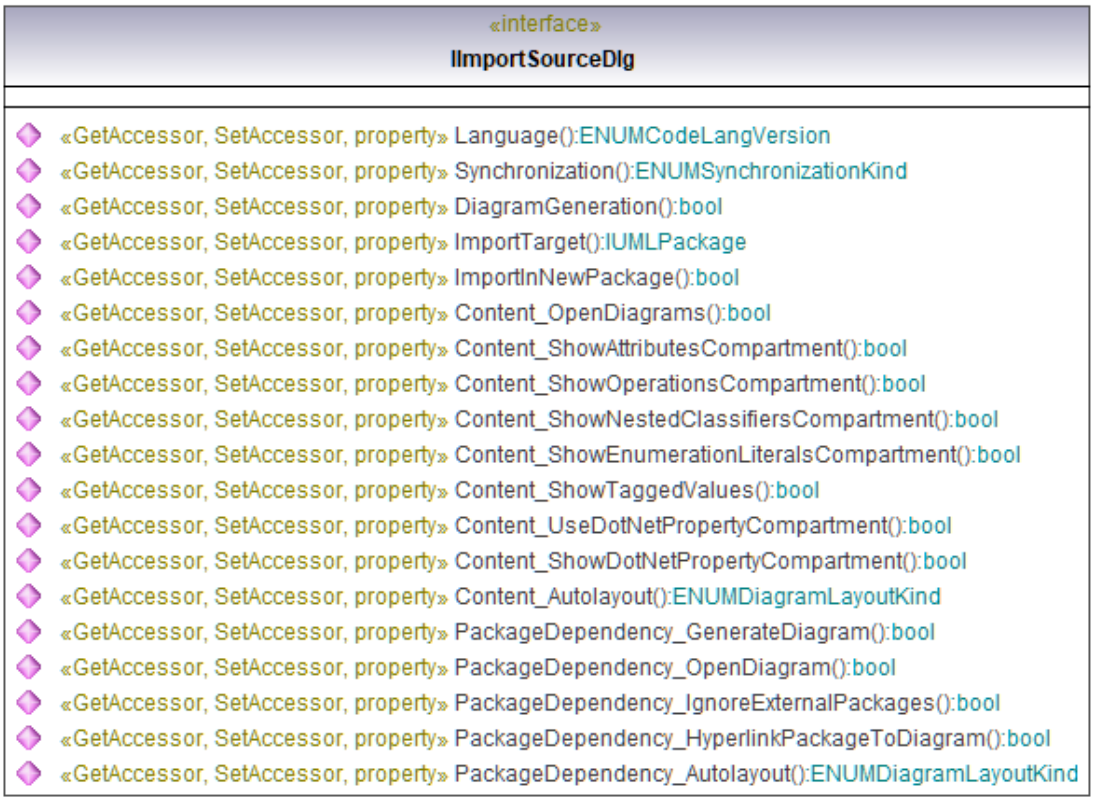
parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

© 2019-2025 Altova GmbH

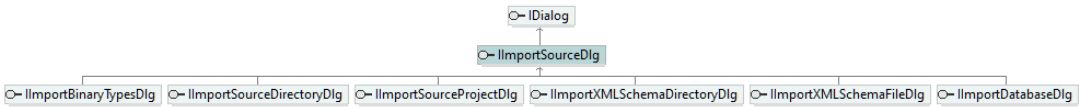
16.4.2.19 UModelAPI - IImportSourceDlg

Interface **IImportSourceDlg**

diagram



hierarchy



Operation **IImportSourceDlg::Content_Autolayout**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**ENUMDiagramLayoutKind** 898			

Operation **IImportSourceDlg::Content_OpenDiagrams**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IImportSourceDlg::Content_ShowAttributesCompartment**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IImportSourceDlg::Content_ShowDotNetPropertyCompartment**

© 2019-2025 Altova GmbH

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IImportSourceDlg::Content_ShowEnumerationLiteralsCompartment**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IImportSourceDlg::Content_ShowNestedClassifiersCompartment**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IImportSourceDlg::Content_ShowOperationsCompartment**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IImportSourceDlg::Content_ShowTaggedValues**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IImportSourceDlg::Content_UseDotNetPropertyCompartment**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IImportSourceDlg::DiagramGeneration**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IImportSourceDlg::ImportInNewPackage**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IImportSourceDlg::ImportTarget**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLPackage** 1134			

Operation **IImportSourceDlg::Language**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**ENUMCodeLangVersion** 898			

Operation **IImportSourceDlg::PackageDependency_Autolayout**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**ENUMDiagramLayoutKind** 898			

Operation **IImportSourceDlg::PackageDependency_GenerateDiagram**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IImportSourceDlg::PackageDependency_HyperlinkPackageToDiagram**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IImportSourceDlg::PackageDependency_IgnoreExternalPackages**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IImportSourceDlg::PackageDependency_OpenDiagram**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

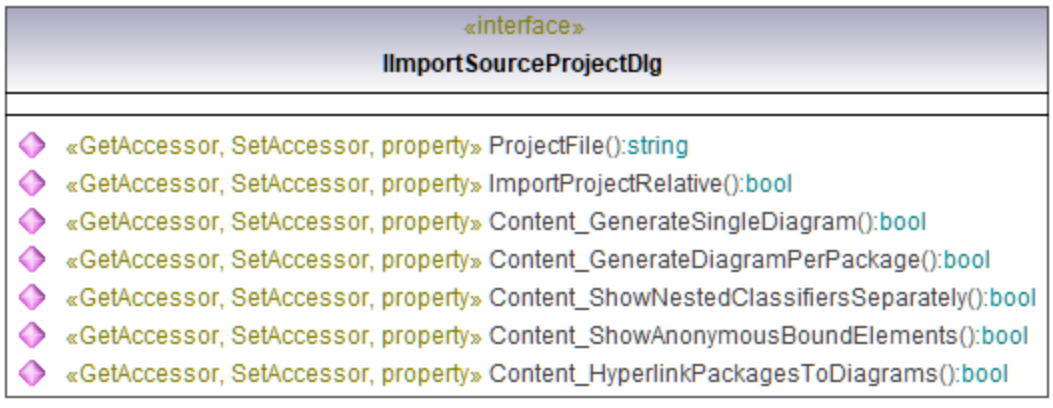
Operation **IImportSourceDlg::Synchronization**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**ENUMSynchronizationKind** 902			

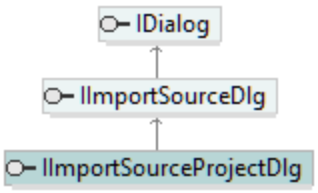
16.4.2.20 UModelAPI - IImportSourceProjectDlg

Interface **IImportSourceProjectDlg**

diagram



 © 2019-2025 Altova GmbH

hierarchy	
typedElements	Interface **IDialogs** 830 Interface **IDocument** 832 Operation **ImportSourceProjectDlg** 831 Operation **ImportSourceProject** 835

Operation **IImportSourceProjectDlg::Content_GenerateDiagramPerPackage**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IImportSourceProjectDlg::Content_GenerateSingleDiagram**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IImportSourceProjectDlg::Content_HyperlinkPackagesToDiagrams**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IImportSourceProjectDlg::Content_ShowAnonymousBoundElements**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IImportSourceProjectDlg::Content_ShowNestedClassifiersSeparately**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IImportSourceProjectDlg::ImportProjectRelative**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

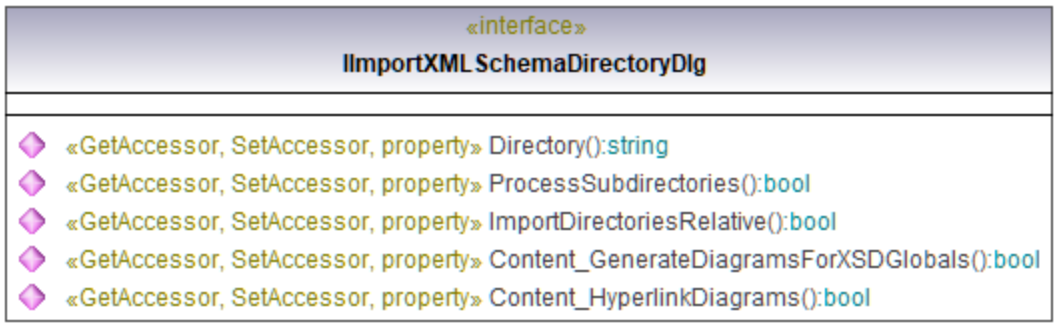
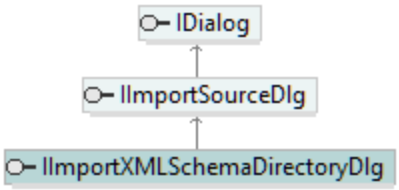
Operation **IImportSourceProjectDlg::ProjectFile**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**string**			

© 2019-2025 Altova GmbH

16.4.2.21 UModelAPI - IImportXMLSchemaDirectoryDlg

Interface **IImportXMLSchemaDirectoryDlg**

diagram	
hierarchy	
typedElements	Interface IDialogs 830 Interface IDocument 832 Operation ImportXMLSchemaDirectoryDlg 831 Operation ImportXMLSchemaDirectory 835

Operation **IImportXMLSchemaDirectoryDlg::Content_GenerateDiagramsForXSDGlobals**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IImportXMLSchemaDirectoryDlg::Content_HyperlinkDiagrams**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IImportXMLSchemaDirectoryDlg::Directory**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**string**			

Operation **IImportXMLSchemaDirectoryDlg::ImportDirectoriesRelative**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IImportXMLSchemaDirectoryDlg::ProcessSubdirectories**

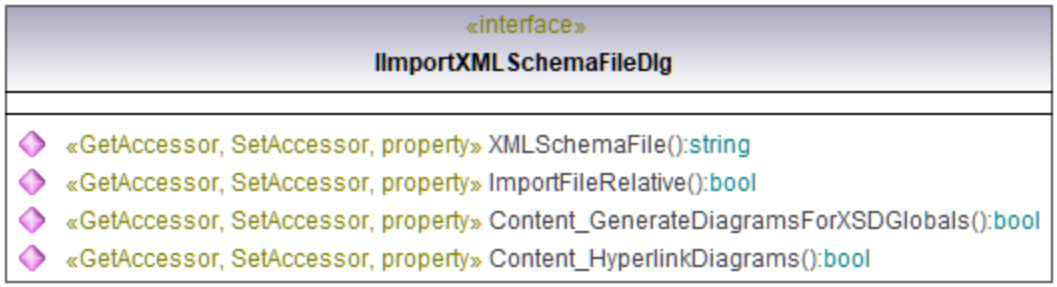
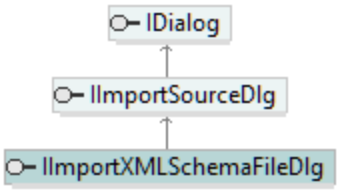
parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

© 2019-2025 Altova GmbH

UML documentation generated by UModel UML Editor http://www.altova.com/umodel Wed Sep 25 10:33:04 2024

16.4.2.22 UModelAPI - IImportXMLSchemaFileDlg

Interface **IImportXMLSchemaFileDlg**

diagram	
hierarchy	
typedElements	Interface IDialogs 830 Interface IDocument 832 Operation ImportXMLSchemaFileDlg 831 Operation ImportXMLSchemaFile 835

Operation **IImportXMLSchemaFileDlg::Content_GenerateDiagramsForXSDGlobals**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IImportXMLSchemaFileDlg::Content_HyperlinkDiagrams**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IImportXMLSchemaFileDlg::ImportFileRelative**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

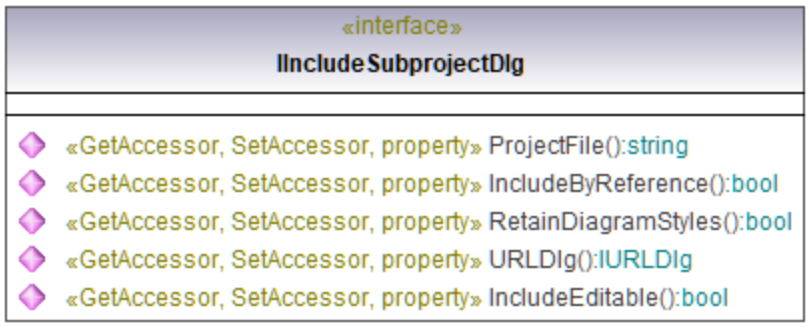
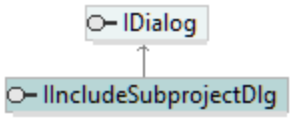
Operation **IImportXMLSchemaFileDlg::XMLSchemaFile**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**string**			

UML documentation generated by UModel UML Editor http://www.altova.com/umodel Wed Sep 25 10:33:04 2024

16.4.2.23 UModelAPI - IIncludeSubprojectDlg

Interface **IIncludeSubprojectDlg**

diagram	
hierarchy	
typedElements	Interface IDialogs 830 Interface IDocument 832 Operation IncludeSubprojectDlg 831 Operation IncludeSubproject 836

Operation **IIncludeSubprojectDlg::IncludeByReference**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IIncludeSubprojectDlg::IncludeEditable**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IIncludeSubprojectDlg::ProjectFile**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**string**			

Operation **IIncludeSubprojectDlg::RetainDiagramStyles**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

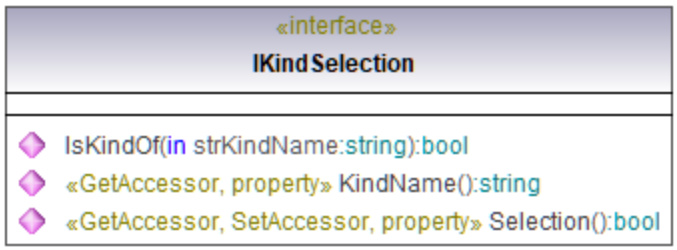
Operation **IIncludeSubprojectDlg::URLDlg**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IURLDlg 890			

© 2019-2025 Altova GmbH

16.4.2.24 UModelAPI - IKindSelection

Interface **IKindSelection**

diagram	
typedElements	Interface IKindSelectionList 865 Operation Item 866

Operation **IKindSelection::IsKindOf**

parameter	name	direction	type	type modifier	multiplicity	default
	strKindName	**in**	**string**			
	return	**return**	**bool**			

Operation **IKindSelection::KindName**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**string**			

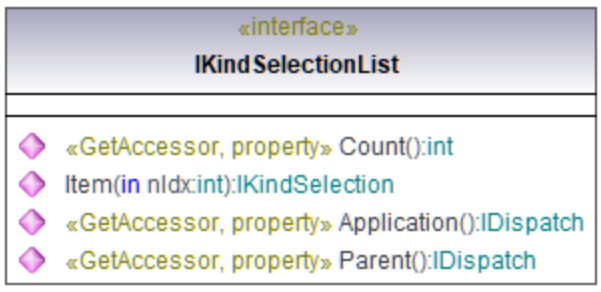
Operation **IKindSelection::Selection**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

UML documentation generated by **UModel** UML Editor **http://www.altova.com/umodel** Wed Sep 25 10:33:04 2024

16.4.2.25 UModelAPI - IKindSelectionList

Interface **IKindSelectionList**

diagram	
typedElements	Interface IGenerateDocumentationDlg 841 Operation Include_Diagrams 845 Include_Elements 846

Operation **IKindSelectionList::Application**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IDispatch**			

Operation **IKindSelectionList::Count**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**int**			

Operation **IKindSelectionList::Item**

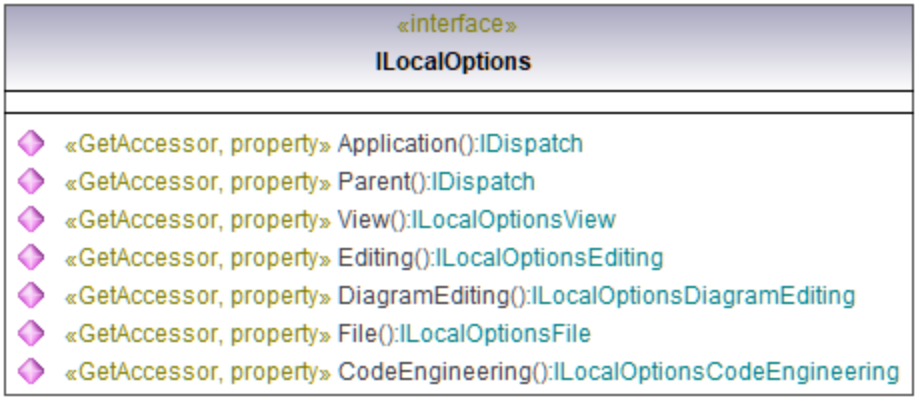
parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	**IKindSelection** 865			

Operation **IKindSelectionList::Parent**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IDispatch**			

16.4.2.26 UModelAPI - ILocalOptions

Interface **ILocalOptions**

diagram	
typedElements	Interface **IApplication** 818 Operation **Options** 820

Operation **ILocalOptions::Application**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IDispatch**			

Operation **ILocalOptions::CodeEngineering**

 © 2019-2025 Altova GmbH

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**ILocalOptionsCodeEngineering** 868			

Operation **ILocalOptions::DiagramEditing**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**ILocalOptionsDiagramEditing** 870			

Operation **ILocalOptions::Editing**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**ILocalOptionsEditing** 873			

Operation **ILocalOptions::File**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**ILocalOptionsFile** 874			

Operation **ILocalOptions::Parent**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IDispatch**			

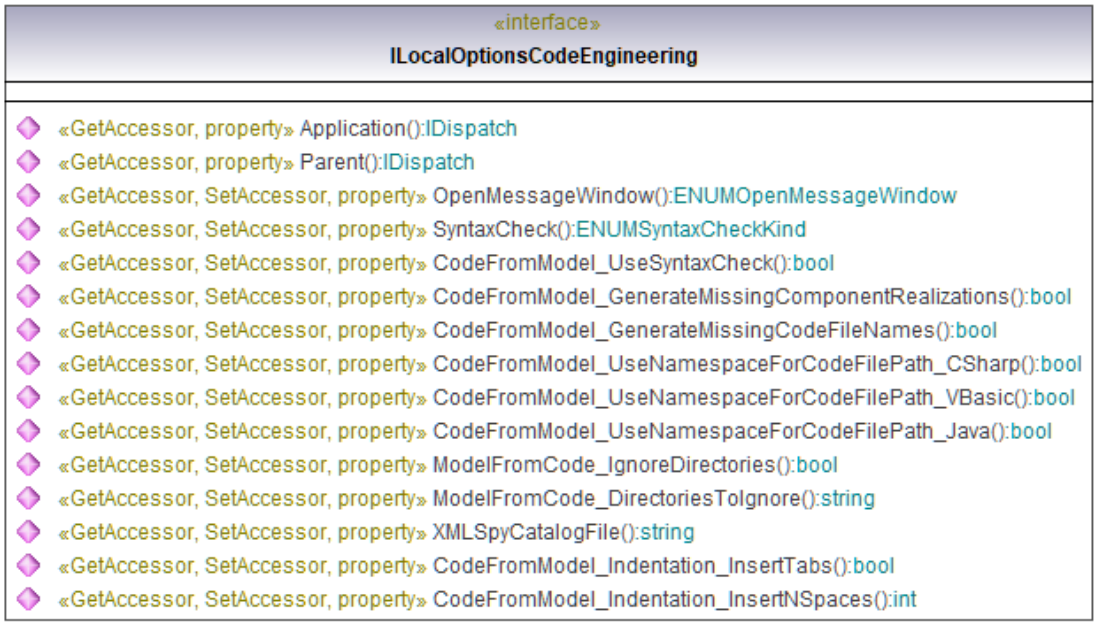
Operation **ILocalOptions::View**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**ILocalOptionsView** 876			

© 2019-2025 Altova GmbH

16.4.2.27 UModelAPI - ILocalOptionsCodeEngineering

Interface **ILocalOptionsCodeEngineering**

diagram	
typedElements	Interface **ILocalOptions** 866 Operation **CodeEngineering** 866

Operation **ILocalOptionsCodeEngineering::Application**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IDispatch**			

Operation **ILocalOptionsCodeEngineering::CodeFromModel_GenerateMissingCodeFileNames**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **ILocalOptionsCodeEngineering::CodeFromModel_GenerateMissingComponentRealizations**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **ILocalOptionsCodeEngineering::CodeFromModel_Indentation_InsertNSpaces**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**int**			

Operation **ILocalOptionsCodeEngineering::CodeFromModel_Indentation_InsertTabs**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **ILocalOptionsCodeEngineering::CodeFromModel_UseNamespaceForCodeFilePath_CSharp**

© 2019-2025 Altova GmbH

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **ILocalOptionsCodeEngineering::CodeFromModel_UseNamespaceForCodeFilePath_Java**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **ILocalOptionsCodeEngineering::CodeFromModel_UseNamespaceForCodeFilePath_VBasic**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **ILocalOptionsCodeEngineering::CodeFromModel_UseSyntaxCheck**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **ILocalOptionsCodeEngineering::ModelFromCode_DirectoriesToIgnore**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**string**			

Operation **ILocalOptionsCodeEngineering::ModelFromCode_IgnoreDirectories**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **ILocalOptionsCodeEngineering::OpenMessageWindow**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**ENUMOpenMessageWindow** 901			

Operation **ILocalOptionsCodeEngineering::Parent**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IDispatch**			

Operation **ILocalOptionsCodeEngineering::SyntaxCheck**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**ENUMSyntaxCheckKind** 902			

Operation **ILocalOptionsCodeEngineering::XMLSpyCatalogFile**

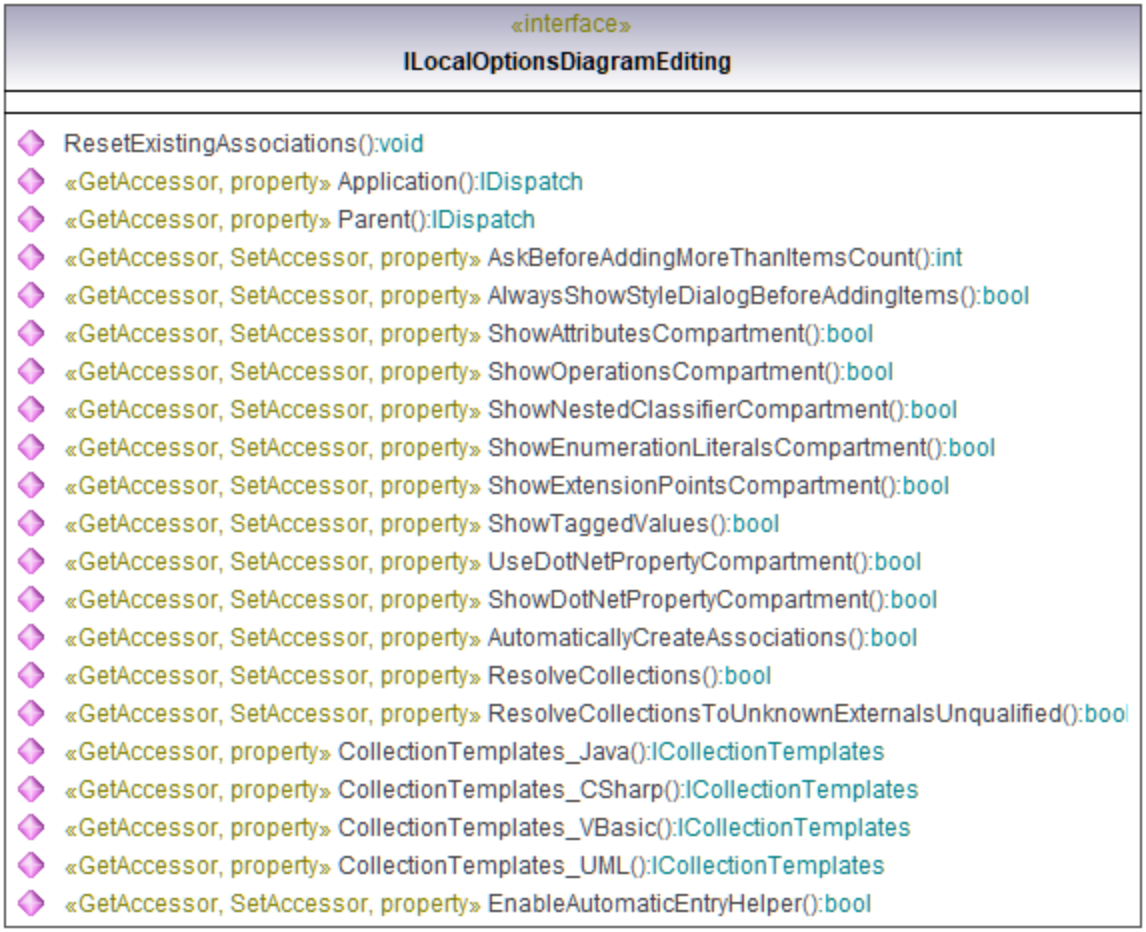
parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**string**			

© 2019-2025 Altova GmbH

16.4.2.28 UModelAPI - ILocalOptionsDiagramEditing

Interface **ILocalOptionsDiagramEditing**

diagram



typedElements

Interface **ILocalOptions** 866 Operation **DiagramEditing** 867

Operation **ILocalOptionsDiagramEditing::AlwaysShowStyleDialogBeforeAddingItems**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **ILocalOptionsDiagramEditing::Application**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IDispatch**			

Operation **ILocalOptionsDiagramEditing::AskBeforeAddingMoreThanItemsCount**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**int**			

© 2019-2025 Altova GmbH

Operation **ILocalOptionsDiagramEditing::AutomaticallyCreateAssociations**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **ILocalOptionsDiagramEditing::CollectionTemplates_CSharp**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**ICollectionTemplates** 824			

Operation **ILocalOptionsDiagramEditing::CollectionTemplates_Java**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**ICollectionTemplates** 824			

Operation **ILocalOptionsDiagramEditing::CollectionTemplates_UML**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**ICollectionTemplates** 824			

Operation **ILocalOptionsDiagramEditing::CollectionTemplates_VBasic**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**ICollectionTemplates** 824			

Operation **ILocalOptionsDiagramEditing::EnableAutomaticEntryHelper**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **ILocalOptionsDiagramEditing::Parent**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IDispatch**			

Operation **ILocalOptionsDiagramEditing::ResetExistingAssociations**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**void**			

Operation **ILocalOptionsDiagramEditing::ResolveCollections**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **ILocalOptionsDiagramEditing::ResolveCollectionsToUnknownExternalsUnqualified**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **ILocalOptionsDiagramEditing::ShowAttributesCompartment**

parameter	name	direction	type	type modifier	multiplicity	default

	return	**return**	**bool**

Operation **ILocalOptionsDiagramEditing::ShowDotNetPropertyCompartment**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **ILocalOptionsDiagramEditing::ShowEnumerationLiteralsCompartment**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **ILocalOptionsDiagramEditing::ShowExtensionPointsCompartment**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **ILocalOptionsDiagramEditing::ShowNestedClassifierCompartment**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **ILocalOptionsDiagramEditing::ShowOperationsCompartment**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **ILocalOptionsDiagramEditing::ShowTaggedValues**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

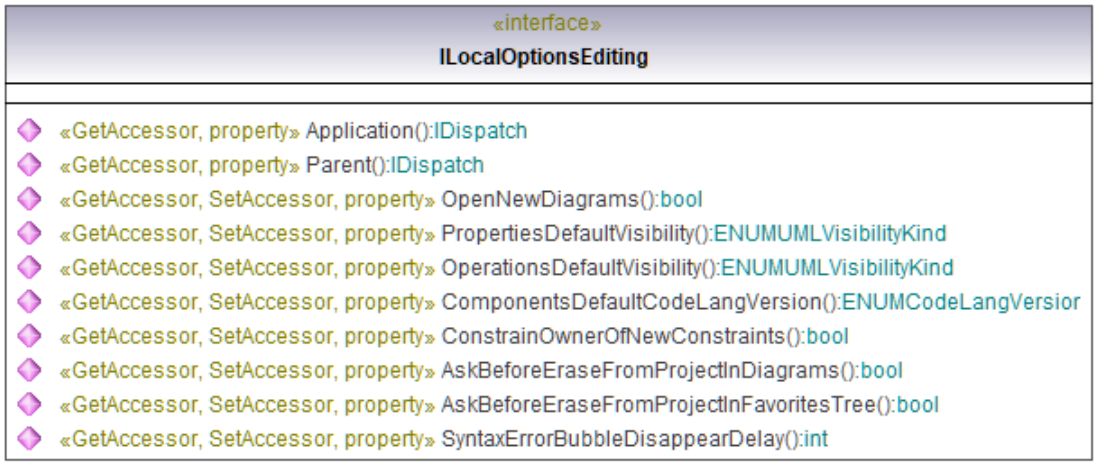
Operation **ILocalOptionsDiagramEditing::UseDotNetPropertyCompartment**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

© 2019-2025 Altova GmbH

16.4.2.29 UModelAPI - ILocalOptionsEditing

Interface **ILocalOptionsEditing**

diagram	
typedElements	Interface ILocalOptions 866 Operation Editing 867

Operation **ILocalOptionsEditing::Application**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IDispatch**			

Operation **ILocalOptionsEditing::AskBeforeEraseFromProjectInDiagrams**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **ILocalOptionsEditing::AskBeforeEraseFromProjectInFavoritesTree**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **ILocalOptionsEditing::ComponentsDefaultCodeLangVersion**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	ENUMCodeLangVersion 898			

Operation **ILocalOptionsEditing::ConstrainOwnerOfNewConstraints**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **ILocalOptionsEditing::OpenNewDiagrams**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **ILocalOptionsEditing::OperationsDefaultVisibility**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**ENUMUMLVisibilityKind** 1273			

Operation **ILocalOptionsEditing::Parent**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IDispatch**			

Operation **ILocalOptionsEditing::PropertiesDefaultVisibility**

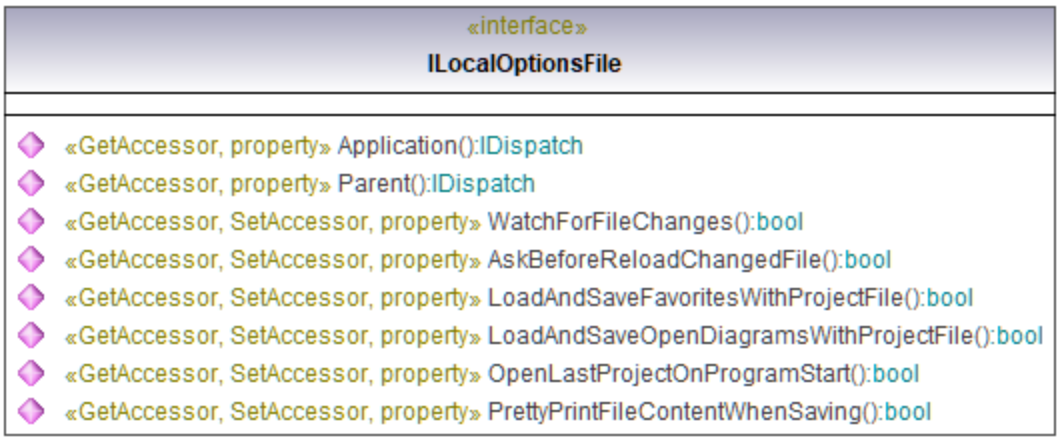
parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**ENUMUMLVisibilityKind** 1273			

Operation **ILocalOptionsEditing::SyntaxErrorBubbleDisappearDelay**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**int**			

16.4.2.30 UModelAPI - ILocalOptionsFile

Interface **ILocalOptionsFile**

diagram	
typedElements	Interface **ILocalOptions** 866 Operation **File** 867

Operation **ILocalOptionsFile::Application**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IDispatch**			

Operation **ILocalOptionsFile::AskBeforeReloadChangedFile**

 © 2019-2025 Altova GmbH

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **ILocalOptionsFile::LoadAndSaveFavoritesWithProjectFile**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **ILocalOptionsFile::LoadAndSaveOpenDiagramsWithProjectFile**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **ILocalOptionsFile::OpenLastProjectOnProgramStart**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **ILocalOptionsFile::Parent**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IDispatch**			

Operation **ILocalOptionsFile::PrettyPrintFileContentWhenSaving**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

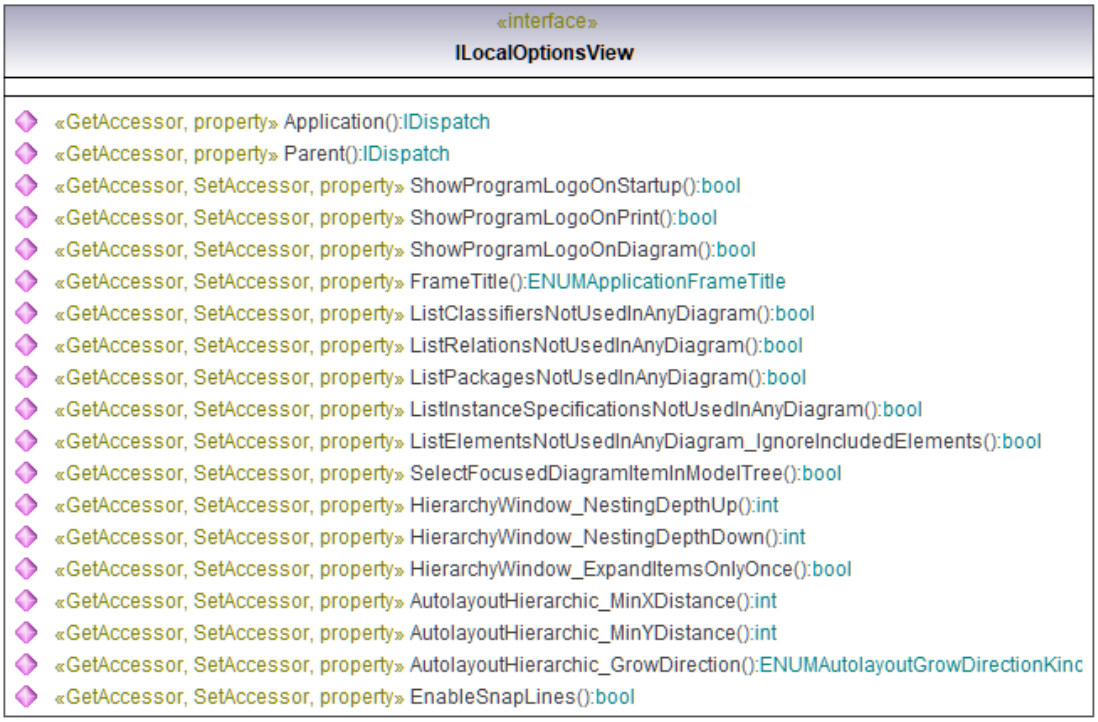
Operation **ILocalOptionsFile::WatchForFileChanges**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

© 2019-2025 Altova GmbH

16.4.2.31 UModelAPI - ILocalOptionsView

Interface **ILocalOptionsView**

diagram	
typedElements	Interface **ILocalOptions** 866 Operation **View** 867

Operation **ILocalOptionsView::Application**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IDispatch**			

Operation **ILocalOptionsView::AutolayoutHierarchic_GrowDirection**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**ENUMAutolayoutGrowDirectionKind** 897			

Operation **ILocalOptionsView::AutolayoutHierarchic_MinXDistance**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**int**			

Operation **ILocalOptionsView::AutolayoutHierarchic_MinYDistance**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**int**			

Operation **ILocalOptionsView::EnableSnapLines**

© 2019-2025 Altova GmbH

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **ILocalOptionsView::FrameTitle**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**ENUMApplication FrameTitle** 896			

Operation **ILocalOptionsView::HierarchyWindow_ExpandItemsOnlyOnce**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **ILocalOptionsView::HierarchyWindow_NestingDepthDown**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**int**			

Operation **ILocalOptionsView::HierarchyWindow_NestingDepthUp**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**int**			

Operation **ILocalOptionsView::ListClassifiersNotUsedInAnyDiagram**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **ILocalOptionsView::ListElementsNotUsedInAnyDiagram_IgnoreIncludedElements**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **ILocalOptionsView::ListInstanceSpecificationsNotUsedInAnyDiagram**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **ILocalOptionsView::ListPackagesNotUsedInAnyDiagram**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **ILocalOptionsView::ListRelationsNotUsedInAnyDiagram**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **ILocalOptionsView::Parent**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IDispatch**			

Operation **ILocalOptionsView::SelectFocusedDiagramItemInModelTree**

parameter	name	direction	type	type modifier	multiplicity	default

	return	**return**	**bool**

Operation **ILocalOptionsView::ShowProgramLogoOnDiagram**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **ILocalOptionsView::ShowProgramLogoOnPrint**

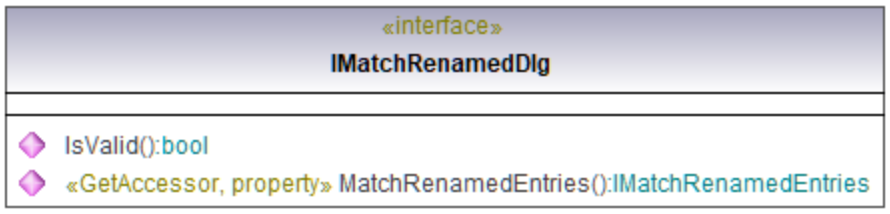
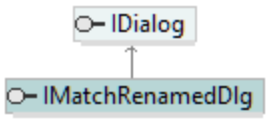
parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **ILocalOptionsView::ShowProgramLogoOnStartup**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

16.4.2.32 UModelAPI - IMatchRenamedDlg

Interface **IMatchRenamedDlg**

diagram	
hierarchy	
typedElements	Interface **ISynchronizationEvents** 895 Operation **OnMatchRenamed** 895

Operation **IMatchRenamedDlg::IsValid**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IMatchRenamedDlg::MatchRenamedEntries**

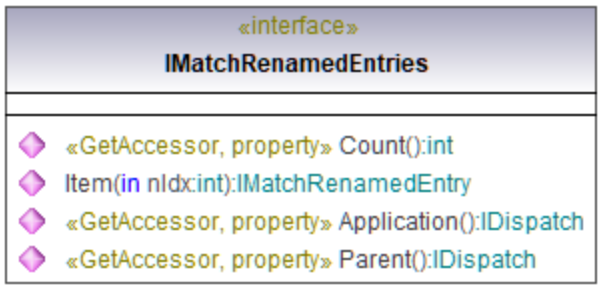
parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IMatchRenamedEntries** 879			

 © 2019-2025 Altova GmbH

UML documentation generated by UModel UML Editor http://www.altova.com/umodel Wed Sep 25 10:33:04 2024

16.4.2.33 UModelAPI - IMatchRenamedEntries

Interface **IMatchRenamedEntries**

diagram	
typedElements	Interface IMatchRenamedDlg 878 Operation MatchRenamedEntries 878

Operation **IMatchRenamedEntries::Application**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IDispatch**			

Operation **IMatchRenamedEntries::Count**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**int**			

Operation **IMatchRenamedEntries::Item**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	IMatchRenamedEntry 880			

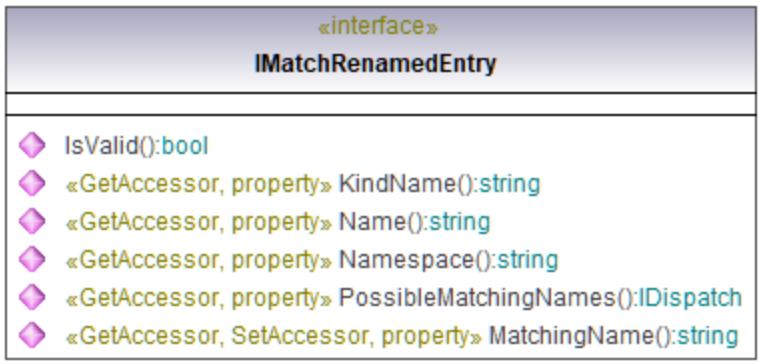
Operation **IMatchRenamedEntries::Parent**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IDispatch**			

UML documentation generated by UModel UML Editor http://www.altova.com/umodel Wed Sep 25 10:33:04 2024

16.4.2.34 UModelAPI - IMatchRenamedEntry

Interface **IMatchRenamedEntry**

diagram	
typedElements	Interface **IMatchRenamedEntries** 879 Operation **Item** 879

Operation **IMatchRenamedEntry::IsValid**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IMatchRenamedEntry::KindName**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**string**			

Operation **IMatchRenamedEntry::MatchingName**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**string**			

Operation **IMatchRenamedEntry::Name**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**string**			

Operation **IMatchRenamedEntry::Namespace**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**string**			

Operation **IMatchRenamedEntry::PossibleMatchingNames**

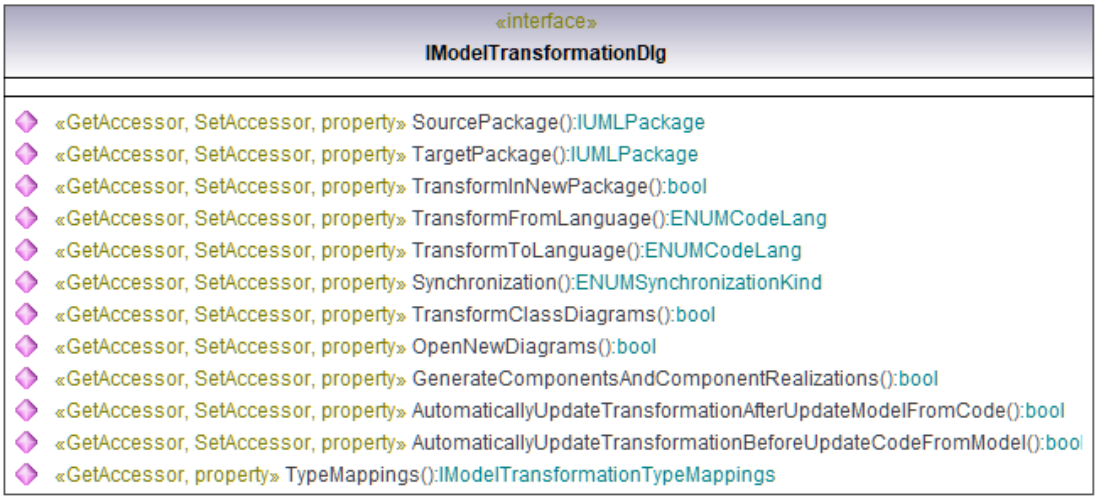
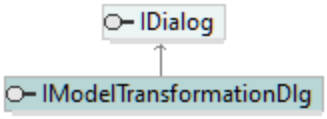
parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IDispatch**			
documentation	Returns an array of values of type **string**.					

© 2019-2025 Altova GmbH

UML documentation generated by UModel UML Editor http://www.altova.com/umodel Wed Sep 25 10:33:04 2024

16.4.2.35 UModelAPI - IModelTransformationDlg

Interface **IModelTransformationDlg**

diagram	
hierarchy	
typedElements	Interface IDialogs 830 Interface IDocument 832 Operation ModelTransformationDlg 831 Operation ModelTransformation 836

Operation **IModelTransformationDlg::AutomaticallyUpdateTransformationAfterUpdateModelFromCode**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IModelTransformationDlg::AutomaticallyUpdateTransformationBeforeUpdateCodeFromModel**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IModelTransformationDlg::GenerateComponentsAndComponentRealizations**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IModelTransformationDlg::OpenNewDiagrams**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IModelTransformationDlg::SourcePackage**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLPackage** 1134			

Operation **IModelTransformationDlg::Synchronization**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**ENUMSynchronizationKind** 902			

Operation **IModelTransformationDlg::TargetPackage**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLPackage** 1134			

Operation **IModelTransformationDlg::TransformClassDiagrams**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IModelTransformationDlg::TransformFromLanguage**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**ENUMCodeLang** 897			

Operation **IModelTransformationDlg::TransformInNewPackage**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IModelTransformationDlg::TransformToLanguage**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**ENUMCodeLang** 897			

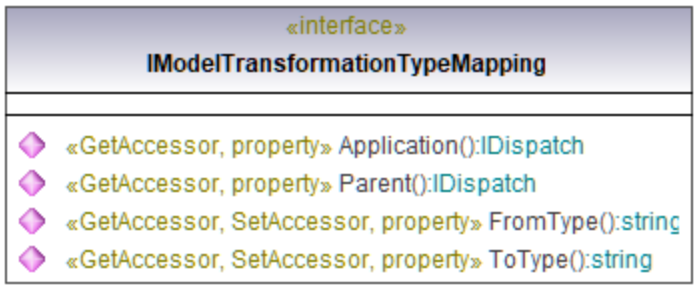
Operation **IModelTransformationDlg::TypeMappings**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IModelTransformationTypeMappings** 884			

© 2019-2025 Altova GmbH

16.4.2.36 UModelAPI - IModelTransformationTypeMapping

Interface **IModelTransformationTypeMapping**

diagram	
typedElements	Interface IModelTransformationTypeMappings 884 Operation InsertItemAt 884 Item 884

Operation **IModelTransformationTypeMapping::Application**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IDispatch**			

Operation **IModelTransformationTypeMapping::FromType**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**string**			

Operation **IModelTransformationTypeMapping::Parent**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IDispatch**			

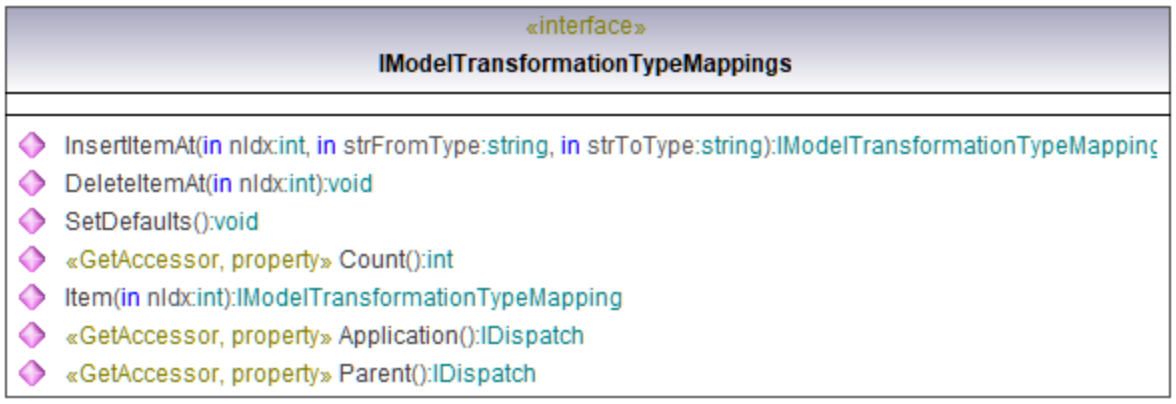
Operation **IModelTransformationTypeMapping::ToType**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**string**			

UML documentation generated by **UModel** UML Editor **http://www.altova.com/umodel** Wed Sep 25 10:33:04 2024

16.4.2.37 UModelAPI - IModelTransformationTypeMappings

Interface **IModelTransformationTypeMappings**

diagram	
typedElements	Interface IModelTransformationDlg 881 Operation TypeMappings 882

Operation **IModelTransformationTypeMappings::Application**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IDispatch**			

Operation **IModelTransformationTypeMappings::Count**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**int**			

Operation **IModelTransformationTypeMappings::DeleteItemAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	**void**			

Operation **IModelTransformationTypeMappings::InsertItemAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	strFromType	**in**	**string**			
	strToType	**in**	**string**			
	return	**return**	IModelTransformationTypeMapping 883			

Operation **IModelTransformationTypeMappings::Item**

© 2019-2025 Altova GmbH

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	**IModelTransformationTypeMapping** 883			

Operation **IModelTransformationTypeMappings::Parent**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IDispatch**			

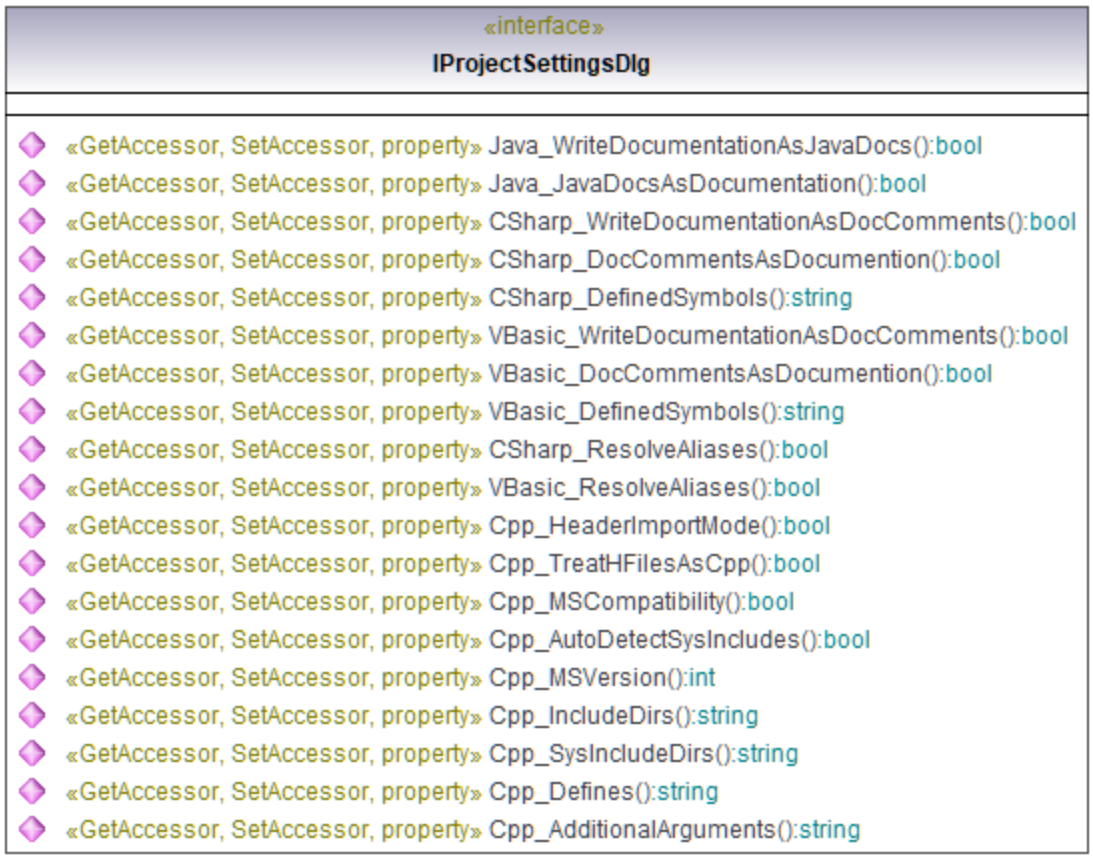
Operation **IModelTransformationTypeMappings::SetDefaults**

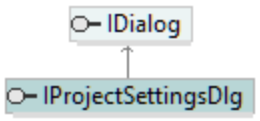
parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**void**			

16.4.2.38 UModelAPI - IProjectSettingsDlg

Interface **IProjectSettingsDlg**

diagram



hierarchy	
typedElements	Interface IDialogs 830 Interface IDocument 832 Operation ProjectSettingsDlg 832 Operation ProjectSettings 837

Operation **IProjectSettingsDlg::Cpp_AdditionalArguments**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**string**			

Operation **IProjectSettingsDlg::Cpp_AutoDetectSysIncludes**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IProjectSettingsDlg::Cpp_Defines**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**string**			

Operation **IProjectSettingsDlg::Cpp_HeaderImportMode**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IProjectSettingsDlg::Cpp_IncludeDirs**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**string**			

Operation **IProjectSettingsDlg::Cpp_MSCompatibility**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IProjectSettingsDlg::Cpp_MSVersion**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**int**			

Operation **IProjectSettingsDlg::Cpp_SysIncludeDirs**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**string**			

Operation **IProjectSettingsDlg::Cpp_TreatHFilesAsCpp**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IProjectSettingsDlg::CSharp_DefinedSymbols**

© 2019-2025 Altova GmbH

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**string**			

Operation **IProjectSettingsDlg::CSharp_DocCommentsAsDocumention**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IProjectSettingsDlg::CSharp_ResolveAliases**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IProjectSettingsDlg::CSharp_WriteDocumentationAsDocComments**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IProjectSettingsDlg::Java_JavaDocsAsDocumentation**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IProjectSettingsDlg::Java_WriteDocumentationAsJavaDocs**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IProjectSettingsDlg::VBasic_DefinedSymbols**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**string**			

Operation **IProjectSettingsDlg::VBasic_DocCommentsAsDocumention**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IProjectSettingsDlg::VBasic_ResolveAliases**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

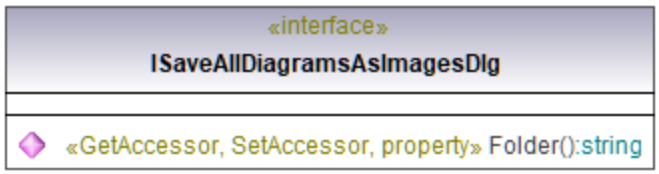
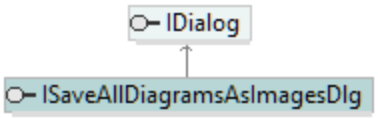
Operation **IProjectSettingsDlg::VBasic_WriteDocumentationAsDocComments**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

© 2019-2025 Altova GmbH

16.4.2.39 UModelAPI - ISaveAllDiagramsAsImagesDlg

Interface **ISaveAllDiagramsAsImagesDlg**

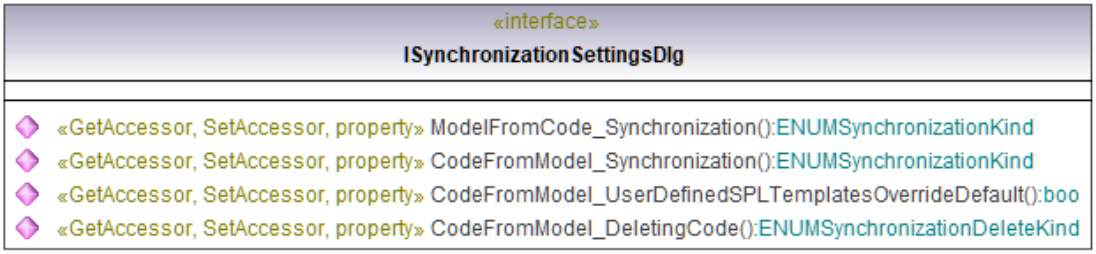
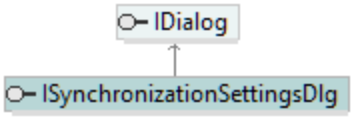
diagram	
hierarchy	
typedElements	Interface IDialogs 830 Interface IDocument 832 Operation SaveAllDiagramsAsImagesDlg 832 Operation SaveAllDiagramsAsImages 838

Operation **ISaveAllDiagramsAsImagesDlg::Folder**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**string**			

16.4.2.40 UModelAPI - ISynchronizationSettingsDlg

Interface **ISynchronizationSettingsDlg**

diagram	
hierarchy	
typedElements	Interface IDialogs 830 Operation SynchronizationSettingsDlg 832

 © 2019-2025 Altova GmbH

	Interface IDocument 832	Operation SynchronizationSettings 838 SynchronizeCodeFromModel 838 SynchronizeModelFromCode 838

Operation **ISynchronizationSettingsDlg::CodeFromModel_DeletingCode**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	ENUMSynchronizationDeleteKind 901			

Operation **ISynchronizationSettingsDlg::CodeFromModel_Synchronization**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	ENUMSynchronizationKind 902			

Operation **ISynchronizationSettingsDlg::CodeFromModel_UserDefinedSPLTemplatesOverrideDefault**

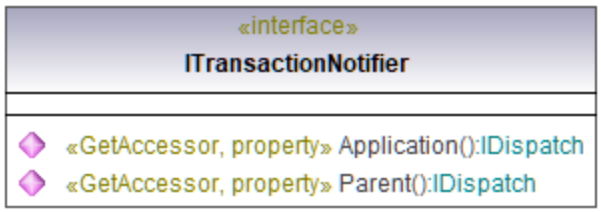
parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **ISynchronizationSettingsDlg::ModelFromCode_Synchronization**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	ENUMSynchronizationKind 902			

16.4.2.41 UModelAPI - ITransactionNotifier

Interface **ITransactionNotifier**

diagram	
typedElements	Interface IDocument 832 Operation TransactionNotifier 839
documentation	Use this interface to register for _ITransactionEvents 895.

Operation **ITransactionNotifier::Application**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IDispatch**			

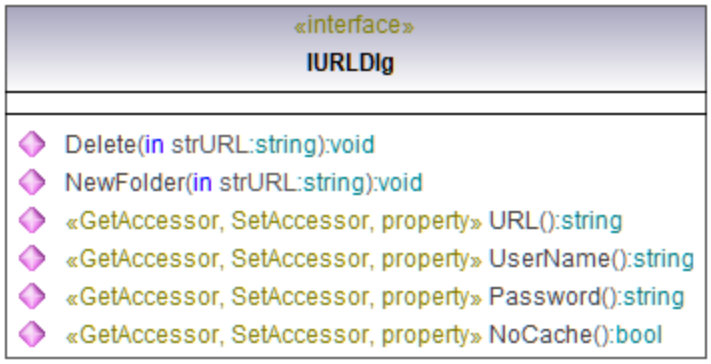
Operation **ITransactionNotifier::Parent**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IDispatch**			

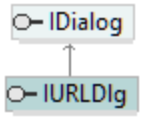
16.4.2.42 UModelAPI - IURLDlg

Interface **IURLDlg**

diagram



hierarchy



typedElements

Interface **IApplication** 818
Interface **IDialogs** 830
Interface **IDocument** 832
Interface **IExportXMIFileDlg** 839
Interface **IIncludeSubprojectDlg** 864
Operation **ImportFromXMIFileFromURL** 819
OpenDocumentFromURL 820
Operation **URLOpenDialog** 832
URLSaveDialog 832
Operation **MergeProjectFromURL** 836
SaveToURL 838
Operation **URLDlg** 840
Operation **URLDlg** 864

Operation **IURLDlg::Delete**

parameter	name	direction	type	type modifier	multiplicity	default
	strURL	**in**	**string**			
	return	**return**	**void**			

Operation **IURLDlg::NewFolder**

parameter	name	direction	type	type modifier	multiplicity	default
	strURL	**in**	**string**			
	return	**return**	**void**			

© 2019-2025 Altova GmbH

Operation **IURLDlg::NoCache**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IURLDlg::Password**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**string**			

Operation **IURLDlg::URL**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**string**			

Operation **IURLDlg::UserName**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**string**			

16.4.2.43 Events

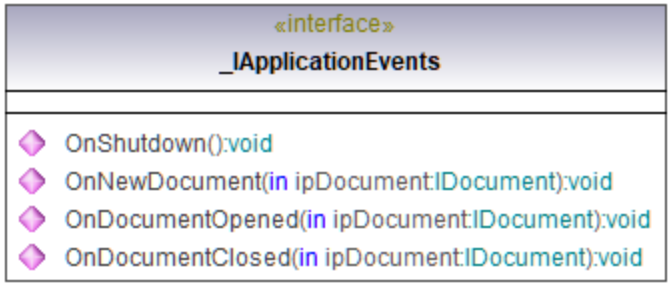
This is a list of all events sent by the UModel API.

A list of events sent on `UMLData` level can be found here[1262].

16.4.2.43.1 UModelAPI - _IApplicationEvents

Interface **_IApplicationEvents**

diagram



Operation **_IApplicationEvents::OnDocumentClosed**

parameter	name	direction	type	type modifier	multiplicity	default
	ipDocument	**in**	**IDocument**[832]			
	return	**return**	**void**			

Operation **_IApplicationEvents::OnDocumentOpened**

parameter	name	direction	type	type modifier	multiplicity	default
	ipDocument	**in**	**IDocument** 832			
	return	**return**	**void**			

Operation **_IApplicationEvents::OnNewDocument**

parameter	name	direction	type	type modifier	multiplicity	default
	ipDocument	**in**	**IDocument** 832			
	return	**return**	**void**			

Operation **_IApplicationEvents::OnShutdown**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**void**			

16.4.2.43.2 UModelAPI - _IDiagramWindowEvents

Interface **_IDiagramWindowEvents**

diagram

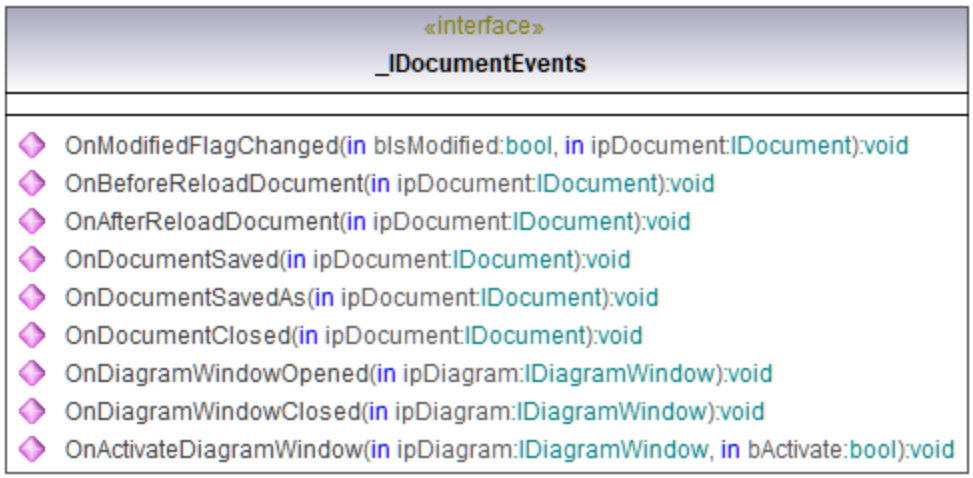
Operation **_IDiagramWindowEvents::OnDiagramWindowClosed**

parameter	name	direction	type	type modifier	multiplicity	default
	ipDiagram	**in**	**IDiagramWindow** 825			
	return	**return**	**void**			

 © 2019-2025 Altova GmbH

16.4.2.43.3 UModelAPI - _IDocumentEvents

Interface **_IDocumentEvents**

diagram	

Operation **_IDocumentEvents::OnActivateDiagramWindow**

parameter	name	direction	type	type modifier	multiplicity	default
	ipDiagram	**in**	**IDiagramWindow** 825			
	bActivate	**in**	**bool**			
	return	**return**	**void**			

Operation **_IDocumentEvents::OnAfterReloadDocument**

parameter	name	direction	type	type modifier	multiplicity	default
	ipDocument	**in**	**IDocument** 832			
	return	**return**	**void**			

Operation **_IDocumentEvents::OnBeforeReloadDocument**

parameter	name	direction	type	type modifier	multiplicity	default
	ipDocument	**in**	**IDocument** 832			
	return	**return**	**void**			

Operation **_IDocumentEvents::OnDiagramWindowClosed**

parameter	name	direction	type	type modifier	multiplicity	default
	ipDiagram	**in**	**IDiagramWindow** 825			
	return	**return**	**void**			

Operation **_IDocumentEvents::OnDiagramWindowOpened**

parameter	name	direction	type	type modifier	multiplicity	default
	ipDiagram	**in**	**IDiagramWindow** 825			
	return	**return**	**void**			

Operation **_IDocumentEvents::OnDocumentClosed**

parameter	name	direction	type	type modifier	multiplicity	default
	ipDocument	**in**	**IDocument** 832			
	return	**return**	**void**			

Operation **_IDocumentEvents::OnDocumentSaved**

parameter	name	direction	type	type modifier	multiplicity	default
	ipDocument	**in**	**IDocument** 832			
	return	**return**	**void**			

Operation **_IDocumentEvents::OnDocumentSavedAs**

parameter	name	direction	type	type modifier	multiplicity	default
	ipDocument	**in**	**IDocument** 832			
	return	**return**	**void**			

Operation **_IDocumentEvents::OnModifiedFlagChanged**

parameter	name	direction	type	type modifier	multiplicity	default
	bIsModified	**in**	**bool**			
	ipDocument	**in**	**IDocument** 832			
	return	**return**	**void**			

16.4.2.43.4 UModelAPI - _IFocusedUMLDataEvents

Interface **_IFocusedUMLDataEvents**

diagram	
	«interface» **_IFocusedUMLDataEvents** OnFocusedUMLData(in ipUMLData:IUMLData):void

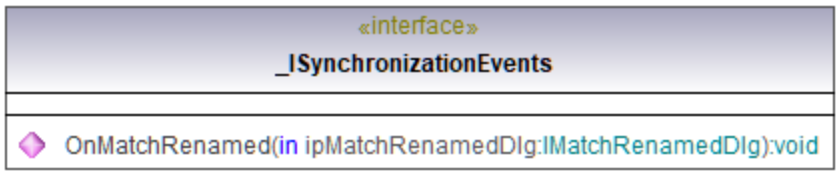
Operation **_IFocusedUMLDataEvents::OnFocusedUMLData**

parameter	name	direction	type	type modifier	multiplicity	default
	ipUMLData	**in**	**IUMLData** 904			
	return	**return**	**void**			

 © 2019-2025 Altova GmbH

16.4.2.43.5 UModelAPI - _ISynchronizationEvents

Interface **_ISynchronizationEvents**

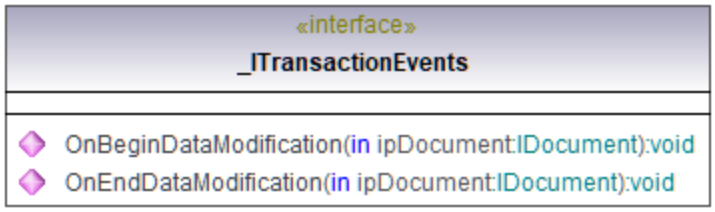
diagram	

Operation **_ISynchronizationEvents::OnMatchRenamed**

parameter	name	direction	type	type modifier	multiplicity	default
	ipMatchRenamedDlg	**in**	**IMatchRenamedDlg** 878			
	return	**return**	**void**			

16.4.2.43.6 UModelAPI - _ITransactionEvents

Interface **_ITransactionEvents**

diagram	
documentation	Use ITransactionNotifier 889 to register for these events

Operation **_ITransactionEvents::OnBeginDataModification**

parameter	name	direction	type	type modifier	multiplicity	default
	ipDocument	**in**	**IDocument** 832			
	return	**return**	**void**			

Operation **_ITransactionEvents::OnEndDataModification**

parameter	name	direction	type	type modifier	multiplicity	default
	ipDocument	**in**	**IDocument** 832			
	return	**return**	**void**			

© 2019-2025 Altova GmbH

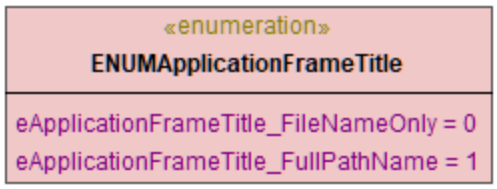
16.4.2.44 Enumerations

This is a list of all enumerations used by the UModel API. If your scripting environment does not support enumerations use the number-values instead.

A list of enumerations defined on `UMLData` level can be found here 1263.

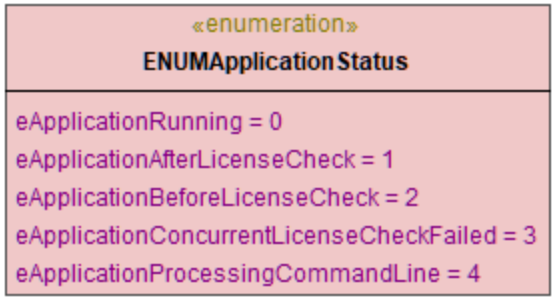
16.4.2.44.1 UModelAPI - ENUMApplicationFrameTitle

Enumeration **ENUMApplicationFrameTitle**

diagram	
typedElements	Interface **ILocalOptionsView** 876 Operation **FrameTitle** 877

16.4.2.44.2 UModelAPI - ENUMApplicationStatus

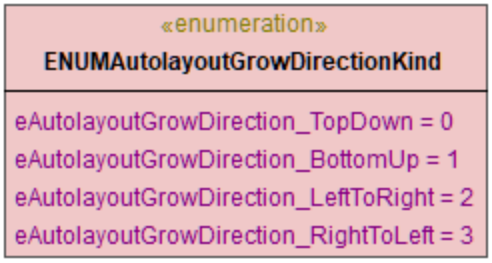
Enumeration **ENUMApplicationStatus**

diagram	
typedElements	Interface **IApplication** 818 Operation **Status** 821

 © 2019-2025 Altova GmbH

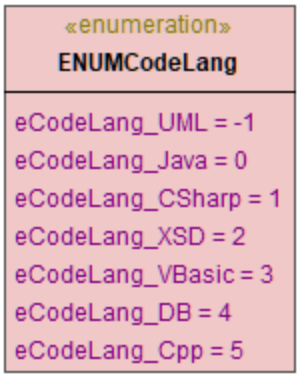
16.4.2.44.3 UModelAPI - ENUMAutolayoutGrowDirectionKind

Enumeration **ENUMAutolayoutGrowDirectionKind**

diagram	
typedElements	Interface ILocalOptionsView 876 Operation AutolayoutHierarchic_GrowDirection 876

16.4.2.44.4 UModelAPI - ENUMCodeLang

Enumeration **ENUMCodeLang**

diagram	
typedElements	Interface IModelTransformationDlg 881 Interface IUMLComponent 1028 Interface IUMLDataAll 911 Interface IUMLPackage 1134 Operation TransformFromLanguage 882 TransformToLanguage 882 Operation CodeLang 1028 Operation CodeLang 917 IsCodeLangNamespace 945 IsCodeLangNamespaceRoot 945 Operation IsCodeLangNamespace 1135 IsCodeLangNamespaceRoot 1135

© 2019-2025 Altova GmbH

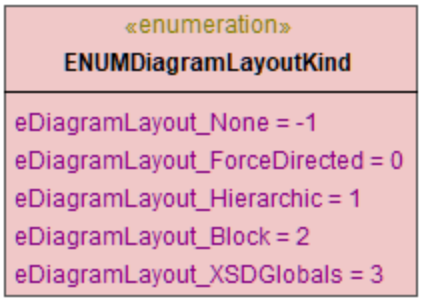
16.4.2.44.5 UModelAPI - ENUMCodeLangVersion

Enumeration **ENUMCodeLangVersion**

diagram	*The diagram is not included because of page size constraints; however, it is available in the HTML version of the manual (https://www.altova.com/manual/en/umodelprofessional/2025.2/).*
typedElements	Interface **IImportSourceDlg** 858 Interface **ILocalOptionsEditing** 873 Interface **IUMLComponent** 1028 Interface **IUMLDataAll** 911 Operation **Language** 859 Operation **ComponentsDefaultCodeLangVersion** 873 Operation **CodeLangVersion** 1028 Operation **CodeLangVersion** 917

16.4.2.44.6 UModelAPI - ENUMDiagramLayoutKind

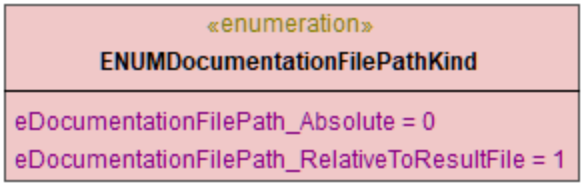
Enumeration **ENUMDiagramLayoutKind**

diagram	
typedElements	Interface **IDiagramWindow** 825 Interface **IImportSourceDlg** 858 Operation **Autolayout** 826 **AutolayoutSelection** 826 Operation **Content_Autolayout** 858 **PackageDependency_Autolayout** 859

 © 2019-2025 Altova GmbH

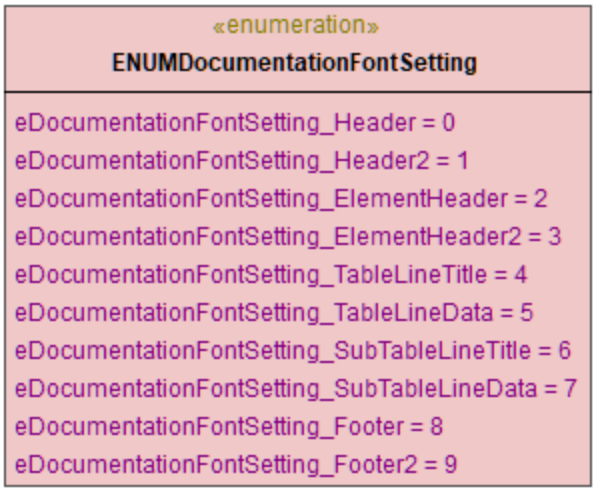
16.4.2.44.7 UModelAPI - ENUMDocumentationFilePathKind

Enumeration **ENUMDocumentationFilePathKind**

diagram	
typedElements	Interface IGenerateDocumentationDlg 841 Operation GenerateLinksToLocalFiles 845

16.4.2.44.8 UModelAPI - ENUMDocumentationFontSetting

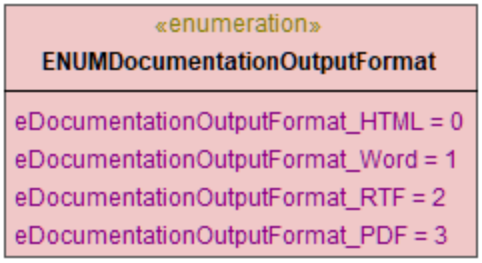
Enumeration **ENUMDocumentationFontSetting**

diagram	
typedElements	Interface IGenerateDocumentationDlg 841 Operation Fonts_GetFace 844 Fonts_GetSize 844 Fonts_GetTextColor 844 Fonts_IsBold 844 Fonts_IsItalic 844 Fonts_IsUnderline 844 Fonts_SetBold 844 Fonts_SetFace 845 Fonts_SetItalic 845 Fonts_SetSize 845 Fonts_SetTextColor 845 Fonts_SetUnderline 845

© 2019-2025 Altova GmbH

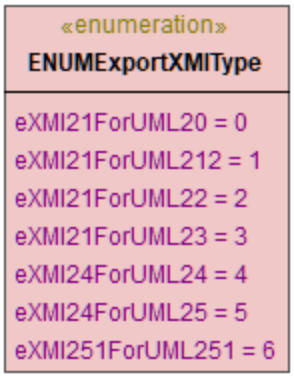
16.4.2.44.9 UModelAPI - ENUMDocumentationOutputFormat

Enumeration **ENUMDocumentationOutputFormat**

diagram	
typedElements	Interface **IGenerateDocumentationDlg** 841 Operation **OutputFormat** 847

16.4.2.44.10 UModelAPI - ENUMExportXMIType

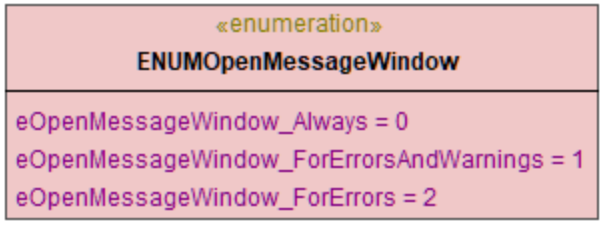
Enumeration **ENUMExportXMIType**

diagram	
typedElements	Interface **IExportXMIFileDlg** 839 Operation **XMIType** 840

 © 2019-2025 Altova GmbH

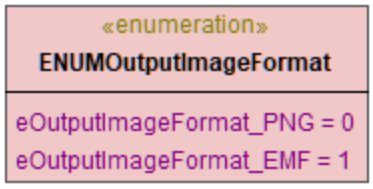
16.4.2.44.11 UModelAPI - ENUMOpenMessageWindow

Enumeration **ENUMOpenMessageWindow**

diagram	
typedElements	Interface **ILocalOptionsCodeEngineering** 868 Operation **OpenMessageWindow** 869

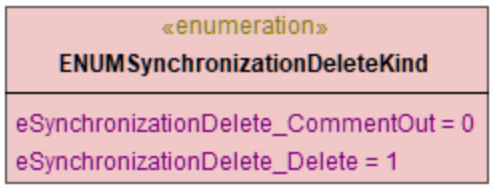
16.4.2.44.12 UModelAPI - ENUMOutputImageFormat

Enumeration **ENUMOutputImageFormat**

diagram	
typedElements	Interface **IDiagramWindow** 825 Interface **IGenerateDocumentationDlg** 841 Operation **SaveDiagramAsImage** 827 Operation **DiagramImageFormat** 843

16.4.2.44.13 UModelAPI - ENUMSynchronizationDeleteKind

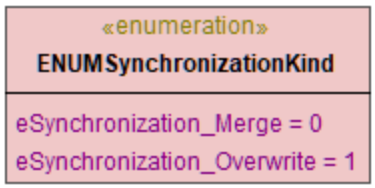
Enumeration **ENUMSynchronizationDeleteKind**

diagram	

typedElements	Interface ISynchronizationSettingsDlg 888	Operation CodeFromModel_DeletingCode 889

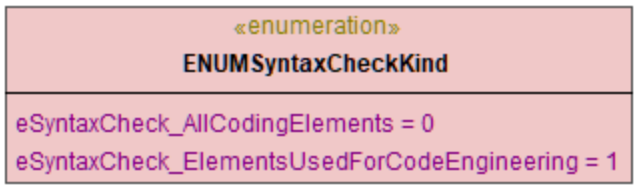
16.4.2.44.14 UModelAPI - ENUMSynchronizationKind

Enumeration **ENUMSynchronizationKind**

diagram	
typedElements	Interface IImportSourceDlg 858 Interface IModelTransformationDlg 881 Interface ISynchronizationSettingsDlg 888 Operation Synchronization 860 Operation Synchronization 882 Operation CodeFromModel_Synchronization 889 ModelFromCode_Synchronization 889

16.4.2.44.15 UModelAPI - ENUMSyntaxCheckKind

Enumeration **ENUMSyntaxCheckKind**

diagram	
typedElements	Interface ILocalOptionsCodeEngineering 868 Operation SyntaxCheck 869

 © 2019-2025 Altova GmbH

16.4.3 UMLData Interfaces

The UMLData interfaces allow direct UML-level access to a UModel document. Using these interfaces, you can read and directly modify the UML representation of the document.

IUMLData 903 is the common base interface of IUMLElement 980 and IUMLGuiElement 1200.

IUMLElements 980 contains elements as defined by the UML specification (see http://www.uml.org).

IUMLGuiElements 1200 contains Altova-specific elements for diagrams, and members used to show IUMLElements 980 on diagrams.

For examples of modifying UML elements and GUI elements, see Object model UMLData 754.

Errors

The `IUMLData` interfaces may return the API error codes listed below.

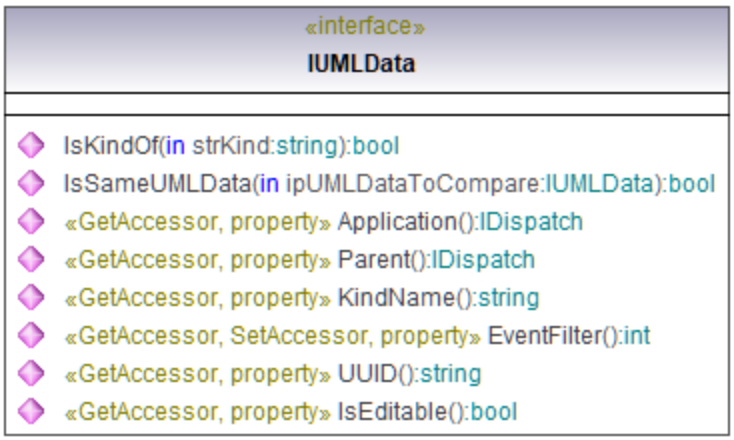
1000	The application object is no longer valid.
1001	Invalid parameter or invalid address for the return parameter was specified.
1002	UModel API is not available in the current edition.
1400	Invalid UMLData modification.
1401	Invalid Waypoint modification.
1402	No changes allowed.
1403	No changes allowed during Undo/Redo.
1404	Element is hidden by Element Style (visibility).
1405	Predefined element not found.
1406	Predefined element is of invalid kind.

For the error codes specific to the UModel API in general, see UModel API Errors 816.

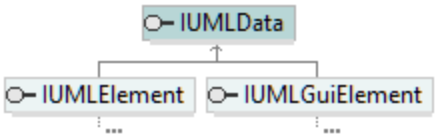
16.4.3.1 UModelAPI - IUMLData

Interface **IUMLData**

diagram



hierarchy



typedElements

Interface _IFocusedUMLDataEvents 894
Interface _IUMLDataEvents 1263

Interface IApplication 818
Interface IDiagramWindow 825
Interface IDocument 832

Interface IUMLCommentTextHyperlink 1026
Interface IUMLData 904
Interface IUMLDataAll 911

Interface IUMLDataList 906
Interface IUMLElement 1050
Interface IUMLGuiDiagram 1211
Interface IUMLGuiRootElement 1234
Interface IUMLGuiTextHyperlink 1250

Interface IUMLHyperlink2Model 1078
Interface IUMLNamedElement 1117

Operation OnFocusedUMLData 894

Operation OnAfterAddChild 1263
OnBeforeErase 1263 OnChanged 1263
OnMoveData 1263

Operation LogMessageWithUMLDataLink 819
Operation FocusedData 826
Operation CanFocusUMLDataInModelTree 833
FocusedUMLData 834
FocusUMLDataInModelTree 834

Operation SetHyperlinkModelElementAddress 1027
Operation IsSameUMLData 905
Operation AddUMLGuiNodeLink 913
FindPredefinedOwnedElement 926
InsertOwnedDiagramAt 939
InsertOwnedHyperlink2ModelAt 940
IsSameUMLData 949
LinkedModelElement 952
SetHyperlinkModelElementAddress 966

Operation ContainsUMLData 910 Item 911
Operation FindPredefinedOwnedElement 1052
Operation AddUMLGuiNodeLink 1212
Operation InsertOwnedDiagramAt 1234
Operation SetHyperlinkModelElementAddress 1251
Operation LinkedModelElement 1078
Operation InsertOwnedHyperlink2ModelAt 1118

© 2019-2025 Altova GmbH

Operation **IUMLData::Application**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IDispatch**			

Operation **IUMLData::EventFilter**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**int**			

Operation **IUMLData::IsEditable**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IUMLData::IsKindOf**

parameter	name	direction	type	type modifier	multiplicity	default
	strKind	**in**	**string**			
	return	**return**	**bool**			

Operation **IUMLData::IsSameUMLData**

parameter	name	direction	type	type modifier	multiplicity	default
	ipUMLDataToCompare	**in**	**IUMLData** 904			
	return	**return**	**bool**			

Operation **IUMLData::KindName**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**string**			

Operation **IUMLData::Parent**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IDispatch**			

Operation **IUMLData::UUID**

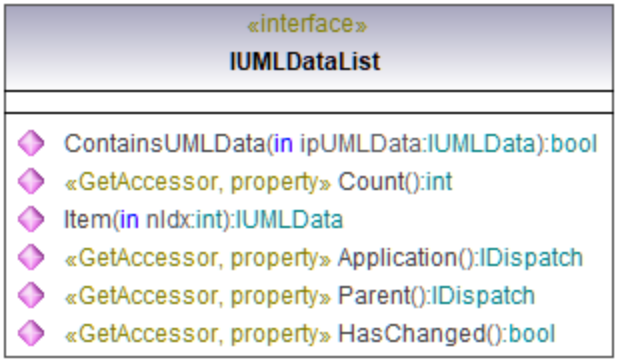
parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**string**			

© 2019-2025 Altova GmbH

16.4.3.2 UModelAPI - IUMLDataList

Interface **IUMLDataList**

diagram



typedElements

Interface IDiagramWindow 825
Interface IUMLAcceptEventAction 981
Interface IUMLAction 983
Interface IUMLActivity 986
Interface IUMLActivityEdge 988
Interface IUMLActivityGroup 991
Interface IUMLActivityNode 992
Interface IUMLActivityPartition 995
Interface IUMLArtifact 998
Interface IUMLAssociation 1000
Interface IUMLBehavior 1002
Interface IUMLBehavioralFeature 1004
Interface IUMLBehavioredClassifier 1006
Interface IUMLCallAction 1008
Interface IUMLClass 1015
Interface IUMLClassifier 1018

Operation SelectedGuiElements 827
Operation ActionTriggers 982, EventActionResults 982
Operation InputPins 983, LocalPostConditions 984, LocalPreConditions 984, OutputPins 984
Operation ActivityEdges 987, ActivityGroups 987, ActivityNodes 987
Operation ActivityPartitions 989, InterruptibleActivityRegions 989, StructuredActivityNodes 990
Operation ContainedEdges 991, ContainedNodes 991, SubGroups 992
Operation IncomingEdges 993, OutgoingEdges 993
Operation Edges 995, Nodes 996, SubPartitions 996
Operation Manifestations 999, NestedArtifacts 999, OwnedAttributes 1000, OwnedOperations 1000
Operation EndTypes 1001, MemberEnds 1001, NavigableOwnedEnds 1001, OwnedEnds 1001
Operation OwnedParameters 1003, Postconditions 1004, Preconditions 1004
Operation Methods 1005, OwnedParameters 1006, RaisedExceptions 1006
Operation InterfaceRealizations 1007, OwnedBehaviors 1007
Operation Results 1009
Operation NestedClassifiers 1016, OwnedOperations 1017, OwnedReceptions 1017, SuperClasses 1017
Operation Attributes 1019, CollaborationUses 1019, Features 1019, Generalizations 1019, Generals 1019, InheritedMembers 1020

© 2019-2025 Altova GmbH

OwnedUseCases 1020 Specifics 1020
UseCases 1021

Interface IUMLClassifierTemplateParameter 1021
Operation ConstrainingClassifiers 1021

Interface IUMLCollaboration 1022
Operation CollaborationRoles 1022
Interface IUMLCollaborationUse 1023
Operation RoleBindings 1024
Interface IUMLCombinedFragment 1024
Operation Operands 1025
Interface IUMLComment 1025
Operation AnnotatedElements 1025
OwnedHyperlinks 1026

Interface IUMLComponent 1028
Operation Realizations 1029
Interface IUMLConnectionPointReference 1031
Operation Entries 1032 Exits 1032
Interface IUMLConstraint 1035
Operation ConstrainedElements 1036
Interface IUMLDataAll 911
Operation ActionTriggers 912 ActivityEdges 912
ActivityGroups 912
ActivityNodes 912
ActivityPartitions 912
ActualGates 912
AllAppliableStereotypes 914
AllWaypoints 914
AnnotatedElements 914
AppliedStereotypes 915
Arguments 915 AttachedNodes 915
Attributes 916
ClientDependencies 917 Clients 917
CollaborationRoles 918
CollaborationUses 918
ConnectionPoints 918
Connections 918
ConstrainedElements 919
ConstrainingClassifiers 919
ContainedEdges 919
ContainedNodes 919 Conveyed 919
DeployedElements 920
Deployments 920 Edges 921
ElementImports 921 EndTypes 921
Entries 921 EventActionResults 925
ExceptionHandlers 925
ExceptionTypes 925 Exits 925
Extends 926 ExtensionLocations 926
ExtensionPoints 926 Features 926
FeaturingClassifiers 926
FormalGates 927 Fragments 927
Generalizations 927 Generals 927
GetOwnedElementsOfKind 928
GuiLinks 929 Handlers 929
ImportedMembers 930 Includes 930
IncomingEdges 930 Incomings 931
InformationFlowRealizations 931
InformationSources 931
InformationTargets 931
InheritedMembers 931
InputElements 931 InputPins 931
InputValues 931 InStates 944
InterfaceRealizations 945
InterruptibleActivityRegions 945
InterruptingEdges 945 Layers 951

		Lifelines 951 LineConnectionWaypoints 951 LineLinks 951 LocalPostConditions 952 LocalPreConditions 952 LowerValues 952 Manifestations 952 MemberEnds 952 Members 953 Messages 953 Methods 953 NavigableOwnedEnds 954 NestedArtifacts 954 NestedClassifiers 954 NestedNodes 955 NestedPackages 955 Nodes 955 Observations 955 Operands 956 OutgoingEdges 956 Outgoings 956 OutputElements 956 OutputPins 956 OutputValues 956 OwnedArguments 957 OwnedAttributes 957 OwnedBehaviors 957 OwnedComments 957 OwnedConnectors 957 OwnedDiagrams 957 OwnedElements 957 OwnedEnds 957 OwnedGuiElements 958 OwnedGuiNodeLinks 958 OwnedHyperlinks 958 OwnedLiterals 958 OwnedMembers 958 OwnedOperations 958 OwnedParameters 958 OwnedPorts 958 OwnedReceptions 958 OwnedRules 958 OwnedStereotypes 958 OwnedTemplateBindings 959 OwnedTemplateParameters 959 OwnedTypes 959 OwnedUseCases 959 PackagedElements 961 PackageImports 961 PackageMerges 961 ParameterSubstitutions 961 Postconditions 962 Preconditions 962 ProfileApplications 962 Qualifiers 963 RaisedExceptions 963 Realizations 963 RealizingConnectors 963 Referred 964 Regions 964 RelatedElements 964 RelativeNodes 964 Results 964 RoleBindings 965 Slots 972 Sources 973 Specifics 973 StereotypeApplications 974 StructuredActivityNodes 974 SubGroups 974 Subjects 974 SubmachineStates 974 SubPartitions 974 SubVertices 975

© 2019-2025 Altova GmbH

SuperClasses 975
SupplierDependencies 975
Suppliers 975 Targets 975
TextLabels 976 Transitions 977
Triggers 977 TypedElements 977
UpperValues 978 UseCases 978
Values 979 Waypoints 979

Interface IUMLDataType 1039
Operation OwnedAttributes 1040
OwnedOperations 1040

Interface IUMLDependency 1041
Operation Clients 1042 Suppliers 1042

Interface IUMLDeploymentTarget 1044
Operation DeployedElements 1044
Deployments 1044

Interface IUMLDirectedRelationship 1046
Operation Sources 1046 Targets 1046

Interface IUMLDuration 1047
Operation Observations 1047

Interface IUMLElement 1050
Operation AllAppliableStereotypes 1051
AppliedStereotypes 1051
GetOwnedElementsOfKind 1052
OwnedComments 1053
OwnedElements 1053
StereotypeApplications 1053

Interface IUMLEncapsulatedClassifier 1055
Operation OwnedPorts 1055

Interface IUMLEnumeration 1056
Operation OwnedLiterals 1057

Interface IUMLExceptionHandler 1059
Operation ExceptionTypes 1060

Interface IUMLExecutableNode 1061
Operation Handlers 1061

Interface IUMLExpansionRegion 1065
Operation InputElements 1066
OutputElements 1066

Interface IUMLExpression 1066
Operation Operands 1067

Interface IUMLExtend 1067
Operation ExtensionLocations 1068

Interface IUMLFeature 1069
Operation FeaturingClassifiers 1070

Interface IUMLGuiDiagram 1211
Operation GuiLinks 1214 Layers 1214
OwnedHyperlinks 1214

Interface IUMLGuiElement 1216
Operation OwnedGuiElements 1217

Interface IUMLGuiLineLink 1222
Operation AllWaypoints 1222
LineConnectionWaypoints 1223
Waypoints 1223

Interface IUMLGuiLink 1224
Operation AttachedNodes 1224
RelativeNodes 1225

Interface IUMLGuiNodeLink 1226
Operation OwnedGuiNodeLinks 1227

Interface IUMLGuiNote 1228
Operation OwnedHyperlinks 1229

Interface IUMLGuiRootElement 1234
Operation OwnedDiagrams 1235

Interface IUMLGuiTextLabelWaypoint 1253
Operation TextLabels 1254

Interface IUMLGuiWaypoint 1261
Operation LineLinks 1261

Interface IUMLInformationFlow 1080
Operation Conveyed 1080
InformationFlowRealizations 1081
InformationSources 1081
InformationTargets 1081
RealizingConnectors 1082

Interface IUMLInstanceSpecification 1085
Operation Slots 1086

Interface IUMLInteraction 1088
Operation FormalGates 1088 Fragments 1089
Lifelines 1089 Messages 1089

Interface IUMLInteractionUse 1092
Operation ActualGates 1093

Interface IUMLInterface 1094
Operation NestedClassifiers 1096
OwnedAttributes 1096
OwnedOperations 1096
OwnedReceptions 1096

	Interface IUMLInterruptibleActivityRegion 1098	Operation InterruptingEdges 1099 Nodes 1099
	Interface IUMLInvocationAction 1101	Operation Arguments 1102
	Interface IUMLMessage 1111	Operation OwnedArguments 1112
	Interface IUMLMultiplicityElement 1116	Operation LowerValues 1117 UpperValues 1117
	Interface IUMLNamedElement 1117	Operation ClientDependencies 1118
		OwnedHyperlinks 1119
		SupplierDependencies 1119
	Interface IUMLNamespace 1120	Operation ElementImports 1120
		ImportedMembers 1120 Members 1121
		OwnedMembers 1121
		OwnedRules 1121
		PackageImports 1122
		PackageMerges 1122
	Interface IUMLNode 1122	Operation NestedNodes 1123
	Interface IUMLObjectNode 1124	Operation ExceptionHandlers 1125 InStates 1125
	Interface IUMLOpaqueAction 1128	Operation InputValues 1128 OutputValues 1129
	Interface IUMLPackage 1134	Operation NestedPackages 1135
		OwnedStereotypes 1136
		OwnedTypes 1136
		PackagedElements 1136
		ProfileApplications 1136
	Interface IUMLProperty 1147	Operation Qualifiers 1149
	Interface IUMLProtocolTransition 1151	Operation Referred 1152
	Interface IUMLRegion 1157	Operation SubVertices 1158 Transitions 1158
	Interface IUMLRelationship 1158	Operation RelatedElements 1159
	Interface IUMLSignal 1160	Operation OwnedAttributes 1161
	Interface IUMLSlot 1162	Operation Values 1163
	Interface IUMLState 1163	Operation ConnectionPoints 1164
		Connections 1164 Regions 1165
	Interface IUMLStateMachine 1167	Operation ConnectionPoints 1168 Regions 1168
		SubmachineStates 1168
	Interface IUMLStructuredActivityNode 1173	Operation Edges 1173 Nodes 1174
	Interface IUMLStructuredClassifier 1174	Operation OwnedAttributes 1175
		OwnedConnectors 1175
	Interface IUMLTemplateableElement 1176	Operation OwnedTemplateBindings 1176
	Interface IUMLTemplateBinding 1177	Operation ParameterSubstitutions 1178
	Interface IUMLTemplateSignature 1181	Operation OwnedTemplateParameters 1181
	Interface IUMLTimeExpression 1184	Operation Observations 1185
	Interface IUMLTransition 1187	Operation Triggers 1188
	Interface IUMLType 1189	Operation TypedElements 1190
	Interface IUMLUseCase 1192	Operation Extends 1193 ExtensionPoints 1193
		Includes 1193 Subjects 1193
	Interface IUMLVertex 1199	Operation Incomings 1199 Outgoings 1199

Operation **IUMLDataList::Application**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IDispatch**			

Operation **IUMLDataList::ContainsUMLData**

parameter	name	direction	type	type modifier	multiplicity	default
	ipUMLData	**in**	IUMLData 904			

© 2019-2025 Altova GmbH

	return	**return**	**bool**

Operation **IUMLDataList::Count**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**int**			

Operation **IUMLDataList::HasChanged**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IUMLDataList::Item**

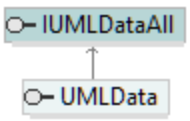
parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	**IUMLData** 904			

Operation **IUMLDataList::Parent**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IDispatch**			

16.4.3.3 UModelAPI - IUMLDataAll

Interface **IUMLDataAll**

diagram	*The diagram is not included because of page size constraints; however, it is available in the HTML version of the manual (https://www.altova.com/manual/en/umodelprofessional/2025.2/).*
hierarchy	

Operation **IUMLDataAll::Abstraction**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLComponent** 1028			

Operation **IUMLDataAll::ActionContext**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLClassifier** 1018			

Operation **IUMLDataAll::ActionInputPin**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLActionInputPin 985			

Operation **IUMLDataAll::ActionTriggers**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLDataList 906			

Operation **IUMLDataAll::ActionValue**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLValueSpecification 1195			

Operation **IUMLDataAll::ActiveLayer**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLGuiDiagramLayer 1215			

Operation **IUMLDataAll::Activity**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLActivity 986			

Operation **IUMLDataAll::ActivityEdges**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLDataList 906			

Operation **IUMLDataAll::ActivityGroups**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLDataList 906			

Operation **IUMLDataAll::ActivityNodes**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLDataList 906			

Operation **IUMLDataAll::ActivityPartitions**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLDataList 906			

Operation **IUMLDataAll::Actual**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLParameterableElement 1141			

Operation **IUMLDataAll::ActualGates**

parameter	name	direction	type	type modifier	multiplicity	default

© 2019-2025 Altova GmbH

	return	**return**	**IUMLDataList** 906

Operation **IUMLDataAll::Addition**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLUseCase** 1192			

Operation **IUMLDataAll::AddOwnedGuiNodeLink**

parameter	name	direction	type	type modifier	multiplicity	default
	ipForUMLData	**in**	**IUMLGuiNodeLink** 1226			
	return	**return**	**void**			

Operation **IUMLDataAll::AddUMLElement**

parameter	name	direction	type	type modifier	multiplicity	default
	strKind	**in**	**string**			
	nLeft	**in**	**int**			
	nTop	**in**	**int**			
	return	**return**	**IUMLGuiNodeLink** 1226			

Operation **IUMLDataAll::AddUMLGuiContainmentLink**

parameter	name	direction	type	type modifier	multiplicity	default
	ipFromLink	**in**	**IUMLGuiLink** 1224			
	ipToLink	**in**	**IUMLGuiLink** 1224			
	return	**return**	**IUMLGuiContainmentLink** 1210			

Operation **IUMLDataAll::AddUMLGuiNodeLink**

parameter	name	direction	type	type modifier	multiplicity	default
	ipForUMLData	**in**	**IUMLData** 904			
	nLeft	**in**	**int**			
	nTop	**in**	**int**			
	return	**return**	**IUMLGuiNodeLink** 1226			

Operation **IUMLDataAll::AddUMLGuiNote**

parameter	name	direction	type	type modifier	multiplicity	default
	nLeft	**in**	**int**			
	nTop	**in**	**int**			
	return	**return**	**IUMLGuiNote** 1228			

Operation **IUMLDataAll::AddUMLGuiNoteLink**

parameter	name	direction	type	type modifier	multiplicity	default
	ipFromNote	**in**	**IUMLGuiNote** 1228			
	ipToLink	**in**	**IUMLGuiNodeLink** 1226			
	return	**return**	**IUMLGuiNoteLink** 1229			

Operation **IUMLDataAll::AddUMLGuiNoteLinkToLine**

parameter	name	direction	type	type modifier	multiplicity	default
	ipFromNote	**in**	IUMLGuiNote 1228			
	ipToLink	**in**	IUMLGuiLineLink 1222			
	nDistanceFromLineBegin	**in**	**int**			
	return	**return**	IUMLGuiNoteLink 1229			

Operation **IUMLDataAll::AddUMLLineElement**

parameter	name	direction	type	type modifier	multiplicity	default
	strKind	**in**	**string**			
	ipFromNode	**in**	IUMLGuiNodeLink 1226			
	ipToNode	**in**	IUMLGuiNodeLink 1226			
	return	**return**	IUMLGuiLineLink 1222			

Operation **IUMLDataAll::Aggregation**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	ENUMUMLAggregationKind 1264			

Operation **IUMLDataAll::Alias**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**string**			

Operation **IUMLDataAll::AllAppliableStereotypes**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLDataList 906			

Operation **IUMLDataAll::AllowSubstitutable**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IUMLDataAll::AllWaypoints**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLDataList 906			

Operation **IUMLDataAll::AnnotatedElements**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLDataList 906			

Operation **IUMLDataAll::Application**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IDispatch**			

© 2019-2025 Altova GmbH

Operation **IUMLDataAll::AppliedElement**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLElement** 1050			

Operation **IUMLDataAll::AppliedProfile**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLProfile** 1145			

Operation **IUMLDataAll::AppliedStereotypes**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			

Operation **IUMLDataAll::ApplyingPackage**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLPackage** 1134			

Operation **IUMLDataAll::ApplyPredefinedStereotype**

parameter	name	direction	type	type modifier	multiplicity	default
	nStereotype	**in**	**ENUMUMLPredefinedElement** 1271			
	return	**return**	**IUMLStereotypeApplication** 1170			

Operation **IUMLDataAll::ApplyStereotype**

parameter	name	direction	type	type modifier	multiplicity	default
	ipStereotype	**in**	**IUMLStereotype** 1169			
	return	**return**	**IUMLStereotypeApplication** 1170			

Operation **IUMLDataAll::Arguments**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			

Operation **IUMLDataAll::Association**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLAssociation** 1000			

Operation **IUMLDataAll::AssociationEnd**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLProperty** 1147			

Operation **IUMLDataAll::AttachedNodes**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			

Operation **IUMLDataAll::AttachedTo**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLGuiLink** 1224			

Operation **IUMLDataAll::Attributes**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			

Operation **IUMLDataAll::BaseClass**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**string**			

Operation **IUMLDataAll::BeginOffset**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**int**			

Operation **IUMLDataAll::Behavior**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLBehavior** 1002			

Operation **IUMLDataAll::BehaviorExecution**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLBehavior** 1002			

Operation **IUMLDataAll::BehaviorSpecification**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLBehavioralFeature** 1004			

Operation **IUMLDataAll::Body**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**string**			

Operation **IUMLDataAll::BooleanValue**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IUMLDataAll::Bottom**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**int**			

Operation **IUMLDataAll::BoundElement**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLTemplateableElement** 1176			

© 2019-2025 Altova GmbH

Operation **IUMLDataAll::CallOperation**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLOperation 1131			

Operation **IUMLDataAll::CallTarget**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLInputPin 1083			

Operation **IUMLDataAll::ChangeExpression**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLValueSpecification 1195			

Operation **IUMLDataAll::Class**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLClass 1015			

Operation **IUMLDataAll::Classifier**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLClassifier 1018			

Operation **IUMLDataAll::ClientDependencies**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLDataList 906			

Operation **IUMLDataAll::Clients**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLDataList 906			

Operation **IUMLDataAll::CodeFileNameCount**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**int**			

Operation **IUMLDataAll::CodeLang**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	ENUMCodeLang 897			

Operation **IUMLDataAll::CodeLangVersion**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	ENUMCodeLangVersion 898			

Operation **IUMLDataAll::CodeOperation**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLOperation 1131			

Operation **IUMLDataAll::CodeProjectFileOrDirectory**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**string**			

Operation **IUMLDataAll::CollaborationRoles**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLDataList 906			

Operation **IUMLDataAll::CollaborationType**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLCollaboration 1022			

Operation **IUMLDataAll::CollaborationUses**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLDataList 906			

Operation **IUMLDataAll::Comment**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**string**			

Operation **IUMLDataAll::Concurrency**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	ENUMUMLCallConcurrencyKind 1264			

Operation **IUMLDataAll::ConnectionPoints**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLDataList 906			

Operation **IUMLDataAll::Connections**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLDataList 906			

Operation **IUMLDataAll::ConnectorKind**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	ENUMUMLConnectorKind 1265			

Operation **IUMLDataAll::ConnectorType**

parameter	name	direction	type	type modifier	multiplicity	default

© 2019-2025 Altova GmbH

	return	**return**	IUMLAssociation 1000

Operation **IUMLDataAll::ConstrainedElements**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLDataList 906			

Operation **IUMLDataAll::ConstrainingClassifiers**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLDataList 906			

Operation **IUMLDataAll::ConstrainingPointX**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**int**			

Operation **IUMLDataAll::ConstrainingPointY**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**int**			

Operation **IUMLDataAll::ContainedEdges**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLDataList 906			

Operation **IUMLDataAll::ContainedNodes**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLDataList 906			

Operation **IUMLDataAll::Container**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLRegion 1157			

Operation **IUMLDataAll::Context**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLNamespace 1120			

Operation **IUMLDataAll::Contract**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLInterface 1094			

Operation **IUMLDataAll::ContrainingAreaIndex**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**int**			

Operation **IUMLDataAll::Conveyed**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			

Operation **IUMLDataAll::Covered**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLLifeline** 1104			

Operation **IUMLDataAll::Datatype**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataType** 1039			

Operation **IUMLDataAll::DecisionInput**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLBehavior** 1002			

Operation **IUMLDataAll::Default**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**string**			

Operation **IUMLDataAll::DefaultLinkName**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**string**			

Operation **IUMLDataAll::DefaultParamValue**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**string**			

Operation **IUMLDataAll::DefaultValue**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLValueSpecification** 1195			

Operation **IUMLDataAll::DefiningFeature**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLStructuralFeature** 1171			

Operation **IUMLDataAll::DeployedElements**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			

Operation **IUMLDataAll::Deployments**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			

© 2019-2025 Altova GmbH

Operation **IUMLDataAll::Direction**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**ENUMUMLParameterDirectionKind** 1271			

Operation **IUMLDataAll::DistanceFromLineBegin**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**int**			

Operation **IUMLDataAll::DoActivity**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLBehavior** 1002			

Operation **IUMLDataAll::Edges**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			

Operation **IUMLDataAll::Effect**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLBehavior** 1002			

Operation **IUMLDataAll::Element**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLElement** 1050			

Operation **IUMLDataAll::ElementImports**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			

Operation **IUMLDataAll::EndOffset**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**int**			

Operation **IUMLDataAll::EndTypes**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			

Operation **IUMLDataAll::Entries**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			

Operation **IUMLDataAll::Entry**

parameter	name	direction	type	type modifier	multiplicity	default

	return	**return**	**IUMLBehavior** 1002

Operation **IUMLDataAll::Enumeration**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLEnumeration** 1056			

Operation **IUMLDataAll::EraseAnnotatedElementAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	**void**			

Operation **IUMLDataAll::EraseCodeFileNameAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	**void**			

Operation **IUMLDataAll::EraseCollaborationRoleAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	**void**			

Operation **IUMLDataAll::EraseConstrainedElementAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	**void**			

Operation **IUMLDataAll::EraseConstrainingClassifierAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	**void**			

Operation **IUMLDataAll::EraseConveyedAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	**void**			

Operation **IUMLDataAll::EraseCoveredByAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	**void**			

Operation **IUMLDataAll::EraseEdgeAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	**void**			

Operation **IUMLDataAll::EraseEntryAt**

© 2019-2025 Altova GmbH

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	**void**			

Operation **IUMLDataAll::EraseExceptionTypeAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	**void**			

Operation **IUMLDataAll::EraseExitAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	**void**			

Operation **IUMLDataAll::EraseExtensionLocationAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	**void**			

Operation **IUMLDataAll::EraseFromDiagram**

parameter	name	direction	type	type modifier	multiplicity	default
	ipVal	**in**	**IUMLGuiElement** 1216			
	return	**return**	**void**			

Operation **IUMLDataAll::EraseFromModel**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**void**			

Operation **IUMLDataAll::EraseInformationFlowRealizationAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	**void**			

Operation **IUMLDataAll::EraseInformationSourceAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	**void**			

Operation **IUMLDataAll::EraseInformationTargetAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	**void**			

Operation **IUMLDataAll::EraseInputElementAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	**void**			

Operation **IUMLDataAll::EraseInStateAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	**void**			

Operation **IUMLDataAll::EraseInterruptingEdgeAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	**void**			

Operation **IUMLDataAll::EraseNodeAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	**void**			

Operation **IUMLDataAll::EraseObservationAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	**void**			

Operation **IUMLDataAll::EraseOutputElementAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	**void**			

Operation **IUMLDataAll::EraseRaisedExceptionAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	**void**			

Operation **IUMLDataAll::EraseRealizingConnectorAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	**void**			

Operation **IUMLDataAll::EraseSubjectAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	**void**			

Operation **IUMLDataAll::EraseWaypointAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	**void**			

Operation **IUMLDataAll::Event**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLEvent** 1058			

© 2019-2025 Altova GmbH

Operation **IUMLDataAll::EventActionResults**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLDataList 906			

Operation **IUMLDataAll::EventFilter**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**int**			

Operation **IUMLDataAll::ExceptionHandlers**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLDataList 906			

Operation **IUMLDataAll::ExceptionInput**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLObjectNode 1124			

Operation **IUMLDataAll::ExceptionTypes**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLDataList 906			

Operation **IUMLDataAll::ExecutionSpecificationFinish**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLExecutionSpecification 1062			

Operation **IUMLDataAll::ExecutionSpecificationStart**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLExecutionSpecification 1062			

Operation **IUMLDataAll::Exit**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLBehavior 1002			

Operation **IUMLDataAll::Exits**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLDataList 906			

Operation **IUMLDataAll::Expr**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLValueSpecification 1195			

Operation **IUMLDataAll::Expression**

parameter	name	direction	type	type modifier	multiplicity	default

	return	**return**	**IUMLExpression** 1066

Operation **IUMLDataAll::ExtendedCase**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLUseCase** 1192			

Operation **IUMLDataAll::Extends**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			

Operation **IUMLDataAll::Extension**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLUseCase** 1192			

Operation **IUMLDataAll::ExtensionLocations**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			

Operation **IUMLDataAll::ExtensionPoints**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			

Operation **IUMLDataAll::Features**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			

Operation **IUMLDataAll::FeaturingClassifiers**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			

Operation **IUMLDataAll::FileName**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**string**			

Operation **IUMLDataAll::FindOwnedMemberWithQualifiedName**

parameter	name	direction	type	type modifier	multiplicity	default
	strName	**in**	**string**			
	return	**return**	**IUMLNamedElement** 1117			

Operation **IUMLDataAll::FindPredefinedOwnedElement**

parameter	name	direction	type	type modifier	multiplicity	default
	nElement	**in**	**ENUMUMLPredefinedElement** 1271			
	bRecursive	**in**	**bool**			
	return	**return**	**IUMLData** 904			

© 2019-2025 Altova GmbH

Operation **IUMLDataAll::Finish**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLOccurrenceSpecification** 1127			

Operation **IUMLDataAll::Formal**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLTemplateParameter** 1178			

Operation **IUMLDataAll::FormalGates**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			

Operation **IUMLDataAll::Fragments**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			

Operation **IUMLDataAll::General**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLClassifier** 1018			

Operation **IUMLDataAll::Generalizations**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			

Operation **IUMLDataAll::Generals**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			

Operation **IUMLDataAll::GeneralValueLifelineNameCompartmentEndPos**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**int**			

Operation **IUMLDataAll::GetCodeFileName**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	**string**			

Operation **IUMLDataAll::GetCodeFilePath**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	**string**			

Operation **IUMLDataAll::GetHSeparatorPosition**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	**int**			

Operation **IUMLDataAll::GetMultiplicity**

parameter	name	direction	type	type modifier	multiplicity	default
	bWithBrackets	**in**	**bool**			
	return	**return**	**string**			

Operation **IUMLDataAll::GetOperation**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLOperation** 1131			

Operation **IUMLDataAll::GetOwnedElementsOfKind**

parameter	name	direction	type	type modifier	multiplicity	default
	strKind	**in**	**string**			
	bRecursive	**in**	**bool**			
	return	**return**	**IUMLDataList** 906			

Operation **IUMLDataAll::GetSeparatorPosition**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	**int**			

Operation **IUMLDataAll::GetSourceLifeline**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLLifeline** 1104			

Operation **IUMLDataAll::GetStateIndex**

parameter	name	direction	type	type modifier	multiplicity	default
	nTimeTickIndex	**in**	**int**			
	return	**return**	**int**			

Operation **IUMLDataAll::GetStereotypeApplicationForPredefinedStereotype**

parameter	name	direction	type	type modifier	multiplicity	default
	nElement	**in**	**ENUMUMLPredefinedElement** 1271			
	return	**return**	**IUMLStereotypeApplication** 1170			

Operation **IUMLDataAll::GetStereotypeApplicationForStereotype**

parameter	name	direction	type	type modifier	multiplicity	default
	ipStereotype	**in**	**IUMLStereotype** 1169			
	return	**return**	**IUMLStereotypeApplication** 1170			

Operation **IUMLDataAll::GetTargetLifeline**

© 2019-2025 Altova GmbH

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLLifeline** 1104			

Operation **IUMLDataAll::GetTextLabelText**

parameter	name	direction	type	type modifier	multiplicity	default
	ipTextLabel	**in**	**IUMLGuiTextLabel** 1251			
	return	**return**	**string**			

Operation **IUMLDataAll::GetTimeTickLength**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	**int**			

Operation **IUMLDataAll::GetVisualStatePosition**

parameter	name	direction	type	type modifier	multiplicity	default
	nStateIndex	**in**	**int**			
	return	**return**	**int**			

Operation **IUMLDataAll::GetVSeparatorPosition**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	**int**			

Operation **IUMLDataAll::Guard**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**string**			

Operation **IUMLDataAll::GuiLinks**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			

Operation **IUMLDataAll::GuiOwner**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLGuiElement** 1216			

Operation **IUMLDataAll::HandlerBody**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLExecutableNode** 1061			

Operation **IUMLDataAll::Handlers**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			

Operation **IUMLDataAll::HSeparatorCount**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**int**			

Operation **IUMLDataAll::IconFileName**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**string**			

Operation **IUMLDataAll::ImplementingClassifier**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLBehavioredClassifier** 1006			

Operation **IUMLDataAll::ImportedElement**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLPackageableElement** 1137			

Operation **IUMLDataAll::ImportedMembers**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			

Operation **IUMLDataAll::ImportedPackage**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLPackage** 1134			

Operation **IUMLDataAll::ImportingNamespace**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLNamespace** 1120			

Operation **IUMLDataAll::InActivity**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLActivity** 986			

Operation **IUMLDataAll::Includes**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			

Operation **IUMLDataAll::IncludingCase**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLUseCase** 1192			

Operation **IUMLDataAll::IncomingEdges**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			

© 2019-2025 Altova GmbH

Operation **IUMLDataAll::Incomings**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			

Operation **IUMLDataAll::InformationFlowRealizations**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			

Operation **IUMLDataAll::InformationSources**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			

Operation **IUMLDataAll::InformationTargets**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			

Operation **IUMLDataAll::InheritedMembers**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			

Operation **IUMLDataAll::InputElements**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			

Operation **IUMLDataAll::InputPins**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			

Operation **IUMLDataAll::InputValues**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			

Operation **IUMLDataAll::InsertActionTriggerAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	**IUMLTrigger** 1188			

Operation **IUMLDataAll::InsertActivityEdgeAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	strKind	**in**	**string**			
	ipFrom	**in**	**IUMLActivityNode** 992			
	ipTo	**in**	**IUMLActivityNode** 992			
	return	**return**	**IUMLActivityEdge** 988			

Operation **IUMLDataAll::InsertActivityGroupAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	strKind	**in**	**string**			
	return	**return**	**IUMLActivityGroup** 991			

Operation **IUMLDataAll::InsertActivityNodeAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	strKind	**in**	**string**			
	return	**return**	**IUMLActivityNode** 992			

Operation **IUMLDataAll::InsertActualGateAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	**IUMLGate** 1074			

Operation **IUMLDataAll::InsertAnnotatedElementAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	ipVal	**in**	**IUMLElement** 1050			
	return	**return**	**void**			

Operation **IUMLDataAll::InsertArgumentAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	**IUMLInputPin** 1083			

Operation **IUMLDataAll::InsertArgumentOfKindAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	strKind	**in**	**string**			
	return	**return**	**IUMLInputPin** 1083			

Operation **IUMLDataAll::InsertCodeFileNameAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	strNewVal	**in**	**string**			
	return	**return**	**void**			

Operation **IUMLDataAll::InsertCollaborationRoleAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	ipVal	**in**	**IUMLConnectableElement** 1030			
	return	**return**	**void**			

© 2019-2025 Altova GmbH

Operation **IUMLDataAll::InsertCollaborationUseAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	**IUMLCollaborationUse** 1023			

Operation **IUMLDataAll::InsertConnectionAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	**IUMLConnectionPointReference** 1031			

Operation **IUMLDataAll::InsertConnectionPointAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	**IUMLPseudostate** 1153			

Operation **IUMLDataAll::InsertConstrainedElementAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	ipVal	**in**	**IUMLElement** 1050			
	return	**return**	**void**			

Operation **IUMLDataAll::InsertConstrainingClassifierAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	ipVal	**in**	**IUMLClassifier** 1018			
	return	**return**	**void**			

Operation **IUMLDataAll::InsertConveyedAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	ipVal	**in**	**IUMLClassifier** 1018			
	return	**return**	**void**			

Operation **IUMLDataAll::InsertCoveredByAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	ipVal	**in**	**IUMLInteractionFragment** 1091			
	return	**return**	**void**			

Operation **IUMLDataAll::InsertDeploymentAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	ipDeployedArtifact	**in**	**IUMLDeployedArtifact** 1042			

	return	**return**	**IUMLDeployment** 1043

Operation **IUMLDataAll::InsertEdgeAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	ipEdge	**in**	**IUMLActivityEdge** 988			
	return	**return**	**void**			

Operation **IUMLDataAll::InsertElementImportAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	ipImportedEleme nt	**in**	**IUMLPackageable Element** 1137			
	return	**return**	**IUMLElementImp ort** 1054			

Operation **IUMLDataAll::InsertEntryAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	ipVal	**in**	**IUMLPseudostate** 1153			
	return	**return**	**void**			

Operation **IUMLDataAll::InsertEventActionResultAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	**IUMLOutputPin** 1133			

Operation **IUMLDataAll::InsertExceptionTypeAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	ipVal	**in**	**IUMLClassifier** 1018			
	return	**return**	**void**			

Operation **IUMLDataAll::InsertExitAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	ipVal	**in**	**IUMLPseudostate** 1153			
	return	**return**	**void**			

Operation **IUMLDataAll::InsertExtendAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	ipExtendedCase	**in**	**IUMLUseCase** 1192			
	return	**return**	**IUMLExtend** 1067			

© 2019-2025 Altova GmbH

Operation **IUMLDataAll::InsertExtensionLocationAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	pExtensionLocation	**in**	**IUMLExtensionPoint** 1069			
	return	**return**	**void**			

Operation **IUMLDataAll::InsertExtensionPointAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	**IUMLExtensionPoint** 1069			

Operation **IUMLDataAll::InsertFormalGateAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	**IUMLGate** 1074			

Operation **IUMLDataAll::InsertFragmentAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	strKind	**in**	**string**			
	return	**return**	**IUMLInteractionFragment** 1091			

Operation **IUMLDataAll::InsertGeneralizationAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	ipGeneral	**in**	**IUMLClassifier** 1018			
	return	**return**	**IUMLGeneralization** 1074			

Operation **IUMLDataAll::InsertHandlerAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	**IUMLExceptionHandler** 1059			

Operation **IUMLDataAll::InsertIncludeAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	ipIncludingCase	**in**	**IUMLUseCase** 1192			
	return	**return**	**IUMLInclude** 1079			

Operation **IUMLDataAll::InsertInformationFlowRealizationAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	ipVal	**in**	**IUMLRelationship** 1153			
	return	**return**	**void**			

Operation **IUMLDataAll::InsertInformationSourceAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	ipVal	**in**	**IUMLNamedElement** 1117			
	return	**return**	**void**			

Operation **IUMLDataAll::InsertInformationTargetAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	ipVal	**in**	**IUMLNamedElement** 1117			
	return	**return**	**void**			

Operation **IUMLDataAll::InsertInputElementAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	ipNode	**in**	**IUMLExpansionNode** 1063			
	return	**return**	**void**			

Operation **IUMLDataAll::InsertInputValueAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	**IUMLInputPin** 1083			

Operation **IUMLDataAll::InsertInStateAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	ipVal	**in**	**IUMLState** 1163			
	return	**return**	**void**			

Operation **IUMLDataAll::InsertInterfaceRealizationAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	ipContract	**in**	**IUMLInterface** 1094			
	return	**return**	**IUMLInterfaceRealization** 1097			

Operation **IUMLDataAll::InsertInterruptingEdgeAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	ipEdge	**in**	**IUMLActivityEdge** 988			
	return	**return**	**void**			

Operation **IUMLDataAll::InsertLayerAt**

parameter	name	direction	type	type modifier	multiplicity	default

© 2019-2025 Altova GmbH

	nIdx	**in**	**int**
	return	**return**	**IUMLGuiDiagramLayer** 1215

Operation **IUMLDataAll::InsertLifelineAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	**IUMLLifeline** 1104			

Operation **IUMLDataAll::InsertLocalPostConditionAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	**IUMLConstraint** 1035			

Operation **IUMLDataAll::InsertLocalPreConditionAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	**IUMLConstraint** 1035			

Operation **IUMLDataAll::InsertLowerUpperValueAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	strLower	**in**	**string**			
	strUpper	**in**	**string**			
	return	**return**	**void**			

Operation **IUMLDataAll::InsertManifestationAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	ipUtilizedElement	**in**	**IUMLPackageableElement** 1137			
	return	**return**	**IUMLManifestation** 1109			

Operation **IUMLDataAll::InsertMessageAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	**IUMLMessage** 1111			

Operation **IUMLDataAll::InsertNestedArtifactAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	**IUMLArtifact** 998			

Operation **IUMLDataAll::InsertNestedClassifierAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			

	strKind	**in**	**string**
	return	**return**	**IUMLClassifier** 1018

Operation **IUMLDataAll::InsertNestedNodeAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	strKind	**in**	**string**			
	return	**return**	**IUMLNode** 1122			

Operation **IUMLDataAll::InsertNodeAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	ipVal	**in**	**IUMLActivityNode** 992			
	return	**return**	**void**			

Operation **IUMLDataAll::InsertObservationAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	ipVal	**in**	**IUMLObservation** 1126			
	return	**return**	**void**			

Operation **IUMLDataAll::InsertOperandAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	**IUMLInteractionOperand** 1091			

Operation **IUMLDataAll::InsertOutputElementAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	ipNode	**in**	**IUMLExpansionNode** 1063			
	return	**return**	**void**			

Operation **IUMLDataAll::InsertOutputValueAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	**IUMLOutputPin** 1133			

Operation **IUMLDataAll::InsertOwnedArgumentAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	strKind	**in**	**string**			
	return	**return**	**IUMLValueSpecification** 1195			

Operation **IUMLDataAll::InsertOwnedAttributeAt**

© 2019-2025 Altova GmbH

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	**IUMLProperty** 1147			

Operation **IUMLDataAll::InsertOwnedBehaviorAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	strKind	**in**	**string**			
	return	**return**	**IUMLBehavior** 1002			

Operation **IUMLDataAll::InsertOwnedCommentAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	**IUMLComment** 1025			

Operation **IUMLDataAll::InsertOwnedCommentTextHyperlinkAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nFromTextPos	**in**	**int**			
	nToTextPos	**in**	**int**			
	strAddress	**in**	**string**			
	return	**return**	**IUMLCommentTextHyperlink** 1026			

Operation **IUMLDataAll::InsertOwnedConnectorAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	ipFrom	**in**	**IUMLConnectableElement** 1030			
	ipTo	**in**	**IUMLConnectableElement** 1030			
	return	**return**	**IUMLConnector** 1033			

Operation **IUMLDataAll::InsertOwnedDiagramAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	ipUMLParent	**in**	**IUMLData** 904			
	strKind	**in**	**string**			
	return	**return**	**IUMLGuiDiagram** 1211			

Operation **IUMLDataAll::InsertOwnedGuiTextHyperlinkAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nFromTextPos	**in**	**int**			
	nToTextPos	**in**	**int**			
	strAddress	**in**	**string**			
	return	**return**	**IUMLGuiTextHyperlink** 1250			

Operation **IUMLDataAll::InsertOwnedHyperlink2FileAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	strFilePathOrUrl	**in**	**string**			
	return	**return**	**IUMLHyperlink2File** 1076			

Operation **IUMLDataAll::InsertOwnedHyperlink2GuiElementAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	ipLinkedGuiElement	**in**	**IUMLGuiVisibleElement** 1260			
	ipLinkedGuiElementCell	**in**	**IUMLNamedElement** 1117			
	return	**return**	**IUMLHyperlink2GuiElement** 1077			

Operation **IUMLDataAll::InsertOwnedHyperlink2ModelAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	ipLinkedData	**in**	**IUMLData** 904			
	return	**return**	**IUMLHyperlink2Model** 1078			

Operation **IUMLDataAll::InsertOwnedLiteralAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	**IUMLEnumerationLiteral** 1057			

Operation **IUMLDataAll::InsertOwnedOperationAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	**IUMLOperation** 1131			

Operation **IUMLDataAll::InsertOwnedParameterAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	**IUMLParameter** 1140			

Operation **IUMLDataAll::InsertOwnedPortAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	**IUMLPort** 1143			

Operation **IUMLDataAll::InsertOwnedReceptionAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	**IUMLReception** 1154			

© 2019-2025 Altova GmbH

Operation **IUMLDataAll::InsertOwnedRuleAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	**IUMLConstraint** 1035			

Operation **IUMLDataAll::InsertOwnedTemplateBindingAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	ipSignature	**in**	**IUMLTemplateSignature** 1181			
	return	**return**	**IUMLTemplateBinding** 1177			

Operation **IUMLDataAll::InsertOwnedTemplateParameterAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	**IUMLClassifierTemplateParameter** 1021			

Operation **IUMLDataAll::InsertOwnedUseCaseAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	**IUMLUseCase** 1192			

Operation **IUMLDataAll::InsertPackagedElementAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	strKind	**in**	**string**			
	return	**return**	**IUMLPackageableElement** 1137			

Operation **IUMLDataAll::InsertPackagedElementRelationshipAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	strKind	**in**	**string**			
	ipFrom	**in**	**IUMLElement** 1050			
	ipTo	**in**	**IUMLElement** 1050			
	return	**return**	**IUMLPackageableElement** 1137			

Operation **IUMLDataAll::InsertPackageImportAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	ipImportedPackage	**in**	**IUMLPackage** 1134			
	return	**return**	**IUMLPackageImport** 1138			

Operation **IUMLDataAll::InsertPackageMergeAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	ipMergedPackage	**in**	**IUMLPackage** 1134			
	return	**return**	**IUMLPackageMerge** 1139			

Operation **IUMLDataAll::InsertParameterSubstitutionAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	ipFormalParameter	**in**	**IUMLTemplateParameter** 1178			
	ipActualParameter	**in**	**IUMLParameterableElement** 1141			
	return	**return**	**IUMLTemplateParameterSubstitution** 1180			

Operation **IUMLDataAll::InsertPostconditionAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	**IUMLConstraint** 1035			

Operation **IUMLDataAll::InsertPreconditionAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	**IUMLConstraint** 1035			

Operation **IUMLDataAll::InsertProfileApplicationAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	ipAppliedProfile	**in**	**IUMLProfile** 1145			
	return	**return**	**IUMLProfileApplication** 1146			

Operation **IUMLDataAll::InsertQualifierAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	**IUMLProperty** 1147			

Operation **IUMLDataAll::InsertRaisedExceptionAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	ipVal	**in**	**IUMLType** 1189			
	return	**return**	**void**			

Operation **IUMLDataAll::InsertRealizationAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			

© 2019-2025 Altova GmbH

	ipRealizingClassifier	**in**	**IUMLClassifier** 1018			
	return	**return**	**IUMLComponentRealization** 1029			

Operation **IUMLDataAll::InsertRealizingConnectorAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	ipVal	**in**	**IUMLConnector** 1033			
	return	**return**	**void**			

Operation **IUMLDataAll::InsertRegionAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	**IUMLRegion** 1157			

Operation **IUMLDataAll::InsertResultAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	**IUMLOutputPin** 1133			

Operation **IUMLDataAll::InsertSlotAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	ipDefiningFeature	**in**	**IUMLStructuralFeature** 1171			
	return	**return**	**IUMLSlot** 1162			

Operation **IUMLDataAll::InsertSlotInstanceValueAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	ipInstance	**in**	**IUMLInstanceSpecification** 1085			
	return	**return**	**IUMLInstanceValue** 1087			

Operation **IUMLDataAll::InsertSubjectAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	pSubject	**in**	**IUMLClassifier** 1018			
	return	**return**	**void**			

Operation **IUMLDataAll::InsertSubPartitionAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	strKind	**in**	**string**			
	return	**return**	**IUMLActivityPartition** 995			

Operation **IUMLDataAll::InsertSubVertexAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	strKind	**in**	**string**			
	return	**return**	**IUMLVertex** 1199			

Operation **IUMLDataAll::InsertTransitionAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	ipSource	**in**	**IUMLVertex** 1199			
	ipTarget	**in**	**IUMLVertex** 1199			
	return	**return**	**IUMLTransition** 1187			

Operation **IUMLDataAll::InsertTriggerAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	**IUMLTrigger** 1188			

Operation **IUMLDataAll::InsertValueAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	strKind	**in**	**string**			
	return	**return**	**IUMLValueSpecification** 1195			

Operation **IUMLDataAll::InsertWaypointAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	**IUMLGuiWaypoint** 1261			

Operation **IUMLDataAll::Instance**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLInstanceSpecification** 1085			

Operation **IUMLDataAll::InStates**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			

Operation **IUMLDataAll::IntegerValue**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**int**			

Operation **IUMLDataAll::InteractionOperator**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**ENUMUMLInteractionOperatorKind**			

© 2019-2025 Altova GmbH

	1269

Operation **IUMLDataAll::Interface**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLInterface** 1094			

Operation **IUMLDataAll::InterfaceRealizations**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			

Operation **IUMLDataAll::InterruptibleActivityRegions**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			

Operation **IUMLDataAll::InterruptingEdges**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			

Operation **IUMLDataAll::Invariant**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLConstraint** 1035			

Operation **IUMLDataAll::IsAbstract**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IUMLDataAll::IsActive**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IUMLDataAll::IsActivityReadOnly**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IUMLDataAll::IsBehavior**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IUMLDataAll::IsCodeLangNamespace**

parameter	name	direction	type	type modifier	multiplicity	default
	nCodeLang	**in**	**ENUMCodeLang** 897			
	return	**return**	**bool**			

Operation **IUMLDataAll::IsCodeLangNamespaceRoot**

parameter	name	direction	type	type modifier	multiplicity	default
	nCodeLang	**in**	**ENUMCodeLang** 897			
	return	**return**	**bool**			

Operation **IUMLDataAll::IsCodeProjectFile**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IUMLDataAll::IsCombineDuplicate**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IUMLDataAll::IsComposite**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IUMLDataAll::IsComputable**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IUMLDataAll::IsConjugated**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IUMLDataAll::IsControl**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IUMLDataAll::IsControlType**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IUMLDataAll::IsDerived**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IUMLDataAll::IsDerivedUnion**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IUMLDataAll::IsDimension**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IUMLDataAll::IsEditable**

© 2019-2025 Altova GmbH

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IUMLDataAll::IsElementVisible**

parameter	name	direction	type	type modifier	multiplicity	default
	ipElement	**in**	**IUMLElement** 1050			
	return	**return**	**bool**			

Operation **IUMLDataAll::IsExternal**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IUMLDataAll::IsFinalSpecialization**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IUMLDataAll::IsFirstEvent**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IUMLDataAll::IsHorizontal**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IUMLDataAll::IsIndirectlyInstantiated**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IUMLDataAll::IsKindOf**

parameter	name	direction	type	type modifier	multiplicity	default
	strKind	**in**	**string**			
	return	**return**	**bool**			

Operation **IUMLDataAll::IsLeaf**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IUMLDataAll::IsLocallyReentrant**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IUMLDataAll::IsLocked**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IUMLDataAll::IsMultiCast**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IUMLDataAll::IsMultiReceive**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IUMLDataAll::IsNavigable**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IUMLDataAll::IsNull**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IUMLDataAll::IsOrdered**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IUMLDataAll::IsOrthogonal**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IUMLDataAll::IsOwnedEnd**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IUMLDataAll::IsPositioned**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IUMLDataAll::IsPredefinedStereotypeApplied**

parameter	name	direction	type	type modifier	multiplicity	default
	nStereotype	**in**	**ENUMUMLPredefinedElement** 1271			
	return	**return**	**bool**			

Operation **IUMLDataAll::IsQuery**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IUMLDataAll::IsReadOnly**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IUMLDataAll::IsReentrant**

© 2019-2025 Altova GmbH

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IUMLDataAll::IsRelative**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IUMLDataAll::IsSameUMLData**

parameter	name	direction	type	type modifier	multiplicity	default
	ipUMLDataToCompare	**in**	**IUMLData** 904			
	return	**return**	**bool**			

Operation **IUMLDataAll::IsService**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IUMLDataAll::IsShared**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IUMLDataAll::IsShowAsGeneralValueLifeline**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IUMLDataAll::IsSimple**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IUMLDataAll::IsStatic**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IUMLDataAll::IsStereotypeApplied**

parameter	name	direction	type	type modifier	multiplicity	default
	ipStereotype	**in**	**IUMLStereotype** 1169			
	return	**return**	**bool**			

Operation **IUMLDataAll::IsSubmachineState**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IUMLDataAll::IsSubstitutable**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IUMLDataAll::IsSynchronous**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IUMLDataAll::IsTextLabelVisible**

parameter	name	direction	type	type modifier	multiplicity	default
	ipTextLabel	**in**	**IUMLGuiTextLabel** 1251			
	return	**return**	**bool**			

Operation **IUMLDataAll::IsUnique**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IUMLDataAll::IsUnmarshall**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IUMLDataAll::IsUseForCodeEngineering**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IUMLDataAll::IsVarArgList**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IUMLDataAll::IsVisible**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IUMLDataAll::JoinSpec**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**string**			

Operation **IUMLDataAll::KindName**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**string**			

Operation **IUMLDataAll::Label**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**string**			

Operation **IUMLDataAll::Language**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**string**			

Operation **IUMLDataAll::Layer**

© 2019-2025 Altova GmbH

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLGuiDiagramLayer** 1215			

Operation **IUMLDataAll::Layers**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			

Operation **IUMLDataAll::Left**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**int**			

Operation **IUMLDataAll::Lifelines**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			

Operation **IUMLDataAll::LineBegin**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLGuiElement** 1216			

Operation **IUMLDataAll::LineConnectionWaypoints**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			

Operation **IUMLDataAll::LineEnd**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLGuiElement** 1216			

Operation **IUMLDataAll::LineLinks**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			

Operation **IUMLDataAll::LinkAddress**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**string**			

Operation **IUMLDataAll::LinkedGuiElement**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLGuiVisibleElement** 1260			

Operation **IUMLDataAll::LinkedGuiElementCell**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLNamedElement** 1117			

Operation **IUMLDataAll::LinkedModelElement**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLData 904			

Operation **IUMLDataAll::LinkedOwner**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLElement 1050			

Operation **IUMLDataAll::LocalPostConditions**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLDataList 906			

Operation **IUMLDataAll::LocalPreConditions**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLDataList 906			

Operation **IUMLDataAll::Location**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLDeploymentTarget 1044			

Operation **IUMLDataAll::LowerValues**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLDataList 906			

Operation **IUMLDataAll::Manifestations**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLDataList 906			

Operation **IUMLDataAll::Mapping**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLOpaqueExpression 1130			

Operation **IUMLDataAll::Max**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLValueSpecification 1195			

Operation **IUMLDataAll::MaxInt**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLValueSpecification 1195			

Operation **IUMLDataAll::MemberEnds**

parameter	name	direction	type	type modifier	multiplicity	default

© 2019-2025 Altova GmbH

	return	**return**	IUMLDataList 906

Operation **IUMLDataAll::Members**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLDataList 906			

Operation **IUMLDataAll::MergedPackage**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLPackage 1134			

Operation **IUMLDataAll::MergeLayersAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nFromIdx	**in**	**int**			
	nToIdx	**in**	**int**			
	return	**return**	**void**			

Operation **IUMLDataAll::Message**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLMessage 1111			

Operation **IUMLDataAll::MessageKind**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	ENUMUMLMessageKind 1270			

Operation **IUMLDataAll::Messages**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLDataList 906			

Operation **IUMLDataAll::MessageSort**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	ENUMUMLMessageSort 1270			

Operation **IUMLDataAll::MetaClass**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**string**			

Operation **IUMLDataAll::Methods**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLDataList 906			

Operation **IUMLDataAll::MiddleWaypoint**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLGuiMiddleWaypoint 1225			

Operation **IUMLDataAll::Min**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLValueSpecification** 1195			

Operation **IUMLDataAll::MinInt**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLValueSpecification** 1195			

Operation **IUMLDataAll::Mode**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**ENUMUMLExpansionKind** 1266			

Operation **IUMLDataAll::MoveTo**

parameter	name	direction	type	type modifier	multiplicity	default
	nLeft	**in**	**int**			
	nTop	**in**	**int**			
	return	**return**	**void**			

Operation **IUMLDataAll::MustIsolate**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IUMLDataAll::Name**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**string**			

Operation **IUMLDataAll::NameCompartmentEndPos**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**int**			

Operation **IUMLDataAll::Namespace**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLNamespace** 1120			

Operation **IUMLDataAll::NavigableOwnedEnds**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			

Operation **IUMLDataAll::NestedArtifacts**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			

Operation **IUMLDataAll::NestedClassifiers**

© 2019-2025 Altova GmbH

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			

Operation **IUMLDataAll::NestedNodes**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			

Operation **IUMLDataAll::NestedPackages**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			

Operation **IUMLDataAll::NestingInterface**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLInterface** 1094			

Operation **IUMLDataAll::NestingPackage**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLPackage** 1134			

Operation **IUMLDataAll::Nodes**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			

Operation **IUMLDataAll::NoteText**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**string**			

Operation **IUMLDataAll::NoteTextEndPos**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**int**			

Operation **IUMLDataAll::NoteTextStartPos**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**int**			

Operation **IUMLDataAll::Observations**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			

Operation **IUMLDataAll::OccurringEvent**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLEvent** 1058			

Operation **IUMLDataAll::OpenLink**

parameter	name	direction	type	type modifier	multiplicity	default

	return	**return**	**void**

Operation **IUMLDataAll::OperandGuard**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLInteractionConstraint** 1090			

Operation **IUMLDataAll::Operands**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			

Operation **IUMLDataAll::Operation**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLOperation** 1131			

Operation **IUMLDataAll::Opposite**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLProperty** 1147			

Operation **IUMLDataAll::Ordering**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**ENUMUMLObjectNodeOrderingKind** 1271			

Operation **IUMLDataAll::OutgoingEdges**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			

Operation **IUMLDataAll::Outgoings**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			

Operation **IUMLDataAll::OutputElements**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			

Operation **IUMLDataAll::OutputPins**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			

Operation **IUMLDataAll::OutputValues**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			

© 2019-2025 Altova GmbH

Operation **IUMLDataAll::OwnedActual**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLParameterableElement** 1141			

Operation **IUMLDataAll::OwnedArguments**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			

Operation **IUMLDataAll::OwnedAttributes**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			

Operation **IUMLDataAll::OwnedBehaviors**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			

Operation **IUMLDataAll::OwnedComments**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			

Operation **IUMLDataAll::OwnedConnectors**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			

Operation **IUMLDataAll::OwnedDiagrams**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			

Operation **IUMLDataAll::OwnedDocComment**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLComment** 1025			

Operation **IUMLDataAll::OwnedDocCommentBody**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**string**			

Operation **IUMLDataAll::OwnedElements**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			

Operation **IUMLDataAll::OwnedEnds**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			

Operation **IUMLDataAll::OwnedGuiElements**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLDataList 906			

Operation **IUMLDataAll::OwnedGuiNodeLinks**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLDataList 906			

Operation **IUMLDataAll::OwnedHyperlinks**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLDataList 906			

Operation **IUMLDataAll::OwnedLiterals**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLDataList 906			

Operation **IUMLDataAll::OwnedMembers**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLDataList 906			

Operation **IUMLDataAll::OwnedOperations**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLDataList 906			

Operation **IUMLDataAll::OwnedParameteredElement**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLParameterableElement 1141			

Operation **IUMLDataAll::OwnedParameters**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLDataList 906			

Operation **IUMLDataAll::OwnedPorts**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLDataList 906			

Operation **IUMLDataAll::OwnedReceptions**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLDataList 906			

Operation **IUMLDataAll::OwnedRules**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLDataList 906			

Operation **IUMLDataAll::OwnedStereotypes**

© 2019-2025 Altova GmbH

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLDataList 906			

Operation **IUMLDataAll::OwnedTemplateBindings**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLDataList 906			

Operation **IUMLDataAll::OwnedTemplateParameters**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLDataList 906			

Operation **IUMLDataAll::OwnedTemplateSignature**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLTemplateSignature 1181			

Operation **IUMLDataAll::OwnedTypes**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLDataList 906			

Operation **IUMLDataAll::OwnedUseCases**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLDataList 906			

Operation **IUMLDataAll::Owner**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLElement 1050			

Operation **IUMLDataAll::OwningAssociation**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLAssociation 1000			

Operation **IUMLDataAll::OwningConstraint**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLConstraint 1035			

Operation **IUMLDataAll::OwningElement**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLElement 1050			

Operation **IUMLDataAll::OwningGuiNodeLink**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLGuiNodeLink 1226			

Operation **IUMLDataAll::OwningInstance**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLInstanceSpecification** 1085			

Operation **IUMLDataAll::OwningInstanceSpec**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLInstanceSpecification** 1085			

Operation **IUMLDataAll::OwningLower**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLMultiplicityElement** 1116			

Operation **IUMLDataAll::OwningPackage**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLPackage** 1134			

Operation **IUMLDataAll::OwningParameter**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLParameter** 1140			

Operation **IUMLDataAll::OwningProperty**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLProperty** 1147			

Operation **IUMLDataAll::OwningSignal**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLSignal** 1160			

Operation **IUMLDataAll::OwningSlot**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLSlot** 1162			

Operation **IUMLDataAll::OwningState**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLState** 1163			

Operation **IUMLDataAll::OwningTemplateParameter**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLTemplateParameter** 1178			

Operation **IUMLDataAll::OwningTransition**

© 2019-2025 Altova GmbH

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLProtocolTransition** 1151			

Operation **IUMLDataAll::OwningUpper**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLMultiplicityElement** 1116			

Operation **IUMLDataAll::Package**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLPackage** 1134			

Operation **IUMLDataAll::PackagedElements**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			

Operation **IUMLDataAll::PackageImports**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			

Operation **IUMLDataAll::PackageMerges**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			

Operation **IUMLDataAll::Parameter**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLParameter** 1140			

Operation **IUMLDataAll::ParameteredElement**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLParameterableElement** 1141			

Operation **IUMLDataAll::ParameterSignature**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLTemplateSignature** 1181			

Operation **IUMLDataAll::ParameterSubstitutions**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			

Operation **IUMLDataAll::Parent**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IDispatch**			

Operation **IUMLDataAll::PinValue**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**string**			

Operation **IUMLDataAll::PostCondition**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLConstraint** 1035			

Operation **IUMLDataAll::Postconditions**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			

Operation **IUMLDataAll::PostTypeModifier**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**string**			

Operation **IUMLDataAll::PosX**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**int**			

Operation **IUMLDataAll::PosY**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**int**			

Operation **IUMLDataAll::PreCondition**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLConstraint** 1035			

Operation **IUMLDataAll::Preconditions**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			

Operation **IUMLDataAll::ProfileApplications**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			

Operation **IUMLDataAll::ProtectedNode**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLExecutableNode** 1061			

Operation **IUMLDataAll::Protocol**

parameter	name	direction	type	type modifier	multiplicity	default

© 2019-2025 Altova GmbH

	return	**return**	**IUMLProtocolStateMachine** 1150

Operation **IUMLDataAll::PseudostateKind**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**ENUMUMLPseudostateKind** 1272			

Operation **IUMLDataAll::QualifiedName**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**string**			

Operation **IUMLDataAll::Qualifiers**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			

Operation **IUMLDataAll::RaisedExceptions**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			

Operation **IUMLDataAll::Realizations**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			

Operation **IUMLDataAll::RealizingClassifier**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLClassifier** 1018			

Operation **IUMLDataAll::RealizingConnectors**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			

Operation **IUMLDataAll::ReceiveEvent**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLMessageEnd** 1113			

Operation **IUMLDataAll::ReceivingPackage**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLPackage** 1134			

Operation **IUMLDataAll::ReferencedDiagram**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLGuiDiagram** 1211			

Operation **IUMLDataAll::Referred**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			

Operation **IUMLDataAll::RefersTo**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLInteraction** 1088			

Operation **IUMLDataAll::RegionAsInput**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLExpansionRegion** 1065			

Operation **IUMLDataAll::RegionAsOutput**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLExpansionRegion** 1065			

Operation **IUMLDataAll::Regions**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			

Operation **IUMLDataAll::RelatedElements**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			

Operation **IUMLDataAll::RelativeNodes**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			

Operation **IUMLDataAll::Represents**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLConnectableElement** 1030			

Operation **IUMLDataAll::Result**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLOutputPin** 1133			

Operation **IUMLDataAll::Results**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			

Operation **IUMLDataAll::Right**

parameter	name	direction	type	type modifier	multiplicity	default

© 2019-2025 Altova GmbH

	return	**return**	**int**

Operation **IUMLDataAll::Role**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLConnectableElement** 1030			

Operation **IUMLDataAll::RoleBindings**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			

Operation **IUMLDataAll::ScrollPosX**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**int**			

Operation **IUMLDataAll::ScrollPosY**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**int**			

Operation **IUMLDataAll::Selection**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLBehavior** 1002			

Operation **IUMLDataAll::Selector**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLValueSpecification** 1195			

Operation **IUMLDataAll::SendEvent**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLMessageEnd** 1113			

Operation **IUMLDataAll::SendSignal**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLSignal** 1160			

Operation **IUMLDataAll::SeparatorCount**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**int**			

Operation **IUMLDataAll::SetCodeFileName**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	strNewVal	**in**	**string**			
	return	**return**	**void**			

Operation **IUMLDataAll::SetElementVisible**

parameter	name	direction	type	type modifier	multiplicity	default
	ipElement	**in**	**IUMLElement** 1050			
	bVisible	**in**	**bool**			
	return	**return**	**void**			

Operation **IUMLDataAll::SetHSeparatorPosition**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	nPosition	**in**	**int**			
	return	**return**	**void**			

Operation **IUMLDataAll::SetHyperlinkFileAddress**

parameter	name	direction	type	type modifier	multiplicity	default
	strNewVal	**in**	**string**			
	return	**return**	**void**			

Operation **IUMLDataAll::SetHyperlinkGuiElementAddress**

parameter	name	direction	type	type modifier	multiplicity	default
	ipLinkedGuiElemein nt		**IUMLGuiVisibleEl ement** 1260			
	ipLinkedGuiElemein ntCell		**IUMLNamedElem ent** 1117			
	return	**return**	**void**			

Operation **IUMLDataAll::SetHyperlinkModelElementAddress**

parameter	name	direction	type	type modifier	multiplicity	default
	ipLinkedData	**in**	**IUMLData** 904			
	return	**return**	**void**			

Operation **IUMLDataAll::SetMultiplicity**

parameter	name	direction	type	type modifier	multiplicity	default
	strNewVal	**in**	**string**			
	return	**return**	**void**			

Operation **IUMLDataAll::SetName**

parameter	name	direction	type	type modifier	multiplicity	default
	strStartWith	**in**	**string**			
	return	**return**	**string**			

Operation **IUMLDataAll::SetNewActionValue**

parameter	name	direction	type	type modifier	multiplicity	default
	strKind	**in**	**string**			
	return	**return**	**IUMLValueSpecifi cation** 1195			

Operation **IUMLDataAll::SetNewCallTarget**

parameter	name	direction	type	type modifier	multiplicity	default
	strKind	**in**	**string**			

© 2019-2025 Altova GmbH

	return	**return**	IUMLInputPin 1083

Operation **IUMLDataAll::SetNewChangeExpression**

parameter	name	direction	type	type modifier	multiplicity	default
	strKind	**in**	**string**			
	return	**return**	IUMLValueSpecification 1195			

Operation **IUMLDataAll::SetNewDefaultValue**

parameter	name	direction	type	type modifier	multiplicity	default
	strKind	**in**	**string**			
	return	**return**	IUMLValueSpecification 1195			

Operation **IUMLDataAll::SetNewDefaultValueInstanceValue**

parameter	name	direction	type	type modifier	multiplicity	default
	ipInstance	**in**	IUMLInstanceSpecification 1085			
	return	**return**	IUMLInstanceValue 1087			

Operation **IUMLDataAll::SetNewDefaultValueLiteralString**

parameter	name	direction	type	type modifier	multiplicity	default
	strNewVal	**in**	**string**			
	return	**return**	IUMLLiteralString 1108			

Operation **IUMLDataAll::SetNewDoActivity**

parameter	name	direction	type	type modifier	multiplicity	default
	strKind	**in**	**string**			
	return	**return**	IUMLBehavior 1002			

Operation **IUMLDataAll::SetNewEffect**

parameter	name	direction	type	type modifier	multiplicity	default
	strKind	**in**	**string**			
	return	**return**	IUMLBehavior 1002			

Operation **IUMLDataAll::SetNewEntry**

parameter	name	direction	type	type modifier	multiplicity	default
	strKind	**in**	**string**			
	return	**return**	IUMLBehavior 1002			

Operation **IUMLDataAll::SetNewExit**

parameter	name	direction	type	type modifier	multiplicity	default
	strKind	**in**	**string**			

	return	**return**	**IUMLBehavior** 1002			

Operation **IUMLDataAll::SetNewExpr**

parameter	name	direction	type	type modifier	multiplicity	default
	strKind	**in**	**string**			
	return	**return**	**IUMLValueSpecification** 1195			

Operation **IUMLDataAll::SetNewInvariant**

parameter	name	direction	type	type modifier	multiplicity	default
	strKind	**in**	**string**			
	return	**return**	**IUMLConstraint** 1035			

Operation **IUMLDataAll::SetNewMapping**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLOpaqueExpression** 1130			

Operation **IUMLDataAll::SetNewMax**

parameter	name	direction	type	type modifier	multiplicity	default
	strKind	**in**	**string**			
	return	**return**	**IUMLValueSpecification** 1195			

Operation **IUMLDataAll::SetNewMaxInt**

parameter	name	direction	type	type modifier	multiplicity	default
	strKind	**in**	**string**			
	return	**return**	**IUMLValueSpecification** 1195			

Operation **IUMLDataAll::SetNewMin**

parameter	name	direction	type	type modifier	multiplicity	default
	strKind	**in**	**string**			
	return	**return**	**IUMLValueSpecification** 1195			

Operation **IUMLDataAll::SetNewMinInt**

parameter	name	direction	type	type modifier	multiplicity	default
	strKind	**in**	**string**			
	return	**return**	**IUMLValueSpecification** 1195			

Operation **IUMLDataAll::SetNewOperandGuard**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLInteractionConstraint** 1090			

Operation **IUMLDataAll::SetNewOwnedParameteredElement**

© 2019-2025 Altova GmbH

parameter	name	direction	type	type modifier	multiplicity	default
	strKind	**in**	**string**			
	return	**return**	**IUMLParameterableElement** 1141			

Operation **IUMLDataAll::SetNewPostCondition**

parameter	name	direction	type	type modifier	multiplicity	default
	strKind	**in**	**string**			
	return	**return**	**IUMLConstraint** 1035			

Operation **IUMLDataAll::SetNewPreCondition**

parameter	name	direction	type	type modifier	multiplicity	default
	strKind	**in**	**string**			
	return	**return**	**IUMLConstraint** 1035			

Operation **IUMLDataAll::SetNewProtocol**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLProtocolStateMachine** 1150			

Operation **IUMLDataAll::SetNewSelector**

parameter	name	direction	type	type modifier	multiplicity	default
	strKind	**in**	**string**			
	return	**return**	**IUMLValueSpecification** 1195			

Operation **IUMLDataAll::SetNewSignalTarget**

parameter	name	direction	type	type modifier	multiplicity	default
	strKind	**in**	**string**			
	return	**return**	**IUMLInputPin** 1083			

Operation **IUMLDataAll::SetNewSpecification**

parameter	name	direction	type	type modifier	multiplicity	default
	strKind	**in**	**string**			
	return	**return**	**IUMLValueSpecification** 1195			

Operation **IUMLDataAll::SetNewSpecificationInstanceValue**

parameter	name	direction	type	type modifier	multiplicity	default
	ipInstance	**in**	**IUMLInstanceSpecification** 1085			
	return	**return**	**IUMLInstanceValue** 1087			

Operation **IUMLDataAll::SetNewSpecificationLiteralString**

parameter	name	direction	type	type modifier	multiplicity	default
	strNewVal	**in**	**string**			

	return	**return**	**IUMLLiteralString** 1108

Operation **IUMLDataAll::SetNewStateInvariant**

parameter	name	direction	type	type modifier	multiplicity	default
	strKind	**in**	**string**			
	return	**return**	**IUMLConstraint** 1035			

Operation **IUMLDataAll::SetNewTemplateSignature**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLTemplateSignature** 1181			

Operation **IUMLDataAll::SetNewTransitionGuard**

parameter	name	direction	type	type modifier	multiplicity	default
	strKind	**in**	**string**			
	return	**return**	**IUMLConstraint** 1035			

Operation **IUMLDataAll::SetNewWhen**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLTimeExpression** 1184			

Operation **IUMLDataAll::SetOperation**

parameter	name	direction	type	type modifier	multiplicity	default
	ipVal	**in**	**IUMLOperation** 1131			
	return	**return**	**void**			

Operation **IUMLDataAll::SetPos**

parameter	name	direction	type	type modifier	multiplicity	default
	x	**in**	**int**			
	y	**in**	**int**			
	return	**return**	**void**			

Operation **IUMLDataAll::SetPredefinedTaggedValueAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	nProperty	**in**	**ENUMUMLPredefinedElement** 1271			
	strNewValue	**in**	**string**			
	return	**return**	**IUMLValueSpecification** 1195			

Operation **IUMLDataAll::SetRect**

parameter	name	direction	type	type modifier	multiplicity	default
	nLeft	**in**	**int**			
	nTop	**in**	**int**			

© 2019-2025 Altova GmbH

	nRight	**in**	**int**
	nBottom	**in**	**int**
	return	**return**	**void**

Operation **IUMLDataAll::SetScrollPos**

parameter	name	direction	type	type modifier	multiplicity	default
	nX	**in**	**int**			
	nY	**in**	**int**			
	return	**return**	**void**			

Operation **IUMLDataAll::SetSeparatorPosition**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	nPosition	**in**	**int**			
	return	**return**	**void**			

Operation **IUMLDataAll::SetSlotInstanceValueAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	ipForDefiningFeature	**in**	**IUMLStructuralFeature** 1171			
	ipInstance	**in**	**IUMLInstanceSpecification** 1085			
	return	**return**	**IUMLInstanceValue** 1087			

Operation **IUMLDataAll::SetSlotValueAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	ipForDefiningFeature	**in**	**IUMLStructuralFeature** 1171			
	strNewValue	**in**	**string**			
	return	**return**	**IUMLValueSpecification** 1195			

Operation **IUMLDataAll::SetStateIndex**

parameter	name	direction	type	type modifier	multiplicity	default
	nTimeTickIndex	**in**	**int**			
	nNewVal	**in**	**int**			
	return	**return**	**void**			

Operation **IUMLDataAll::SetStateIndexErased**

parameter	name	direction	type	type modifier	multiplicity	default
	nTimeTickIndex	**in**	**int**			
	return	**return**	**void**			

Operation **IUMLDataAll::SetTaggedValueAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	ipDefiningFeature	**in**	**IUMLStructuralFeature** 1171			
	strNewValue	**in**	**string**			

	return	**return**	**IUMLValueSpecification** 1195

Operation **IUMLDataAll::SetTextLabelVisible**

parameter	name	direction	type	type modifier	multiplicity	default
	ipTextLabel	**in**	**IUMLGuiTextLabel** 1251			
	bVisible	**in**	**bool**			
	return	**return**	**void**			

Operation **IUMLDataAll::SetTimeTickLength**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	nNewVal	**in**	**int**			
	return	**return**	**void**			

Operation **IUMLDataAll::SetVisualStatePosition**

parameter	name	direction	type	type modifier	multiplicity	default
	nStateIndex	**in**	**int**			
	nNewVal	**in**	**int**			
	return	**return**	**void**			

Operation **IUMLDataAll::SetVSeparatorPosition**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	nPosition	**in**	**int**			
	return	**return**	**void**			

Operation **IUMLDataAll::Signal**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLSignal** 1160			

Operation **IUMLDataAll::SignalTarget**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLInputPin** 1083			

Operation **IUMLDataAll::Signature**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLTemplateSignature** 1181			

Operation **IUMLDataAll::SingleExecution**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IUMLDataAll::Slots**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			

© 2019-2025 Altova GmbH

Operation **IUMLDataAll::Source**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLActivityNode 992			

Operation **IUMLDataAll::Sources**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLDataList 906			

Operation **IUMLDataAll::Specific**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLClassifier 1018			

Operation **IUMLDataAll::Specification**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLValueSpecification 1195			

Operation **IUMLDataAll::Specifics**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLDataList 906			

Operation **IUMLDataAll::Start**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLOccurrenceSpecification 1127			

Operation **IUMLDataAll::State**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLState 1163			

Operation **IUMLDataAll::StateCompartmentEndPos**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**int**			

Operation **IUMLDataAll::StateInvariant**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLConstraint 1035			

Operation **IUMLDataAll::StateMachine**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLStateMachine 1167			

Operation **IUMLDataAll::Stereotype**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLStereotype 1169			

Operation **IUMLDataAll::StereotypeApplications**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLDataList 906			

Operation **IUMLDataAll::StereotypedElementStyles**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLGuiStyles 1241			

Operation **IUMLDataAll::StringValue**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**string**			

Operation **IUMLDataAll::StructuredActivityNodes**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLDataList 906			

Operation **IUMLDataAll::Styles**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLGuiStyles 1241			

Operation **IUMLDataAll::SubGroups**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLDataList 906			

Operation **IUMLDataAll::Subjects**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLDataList 906			

Operation **IUMLDataAll::Submachine**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLStateMachine 1167			

Operation **IUMLDataAll::SubmachineStates**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLDataList 906			

Operation **IUMLDataAll::SubPartitions**

parameter	name	direction	type	type modifier	multiplicity	default

© 2019-2025 Altova GmbH

	return	**return**	IUMLDataList 906

Operation **IUMLDataAll::SubprojectFilePath**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**string**			

Operation **IUMLDataAll::SubVertices**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLDataList 906			

Operation **IUMLDataAll::SuperClasses**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLDataList 906			

Operation **IUMLDataAll::SuperGroup**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLActivityGroup 991			

Operation **IUMLDataAll::SuperPartition**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLActivityPartition 995			

Operation **IUMLDataAll::SupplierDependencies**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLDataList 906			

Operation **IUMLDataAll::Suppliers**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLDataList 906			

Operation **IUMLDataAll::Symbol**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**string**			

Operation **IUMLDataAll::Target**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLActivityNode 992			

Operation **IUMLDataAll::Targets**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLDataList 906			

Operation **IUMLDataAll::Template**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLTemplateableElement** 1176			

Operation **IUMLDataAll::TemplateBinding**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLTemplateBinding** 1177			

Operation **IUMLDataAll::TemplateParameter**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLTemplateParameter** 1178			

Operation **IUMLDataAll::TextEndPos**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**int**			

Operation **IUMLDataAll::TextLabelElement**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLElement** 1050			

Operation **IUMLDataAll::TextLabelKind**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**ENUMUMLGuiTextLabelKind** 1268			

Operation **IUMLDataAll::TextLabels**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			

Operation **IUMLDataAll::TextStartPos**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**int**			

Operation **IUMLDataAll::TimeObservationEvent**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLNamedElement** 1117			

Operation **IUMLDataAll::TimeTickLengthCount**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**int**			

Operation **IUMLDataAll::Top**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**int**			

© 2019-2025 Altova GmbH

Operation **IUMLDataAll::Transformation**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLBehavior** 1002			

Operation **IUMLDataAll::TransitionGuard**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLConstraint** 1035			

Operation **IUMLDataAll::TransitionKind**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**ENUMUMLTransitionKind** 1273			

Operation **IUMLDataAll::Transitions**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			

Operation **IUMLDataAll::TransitionSource**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLVertex** 1199			

Operation **IUMLDataAll::TransitionTarget**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLVertex** 1199			

Operation **IUMLDataAll::Triggers**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			

Operation **IUMLDataAll::Type**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLType** 1189			

Operation **IUMLDataAll::TypedElements**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			

Operation **IUMLDataAll::UnapplyPredefinedStereotype**

parameter	name	direction	type	type modifier	multiplicity	default
	nStereotype	**in**	**ENUMUMLPredefinedElement** 1271			
	return	**return**	**void**			

Operation **IUMLDataAll::UnapplyStereotype**

parameter	name	direction	type	type modifier	multiplicity	default
	ipStereotype	**in**	**IUMLStereotype** 1169			
	return	**return**	**void**			

Operation **IUMLDataAll::UnlimitedValue**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**int**			

Operation **IUMLDataAll::UpperBound**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**string**			

Operation **IUMLDataAll::UpperValues**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			

Operation **IUMLDataAll::URI**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**string**			

Operation **IUMLDataAll::UseCase**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLUseCase** 1192			

Operation **IUMLDataAll::UseCases**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			

Operation **IUMLDataAll::UseForForwardEngineering**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IUMLDataAll::UserDefinedLinkName**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**string**			

Operation **IUMLDataAll::UtilizedElement**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLPackageable Element** 1137			

Operation **IUMLDataAll::UUID**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**string**			

Operation **IUMLDataAll::Value**

© 2019-2025 Altova GmbH

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**string**			

Operation **IUMLDataAll::Values**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			

Operation **IUMLDataAll::Viewpoint**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**string**			

Operation **IUMLDataAll::Visibility**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**ENUMUMLVisibilityKind** 1273			

Operation **IUMLDataAll::VisualStatePositionCount**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**int**			

Operation **IUMLDataAll::VSeparatorCount**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**int**			

Operation **IUMLDataAll::WasUsedForCodeSynchronization**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IUMLDataAll::Waypoints**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			

Operation **IUMLDataAll::Weight**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**string**			

Operation **IUMLDataAll::When**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLTimeExpression** 1184			

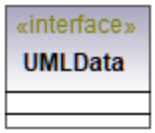
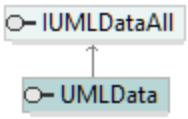
Operation **IUMLDataAll::ZoomFactor**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**int**			

UML documentation generated by **UModel** UML Editor **http://www.altova.com/umodel** Wed Sep 25 10:33:04 2024

16.4.3.4 UModelAPI - UMLData

Interface **UMLData**

diagram	
hierarchy	

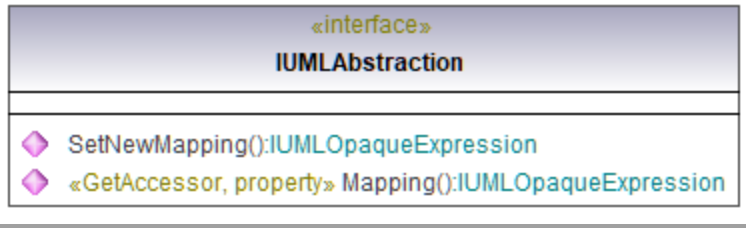
UML documentation generated by **UModel** UML Editor **http://www.altova.com/umodel** Wed Sep 25 10:33:04 2024

16.4.3.5 IUMLElement

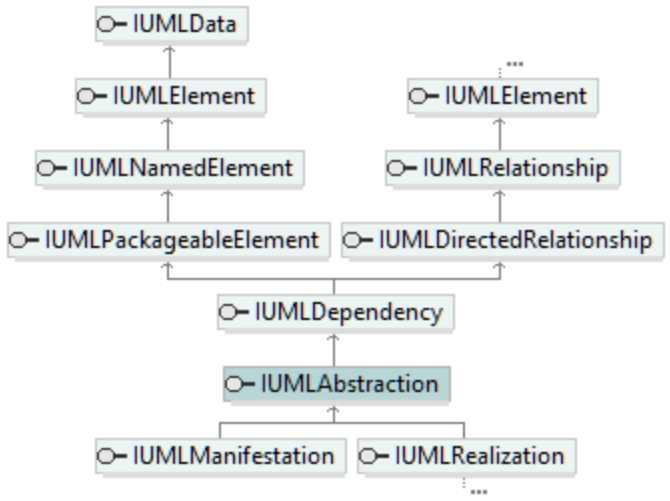
This is a list of elements as defined by OMG in the UML Specification, see http://www.uml.org.

16.4.3.5.1 UModelAPI - IUMLAbstraction

Interface **IUMLAbstraction**

diagram	

 © 2019-2025 Altova GmbH

hierarchy	
	

Operation **IUMLAbstraction::Mapping**

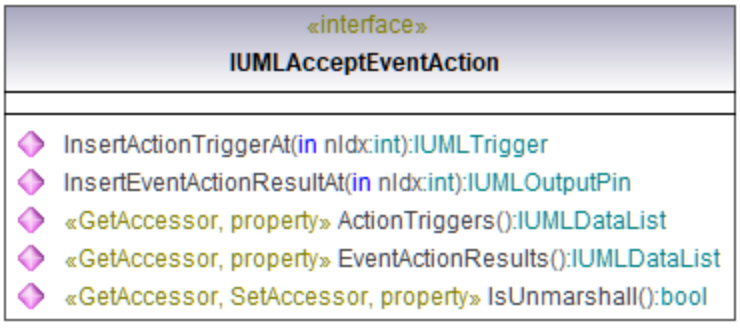
parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLOpaqueExpression 1130			

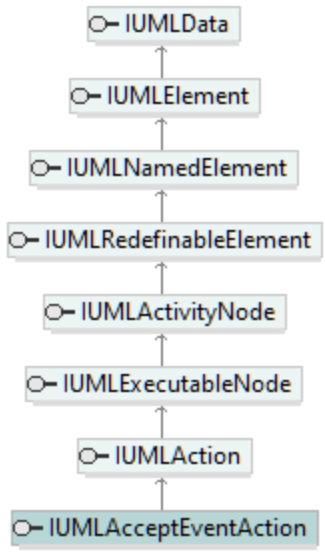
Operation **IUMLAbstraction::SetNewMapping**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLOpaqueExpression 1130			

16.4.3.5.2 UModelAPI - IUMLAcceptEventAction

Interface **IUMLAcceptEventAction**

diagram	
	

hierarchy	

Operation **IUMLAcceptEventAction::ActionTriggers**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			
documentation	A list of elements of type IUMLTrigger 1188.					

Operation **IUMLAcceptEventAction::EventActionResults**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			
documentation	A list of elements of type IUMLOutputPin 1133.					

Operation **IUMLAcceptEventAction::InsertActionTriggerAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	**IUMLTrigger** 1188			

Operation **IUMLAcceptEventAction::InsertEventActionResultAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	**IUMLOutputPin** 1133			

Operation **IUMLAcceptEventAction::IsUnmarshall**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

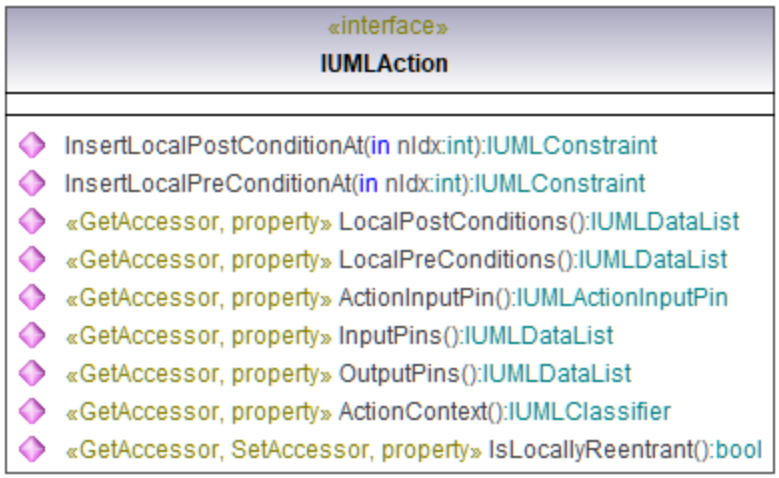
© 2019-2025 Altova GmbH

UML documentation generated by UModel UML Editor http://www.altova.com/umodel Wed Sep 25 10:33:04 2024

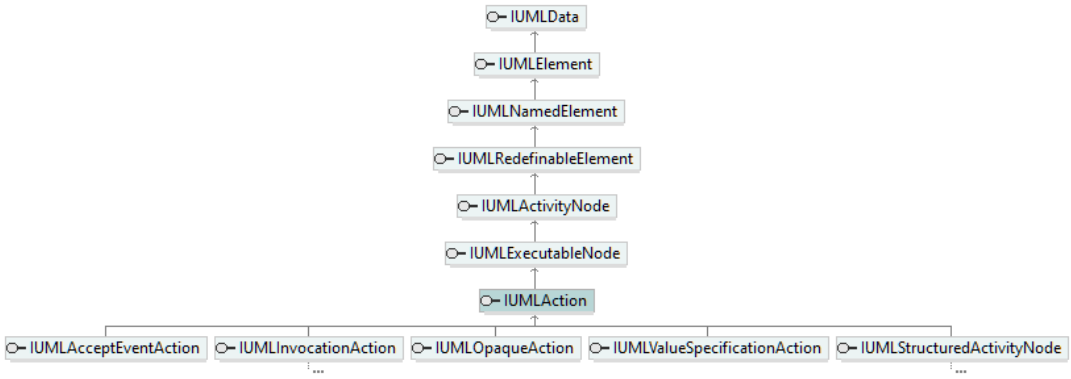
16.4.3.5.3 UModelAPI - IUMLAction

Interface **IUMLAction**

diagram



hierarchy



Operation **IUMLAction::ActionContext**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLClassifier** 1018			

Operation **IUMLAction::ActionInputPin**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLActionInputPin** 985			

Operation **IUMLAction::InputPins**

parameter	name	direction	type	type modifier	multiplicity	default

	return	**return**	**IUMLDataList** 906
documentation	A list of elements of type IUMLInputPin 1083.		

Operation **IUMLAction::InsertLocalPostConditionAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	**IUMLConstraint** 1035			

Operation **IUMLAction::InsertLocalPreConditionAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	**IUMLConstraint** 1035			

Operation **IUMLAction::IsLocallyReentrant**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IUMLAction::LocalPostConditions**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			
documentation	A list of elements of type IUMLConstraint 1035.					

Operation **IUMLAction::LocalPreConditions**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			
documentation	A list of elements of type IUMLConstraint 1035.					

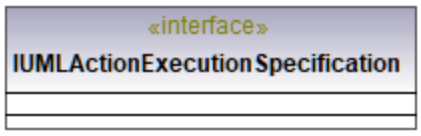
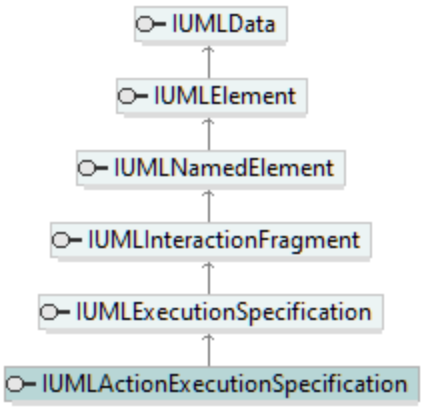
Operation **IUMLAction::OutputPins**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			
documentation	A list of elements of type IUMLOutputPin 1133.					

© 2019-2025 Altova GmbH

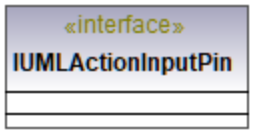
16.4.3.5.4 UModelAPI - IUMLActionExecutionSpecification

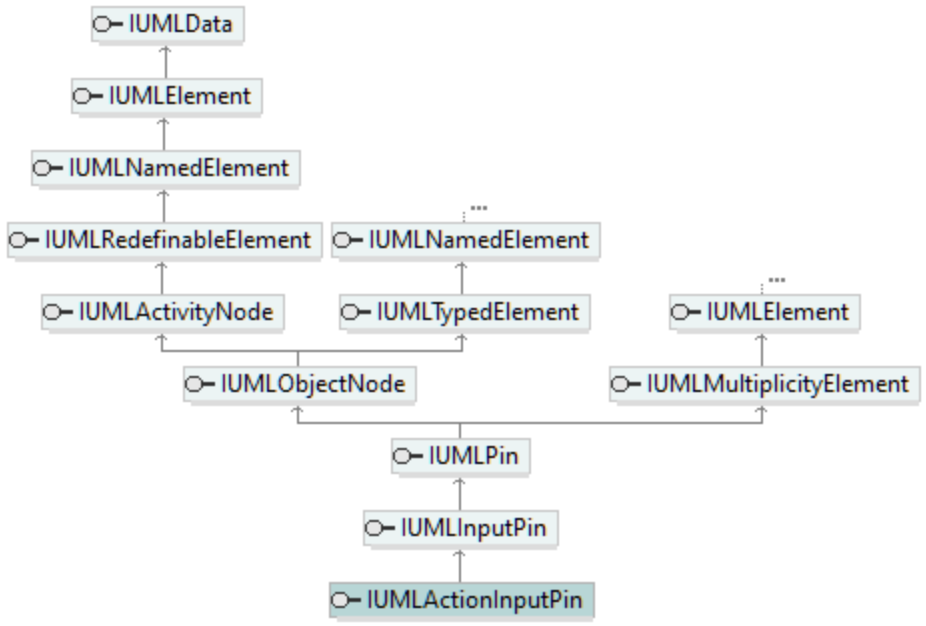
Interface **IUMLActionExecutionSpecification**

diagram	
hierarchy	

16.4.3.5.5 UModelAPI - IUMLActionInputPin

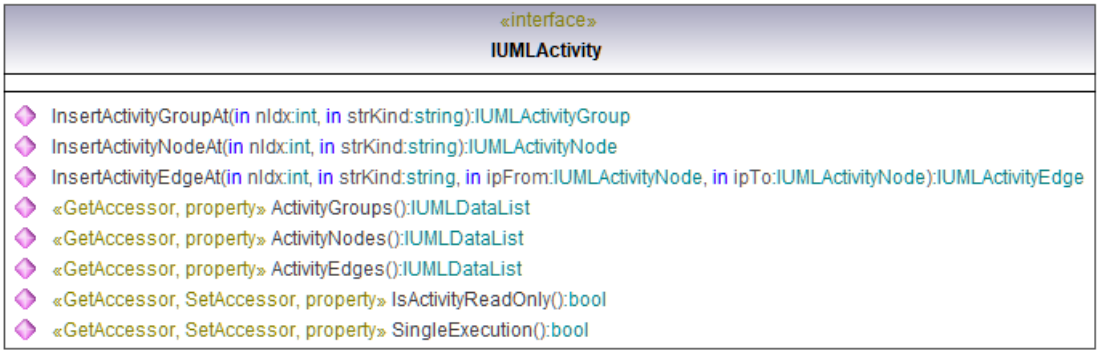
Interface **IUMLActionInputPin**

diagram	

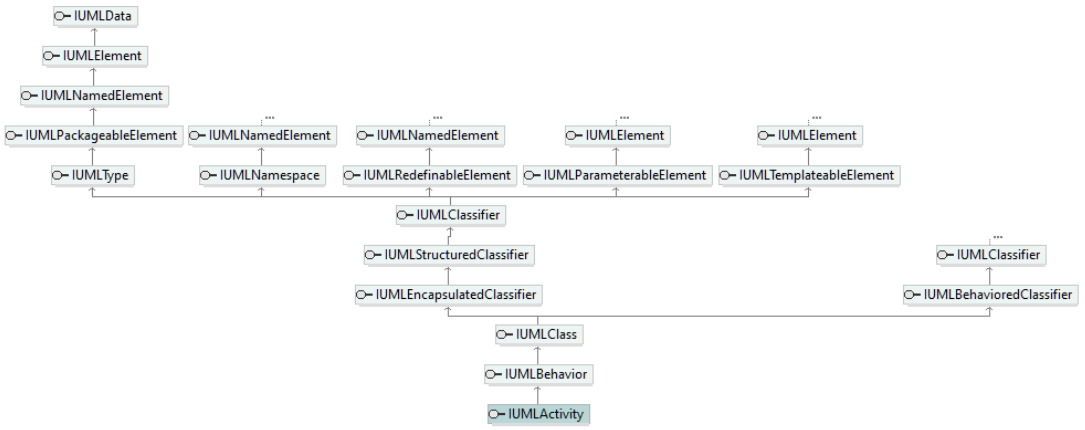
hierarchy	
typedElements	Interface **IUMLAction** 983 Interface **IUMLDataAll** 911 Operation **ActionInputPin** 983 Operation **ActionInputPin** 912

16.4.3.5.6 UModelAPI - IUMLActivity

Interface **IUMLActivity**

diagram	

 © 2019-2025 Altova GmbH

hierarchy	
typedElements	Interface **IUMLActivityEdge** 988 Interface **IUMLActivityGroup** 991 Interface **IUMLDataAll** 911 Operation **Activity** 989 Operation **InActivity** 992 Operation **Activity** 912 **InActivity** 930

Operation **IUMLActivity::ActivityEdges**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			
documentation	A list of elements of type IUMLActivityEdge 988.					

Operation **IUMLActivity::ActivityGroups**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			
documentation	A list of elements of type IUMLActivityGroup 991.					

Operation **IUMLActivity::ActivityNodes**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			
documentation	A list of elements of type IUMLActivityNode 992.					

Operation **IUMLActivity::InsertActivityEdgeAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	strKind	**in**	**string**			
	ipFrom	**in**	**IUMLActivityNode** 992			
	ipTo	**in**	**IUMLActivityNode** 992			
	return	**return**	**IUMLActivityEdge** 988			

Operation **IUMLActivity::InsertActivityGroupAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	strKind	**in**	**string**			
	return	**return**	**IUMLActivityGroup** 991			

Operation **IUMLActivity::InsertActivityNodeAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	strKind	**in**	**string**			
	return	**return**	**IUMLActivityNode** 992			

Operation **IUMLActivity::IsActivityReadOnly**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

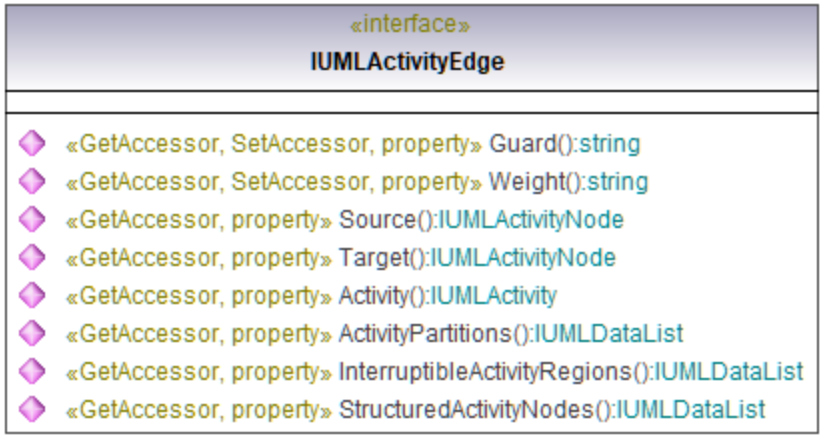
Operation **IUMLActivity::SingleExecution**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

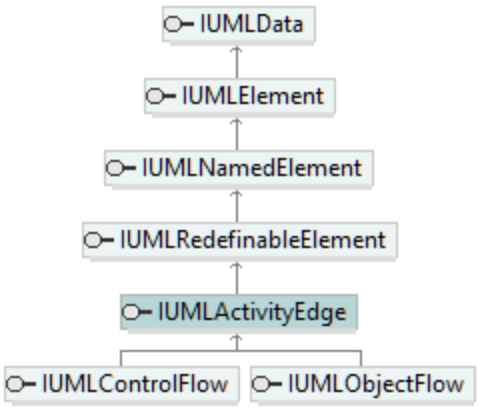
16.4.3.5.7 UModelAPI - IUMLActivityEdge

Interface **IUMLActivityEdge**

diagram



 © 2019-2025 Altova GmbH

hierarchy	
typedElements	Interface **IUMLActivity** 986 Interface **IUMLActivityPartition** 995 Interface **IUMLDataAll** 911 Interface **IUMLInterruptibleActivityRegion** 1098 Interface **IUMLStructuredActivityNode** 1173 Operation **InsertActivityEdgeAt** 987 Operation **InsertEdgeAt** 996 Operation **InsertActivityEdgeAt** 931 **InsertEdgeAt** 934 **InsertInterruptingEdgeAt** 936 Operation **InsertInterruptingEdgeAt** 1098 Operation **InsertEdgeAt** 1174

Operation **IUMLActivityEdge::Activity**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLActivity** 986			

Operation **IUMLActivityEdge::ActivityPartitions**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			
documentation	A list of elements of type IUMLActivityPartition 995.					

Operation **IUMLActivityEdge::Guard**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**string**			

Operation **IUMLActivityEdge::InterruptibleActivityRegions**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			
documentation	A list of elements of type IUMLInterruptibleActivityRegion 1098.					

Operation **IUMLActivityEdge::Source**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLActivityNode** 992			

Operation **IUMLActivityEdge::StructuredActivityNodes**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			
documentation	A list of elements of type IUMLStructuredActivityNode 1173.					

Operation **IUMLActivityEdge::Target**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLActivityNode** 992			

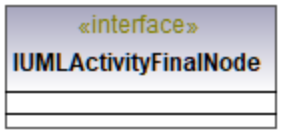
Operation **IUMLActivityEdge::Weight**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**string**			

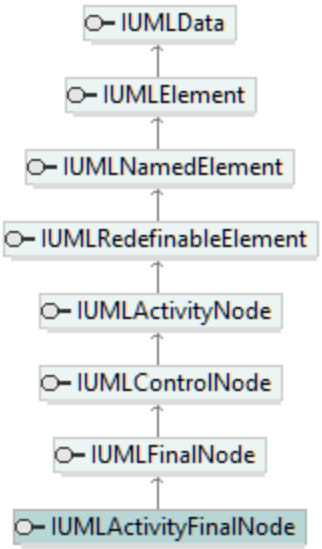
16.4.3.5.8 UModelAPI - IUMLActivityFinalNode

Interface **IUMLActivityFinalNode**

diagram



hierarchy

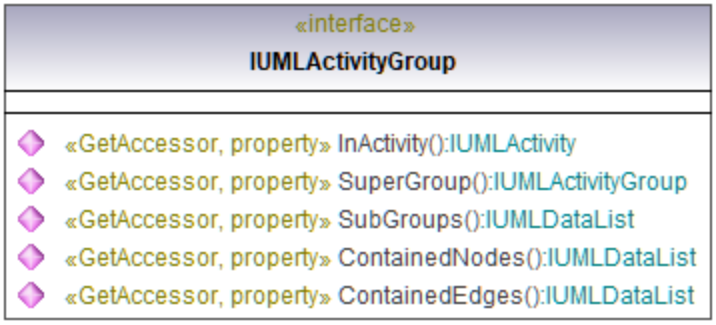
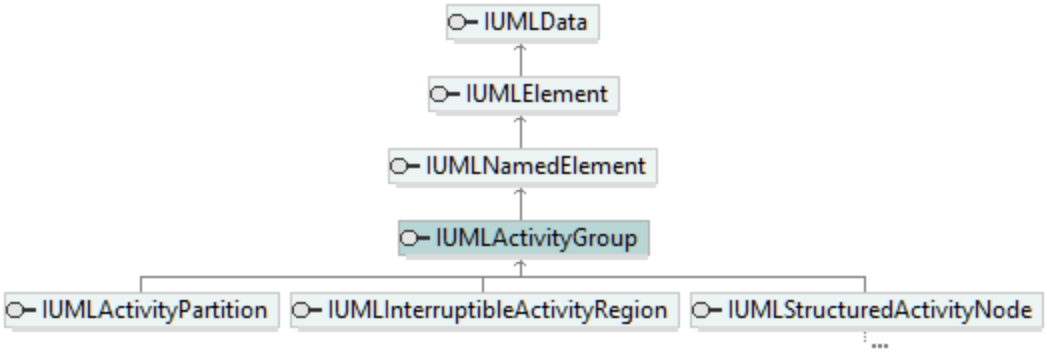


 © 2019-2025 Altova GmbH

UML documentation generated by UModel UML Editor http://www.altova.com/umodel Wed Sep 25 10:33:04 2024

16.4.3.5.9 UModelAPI - IUMLActivityGroup

Interface **IUMLActivityGroup**

diagram	
hierarchy	
typedElements	Interface IUMLActivity 986 Interface IUMLActivityGroup 991 Interface IUMLDataAll 911 Operation InsertActivityGroupAt 987 Operation SuperGroup 992 Operation InsertActivityGroupAt 932 SuperGroup 975

Operation **IUMLActivityGroup::ContainedEdges**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLDataList 906			
documentation	A list of elements of type IUMLActivityEdge 988.					

Operation **IUMLActivityGroup::ContainedNodes**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLDataList 906			

documentation	A list of elements of type IUMLActivityNode 992.

Operation **IUMLActivityGroup::InActivity**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLActivity** 986			

Operation **IUMLActivityGroup::SubGroups**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			
documentation	A list of elements of type IUMLActivityGroup 991.					

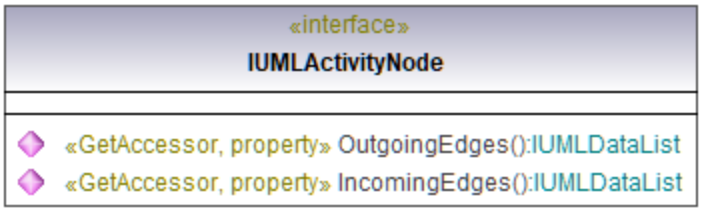
Operation **IUMLActivityGroup::SuperGroup**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLActivityGroup** 991			

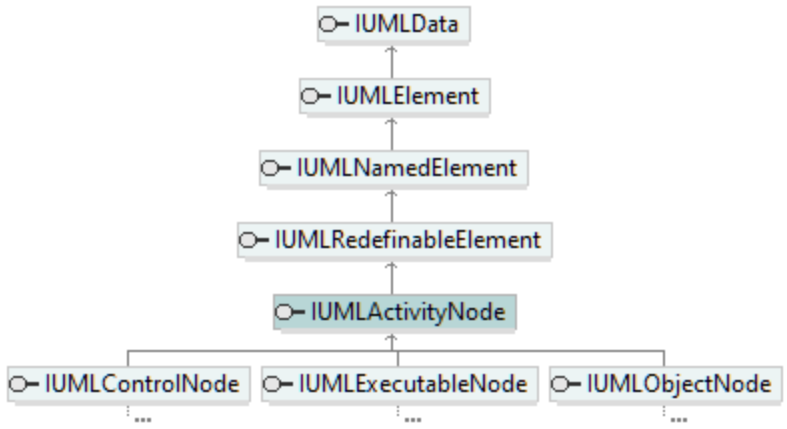
16.4.3.5.10 UModelAPI - IUMLActivityNode

Interface **IUMLActivityNode**

diagram



 © 2019-2025 Altova GmbH

hierarchy	
typedElements	Interface **IUMLActivity** 986 Interface **IUMLActivityEdge** 988 Interface **IUMLActivityPartition** 995 Interface **IUMLDataAll** 911 Interface **IUMLInterruptibleActivityRegion** 1098 Interface **IUMLStructuredActivityNode** 1173 Operation **InsertActivityEdgeAt** 987 **InsertActivityNodeAt** 988 Operation **Source** 989 **Target** 990 Operation **InsertNodeAt** 996 Operation **InsertActivityEdgeAt** 931 **InsertActivityNodeAt** 932 **InsertNodeAt** 938 **Source** 973 **Target** 975 Operation **InsertNodeAt** 1099 Operation **InsertNodeAt** 1174

Operation **IUMLActivityNode::IncomingEdges**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			
documentation	A list of elements of type IUMLActivityEdge 988.					

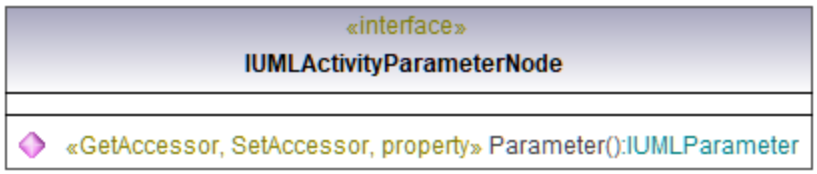
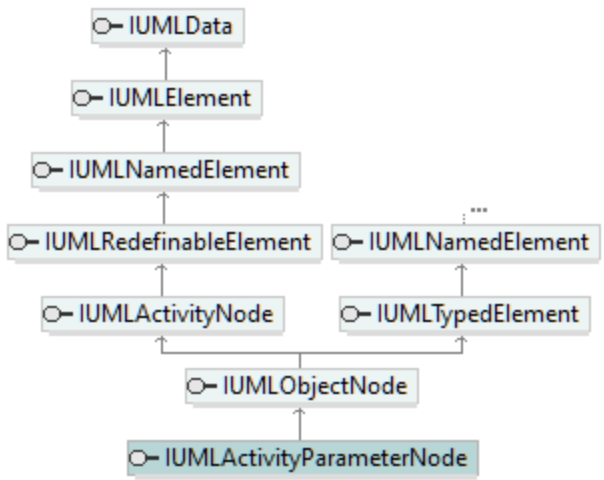
Operation **IUMLActivityNode::OutgoingEdges**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			
documentation	A list of elements of type IUMLActivityEdge 988.					

© 2019-2025 Altova GmbH

16.4.3.5.11 UModelAPI - IUMLActivityParameterNode

Interface **IUMLActivityParameterNode**

diagram	
hierarchy	

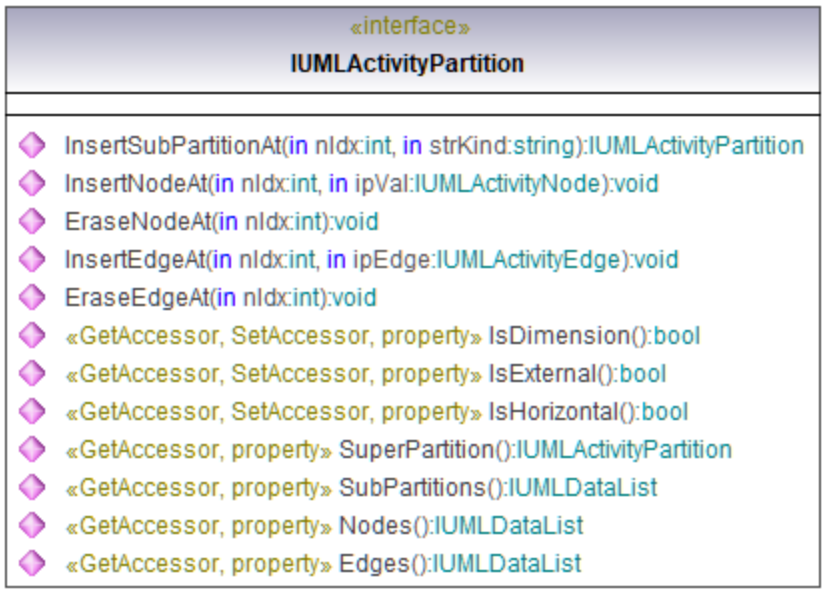
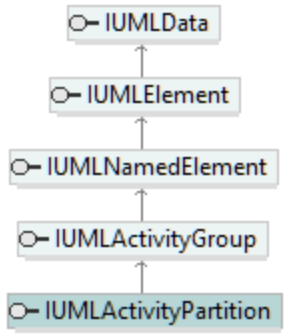
Operation **IUMLActivityParameterNode::Parameter**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLParameter 1140			

© 2019-2025 Altova GmbH

16.4.3.5.12 UModelAPI - IUMLActivityPartition

Interface **IUMLActivityPartition**

diagram	
hierarchy	
typedElements	Interface **IUMLActivityPartition** 995 Interface **IUMLDataAll** 911 Operation **InsertSubPartitionAt** 996 **SuperPartition** 997 Operation **InsertSubPartitionAt** 943 **SuperPartition** 975

Operation **IUMLActivityPartition::Edges**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			
documentation	A list of elements of type IUMLActivityEdge 988.					

Operation **IUMLActivityPartition::EraseEdgeAt**

parameter	name	direction	type	type modifier	multiplicity	default

	nIdx	**in**	**int**
	return	**return**	**void**

Operation **IUMLActivityPartition::EraseNodeAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	**void**			

Operation **IUMLActivityPartition::InsertEdgeAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	ipEdge	**in**	**IUMLActivityEdge** 988			
	return	**return**	**void**			

Operation **IUMLActivityPartition::InsertNodeAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	ipVal	**in**	**IUMLActivityNode** 992			
	return	**return**	**void**			

Operation **IUMLActivityPartition::InsertSubPartitionAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	strKind	**in**	**string**			
	return	**return**	**IUMLActivityPartition** 995			

Operation **IUMLActivityPartition::IsDimension**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IUMLActivityPartition::IsExternal**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IUMLActivityPartition::IsHorizontal**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IUMLActivityPartition::Nodes**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			
documentation	A list of elements of type IUMLActivityNode 992.					

Operation **IUMLActivityPartition::SubPartitions**

© 2019-2025 Altova GmbH

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			
documentation	A list of elements of type IUMLActivityPartition 995.					

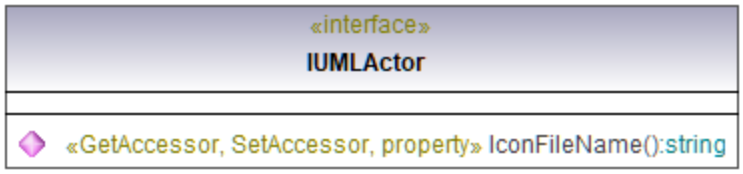
Operation **IUMLActivityPartition::SuperPartition**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLActivityPartition** 995			

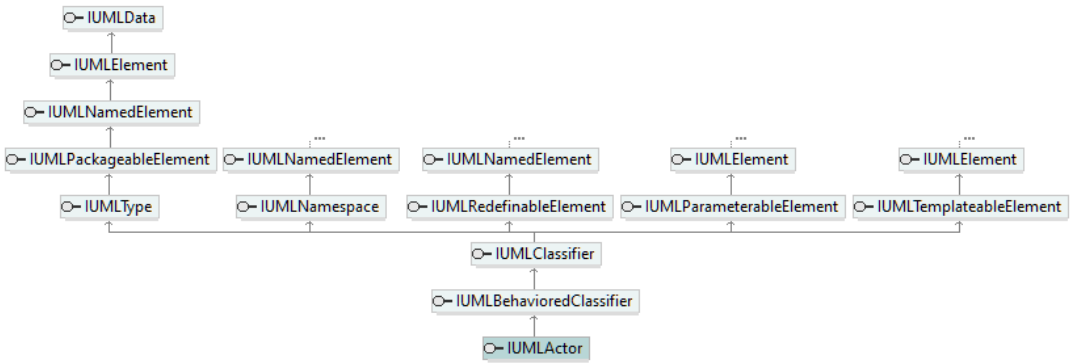
16.4.3.5.13 UModelAPI - IUMLActor

Interface **IUMLActor**

diagram



hierarchy



Operation **IUMLActor::IconFileName**

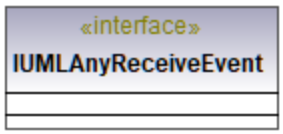
parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**string**			

© 2019-2025 Altova GmbH

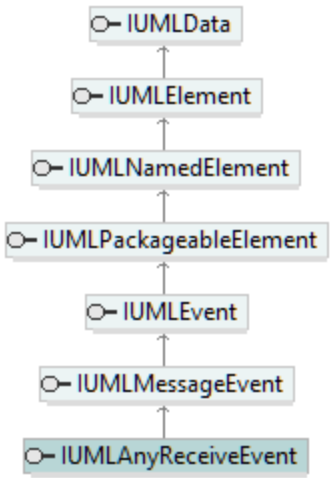
16.4.3.5.14 UModelAPI - IUMLAnyReceiveEvent

Interface **IUMLAnyReceiveEvent**

diagram



hierarchy

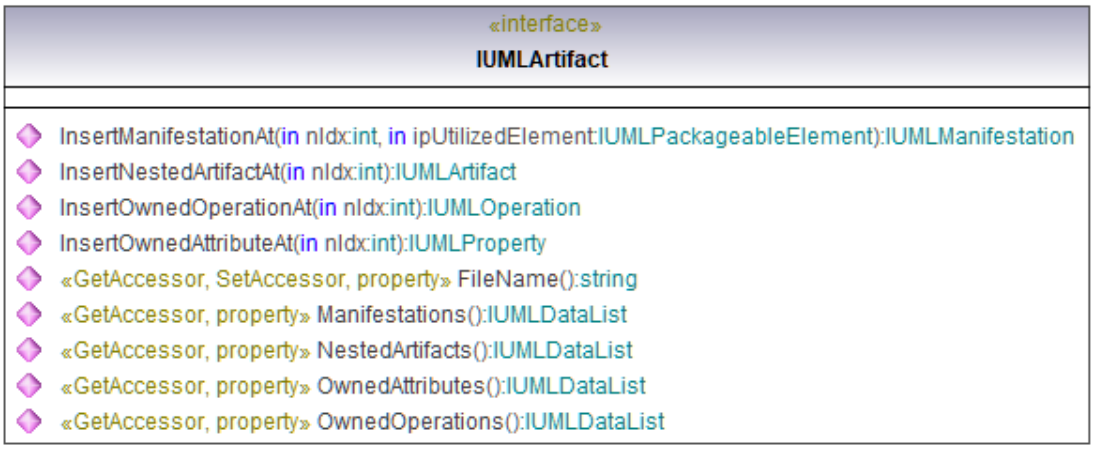


UML documentation generated by **UModel** UML Editor **http://www.altova.com/umodel** Wed Sep 25 10:33:04 2024

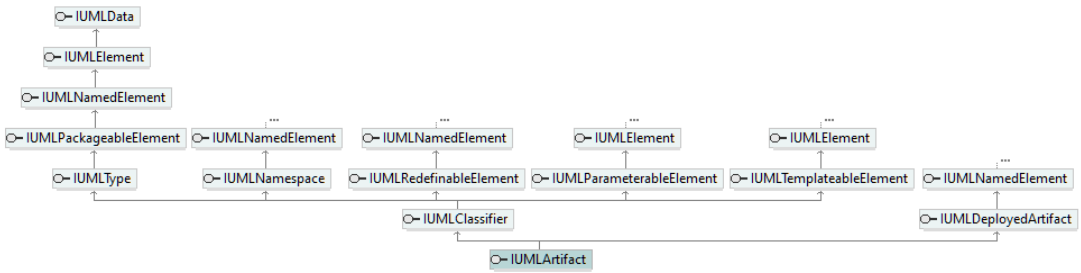
16.4.3.5.15 UModelAPI - IUMLArtifact

Interface **IUMLArtifact**

diagram



© 2019-2025 Altova GmbH

hierarchy	
typedElements	Interface **IUMLArtifact** 998 Interface **IUMLDataAll** 911 Operation **InsertNestedArtifactAt** 999 Operation **InsertNestedArtifactAt** 937

Operation **IUMLArtifact::FileName**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**string**			

Operation **IUMLArtifact::InsertManifestationAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	ipUtilizedElement	**in**	**IUMLPackageableElement** 1137			
	return	**return**	**IUMLManifestation** 1109			

Operation **IUMLArtifact::InsertNestedArtifactAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	**IUMLArtifact** 998			

Operation **IUMLArtifact::InsertOwnedAttributeAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	**IUMLProperty** 1147			

Operation **IUMLArtifact::InsertOwnedOperationAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	**IUMLOperation** 1131			

Operation **IUMLArtifact::Manifestations**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			
documentation	A list of elements of type IUMLManifestation 1109.					

Operation **IUMLArtifact::NestedArtifacts**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			
documentation	A list of elements of type IUMLArtifact 998.					

Operation **IUMLArtifact::OwnedAttributes**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			
documentation	A list of elements of type IUMLProperty 1147.					

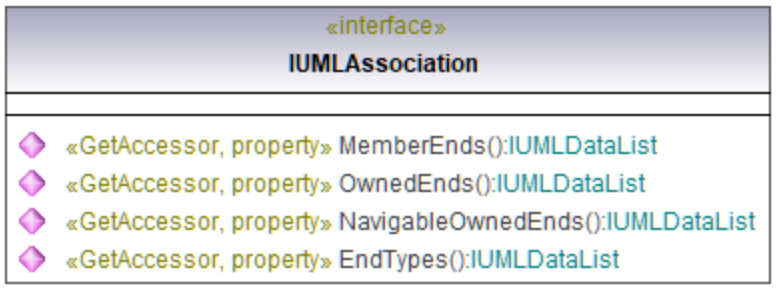
Operation **IUMLArtifact::OwnedOperations**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			
documentation	A list of elements of type IUMLOperation 1131.					

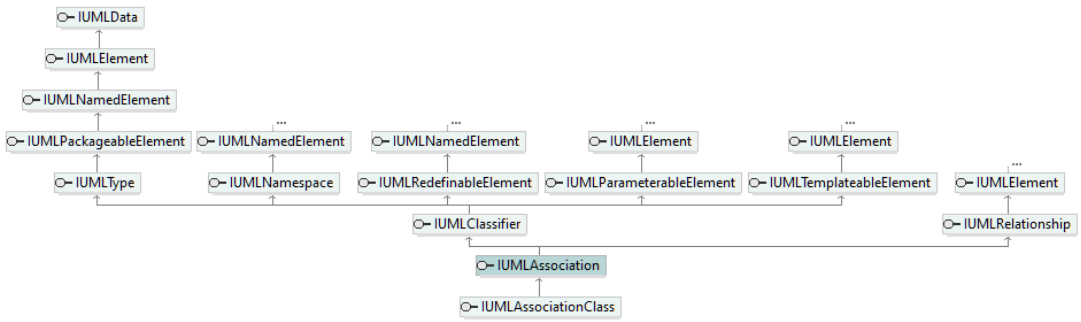
16.4.3.5.16 UModelAPI - IUMLAssociation

Interface **IUMLAssociation**

diagram



hierarchy



 © 2019-2025 Altova GmbH

typedElements	Interface **IUMLConnector** 1033 Interface **IUMLDataAll** 911 Interface **IUMLProperty** 1147 Operation **ConnectorType** 1034 Operation **Association** 915 **ConnectorType** 918 **OwningAssociation** 959 Operation **Association** 1148 **OwningAssociation** 1149

Operation **IUMLAssociation::EndTypes**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			
documentation	A list of elements of type IUMLType 1189.					

Operation **IUMLAssociation::MemberEnds**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			
documentation	A list of elements of type IUMLProperty 1147.					

Operation **IUMLAssociation::NavigableOwnedEnds**

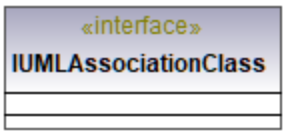
parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			
documentation	A list of elements of type IUMLProperty 1147.					

Operation **IUMLAssociation::OwnedEnds**

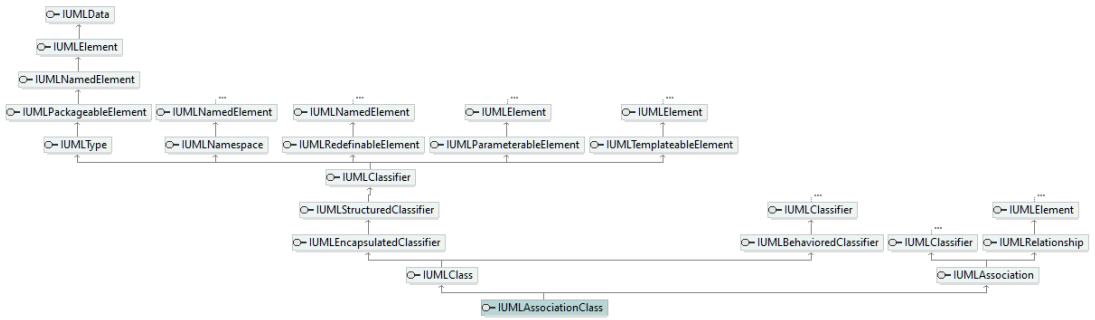
parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			
documentation	A list of elements of type IUMLProperty 1147.					

16.4.3.5.17 UModelAPI - IUMLAssociationClass

Interface **IUMLAssociationClass**

diagram	

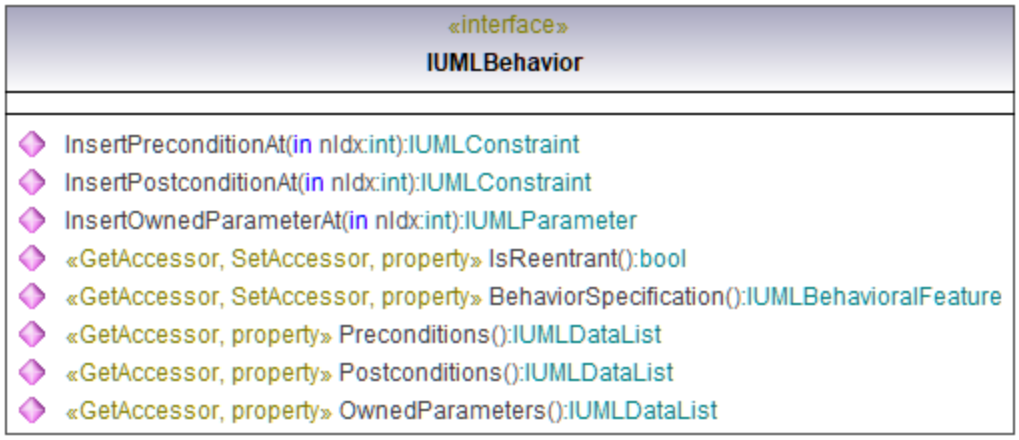
hierarchy



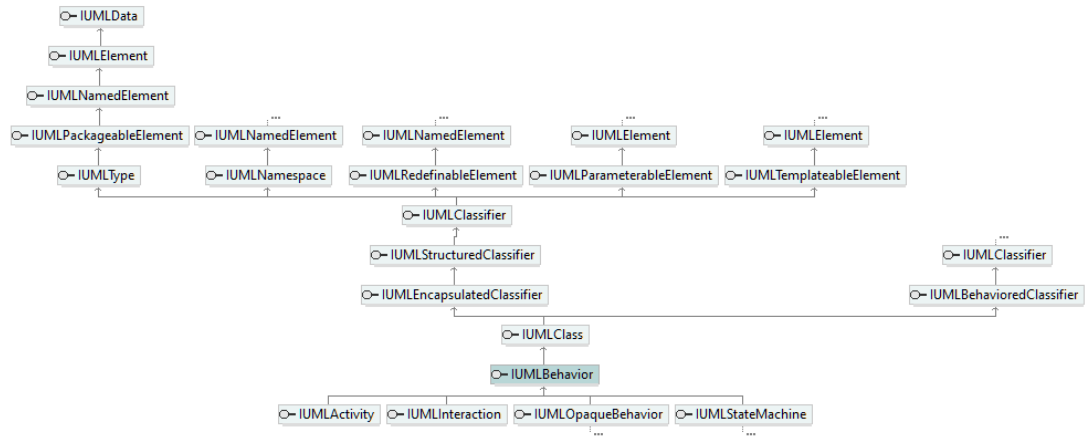
16.4.3.5.18 UModelAPI - IUMLBehavior

Interface **IUMLBehavior**

diagram



hierarchy



 © 2019-2025 Altova GmbH

typedElements		
	Interface **IUMLBehavioredClassifier** 1006	Operation **InsertOwnedBehaviorAt** 1007
	Interface **IUMLBehaviorExecutionSpecification** 1007	Operation **BehaviorExecution** 1008
	Interface **IUMLCallBehaviorAction** 1010	Operation **Behavior** 1010
	Interface **IUMLDataAll** 911	Operation **Behavior** 916 **BehaviorExecution** 916 **DecisionInput** 920 **DoActivity** 921 **Effect** 921 **Entry** 921 **Exit** 925 **InsertOwnedBehaviorAt** 939 **Selection** 965 **SetNewDoActivity** 967 **SetNewEffect** 967 **SetNewEntry** 967 **SetNewExit** 967 **Transformation** 977
	Interface **IUMLDecisionNode** 1041	Operation **DecisionInput** 1041
	Interface **IUMLObjectFlow** 1123	Operation **Transformation** 1124
	Interface **IUMLObjectNode** 1124	Operation **Selection** 1125
	Interface **IUMLState** 1163	Operation **DoActivity** 1164 **Entry** 1164 **Exit** 1164 **SetNewDoActivity** 1165 **SetNewEntry** 1166 **SetNewExit** 1166
	Interface **IUMLTransition** 1187	Operation **Effect** 1187 **SetNewEffect** 1187

Operation **IUMLBehavior::BehaviorSpecification**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLBehavioralFeature** 1004			

Operation **IUMLBehavior::InsertOwnedParameterAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	**IUMLParameter** 1140			

Operation **IUMLBehavior::InsertPostconditionAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	**IUMLConstraint** 1035			

Operation **IUMLBehavior::InsertPreconditionAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	**IUMLConstraint** 1035			

Operation **IUMLBehavior::IsReentrant**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IUMLBehavior::OwnedParameters**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			

documentation	A list of elements of type IUMLParameter 1140.

Operation **IUMLBehavior::Postconditions**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			
documentation	A list of elements of type IUMLConstraint 1035.					

Operation **IUMLBehavior::Preconditions**

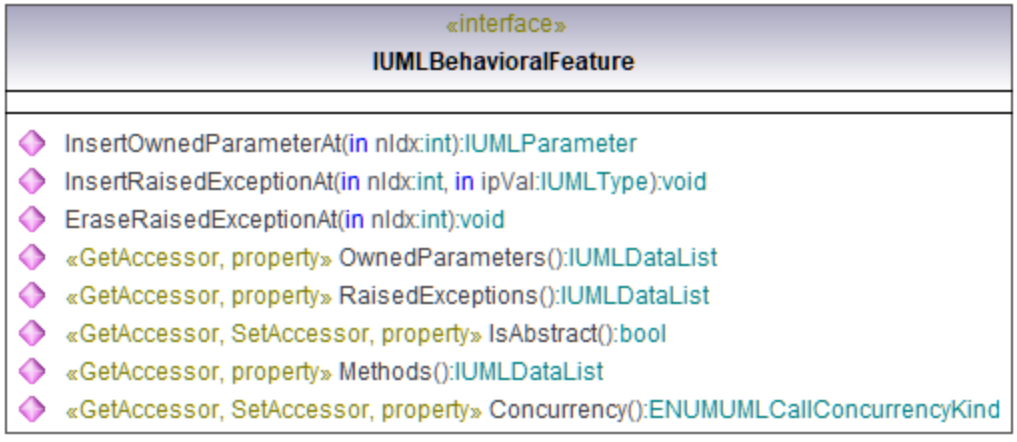
parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			
documentation	A list of elements of type IUMLConstraint 1035.					

UML documentation generated by **UModel** UML Editor **http://www.altova.com/umodel** Wed Sep 25 10:33:04 2024

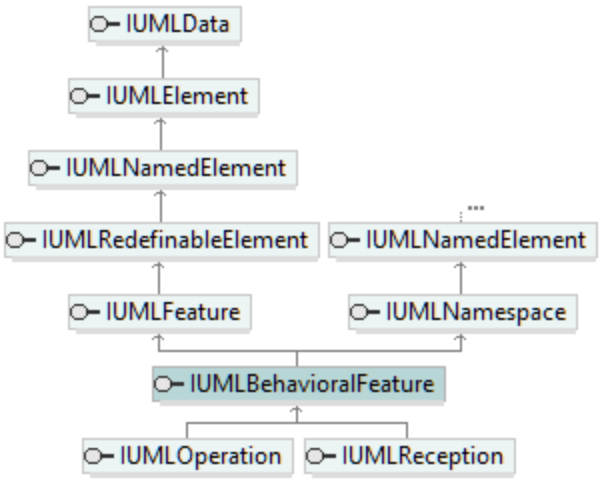
16.4.3.5.19 UModelAPI - IUMLBehavioralFeature

Interface **IUMLBehavioralFeature**

diagram



 © 2019-2025 Altova GmbH

hierarchy	
typedElements	Interface **IUMLBehavior** 1002 Interface **IUMLDataAll** 911 Operation **BehaviorSpecification** 1003 Operation **BehaviorSpecification** 916

Operation **IUMLBehavioralFeature::Concurrency**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**ENUMUMLCallConcurrencyKind** 1264			

Operation **IUMLBehavioralFeature::EraseRaisedExceptionAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	**void**			

Operation **IUMLBehavioralFeature::InsertOwnedParameterAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	**IUMLParameter** 1140			

Operation **IUMLBehavioralFeature::InsertRaisedExceptionAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	ipVal	**in**	**IUMLType** 1189			
	return	**return**	**void**			

Operation **IUMLBehavioralFeature::IsAbstract**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IUMLBehavioralFeature::Methods**

parameter	name	direction	type	type modifier	multiplicity	default

	return	**return**	**IUMLDataList** 906
documentation	A list of elements of type IUMLBehavior 1002.		

Operation **IUMLBehavioralFeature::OwnedParameters**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			
documentation	A list of elements of type IUMLParameter 1140.					

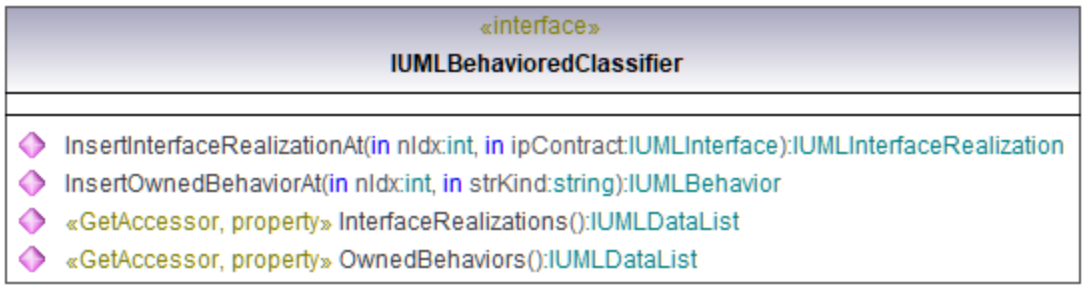
Operation **IUMLBehavioralFeature::RaisedExceptions**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			
documentation	A list of elements of type IUMLType 1189.					

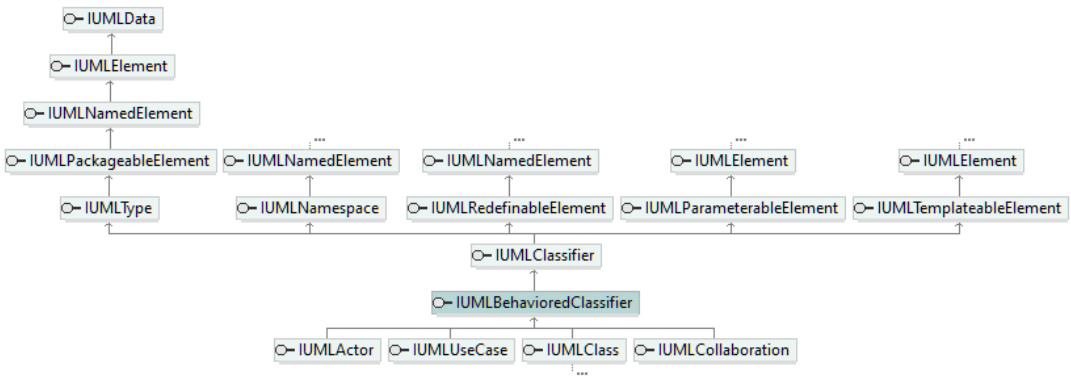
16.4.3.5.20 UModelAPI - IUMLBehavioredClassifier

Interface **IUMLBehavioredClassifier**

diagram



hierarchy



 © 2019-2025 Altova GmbH

typedElements	Interface **IUMLDataAll** 911 Operation **ImplementingClassifier** 930 Interface **IUMLInterfaceRealization** 1097 Operation **ImplementingClassifier** 1097

Operation **IUMLBehavioredClassifier::InsertInterfaceRealizationAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	ipContract	**in**	**IUMLInterface** 1094			
	return	**return**	**IUMLInterfaceRealization** 1097			

Operation **IUMLBehavioredClassifier::InsertOwnedBehaviorAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	strKind	**in**	**string**			
	return	**return**	**IUMLBehavior** 1002			

Operation **IUMLBehavioredClassifier::InterfaceRealizations**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			
documentation	A list of elements of type IUMLInterfaceRealizations 1097.					

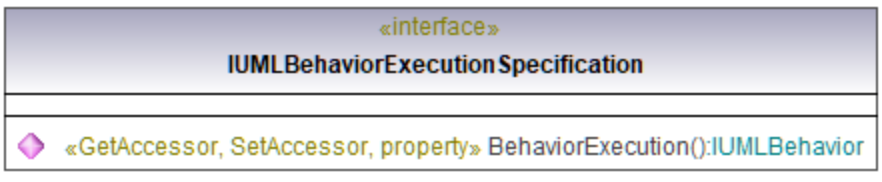
Operation **IUMLBehavioredClassifier::OwnedBehaviors**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			
documentation	A list of elements of type IUMLBehavior 1002.					

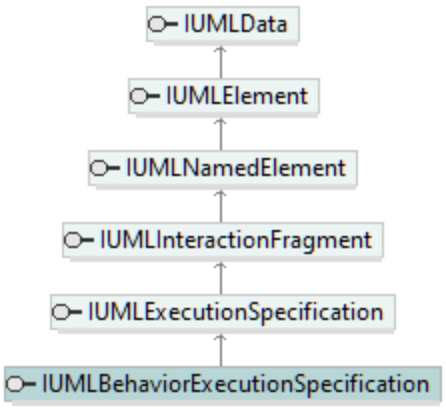
UML documentation generated by **UModel** UML Editor **http://www.altova.com/umodel** Wed Sep 25 10:33:04 2024

16.4.3.5.21 UModelAPI - IUMLBehaviorExecutionSpecification

Interface **IUMLBehaviorExecutionSpecification**

diagram	

hierarchy	



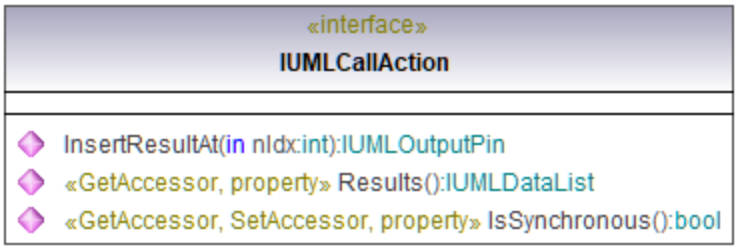
Operation **IUMLBehaviorExecutionSpecification::BehaviorExecution**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLBehavior 1002			

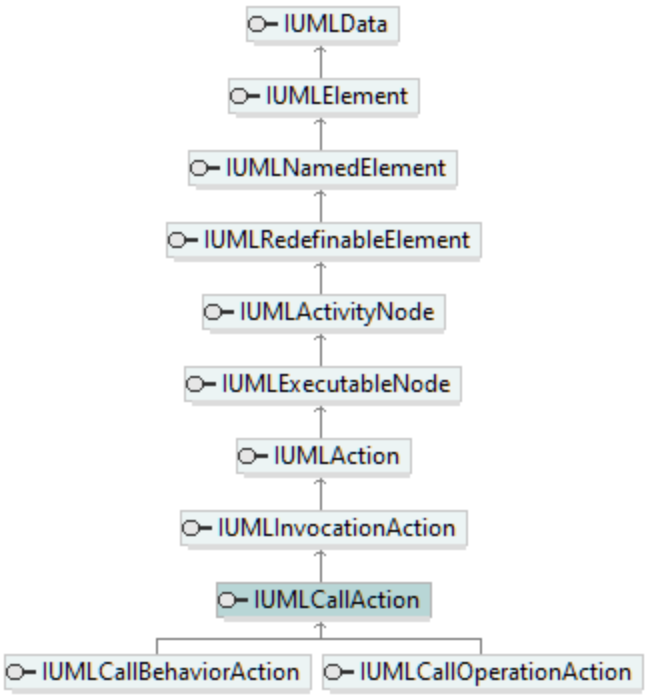
16.4.3.5.22 UModelAPI - IUMLCallAction

Interface **IUMLCallAction**

diagram	



 © 2019-2025 Altova GmbH

hierarchy	

Operation **IUMLCallAction::InsertResultAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	**IUMLOutputPin** 1133			

Operation **IUMLCallAction::IsSynchronous**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

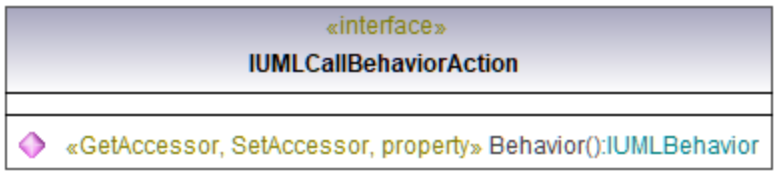
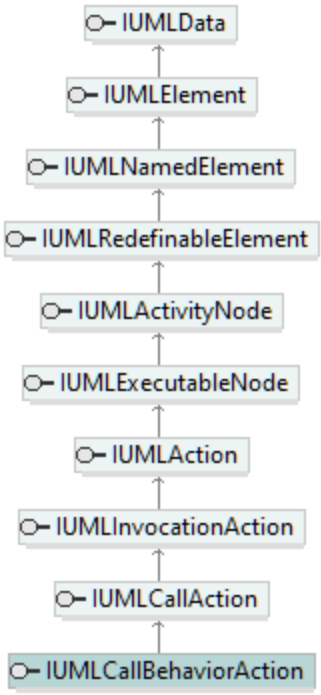
Operation **IUMLCallAction::Results**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			
documentation	A list of elements of type IUMLOutputPin 1133.					

© 2019-2025 Altova GmbH

16.4.3.5.23 UModelAPI - IUMLCallBehaviorAction

Interface **IUMLCallBehaviorAction**

diagram	
hierarchy	

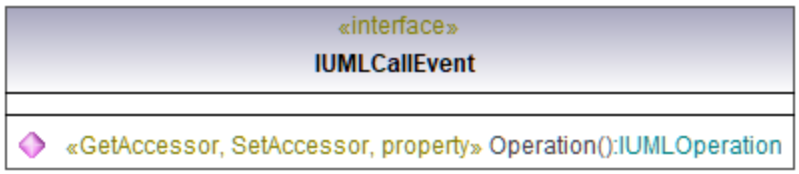
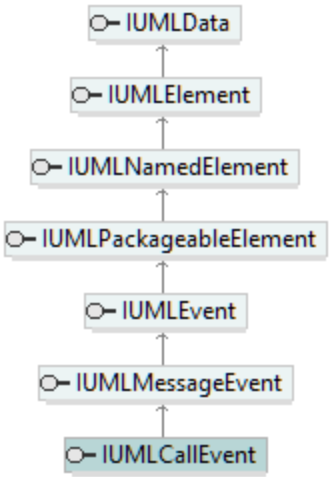
Operation **IUMLCallBehaviorAction::Behavior**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLBehavior 1002			

© 2019-2025 Altova GmbH

16.4.3.5.24 UModelAPI - IUMLCallEvent

Interface **IUMLCallEvent**

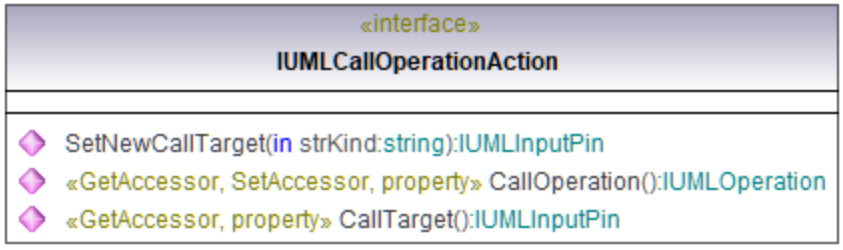
diagram	
hierarchy	

Operation **IUMLCallEvent::Operation**

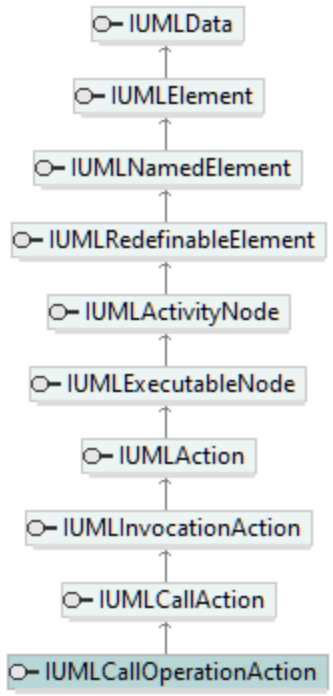
parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLOperation 1131			

16.4.3.5.25 UModelAPI - IUMLCallOperationAction

Interface **IUMLCallOperationAction**

diagram	

hierarchy	



Operation **IUMLCallOperationAction::CallOperation**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLOperation** 1131			

Operation **IUMLCallOperationAction::CallTarget**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLInputPin** 1083			

Operation **IUMLCallOperationAction::SetNewCallTarget**

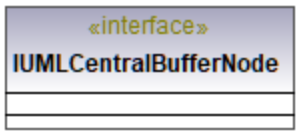
parameter	name	direction	type	type modifier	multiplicity	default
	strKind	**in**	**string**			
	return	**return**	**IUMLInputPin** 1083			

© 2019-2025 Altova GmbH

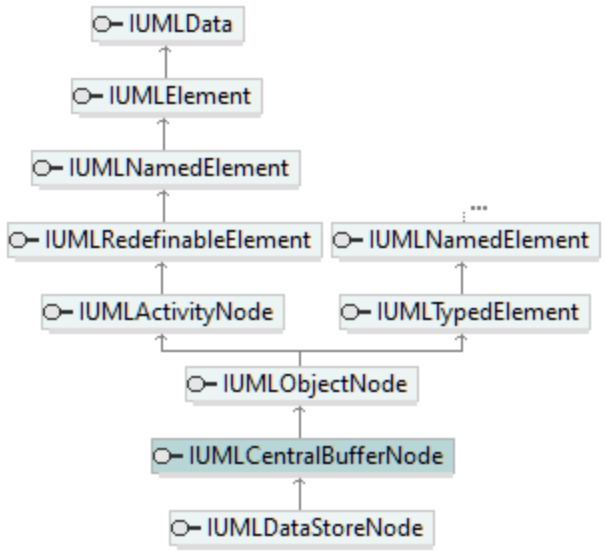
16.4.3.5.26 UModelAPI - IUMLCentralBufferNode

Interface **IUMLCentralBufferNode**

diagram



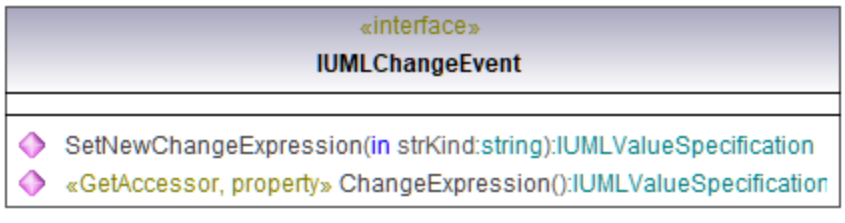
hierarchy

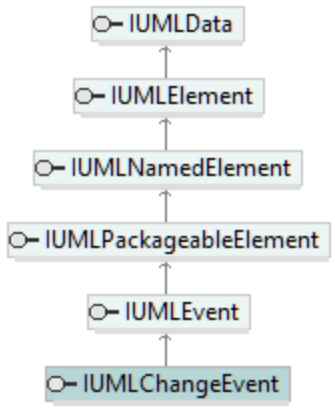


16.4.3.5.27 UModelAPI - IUMLChangeEvent

Interface **IUMLChangeEvent**

diagram



hierarchy	
	

Operation **IUMLChangeEvent::ChangeExpression**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLValueSpecification** 1195			

Operation **IUMLChangeEvent::SetNewChangeExpression**

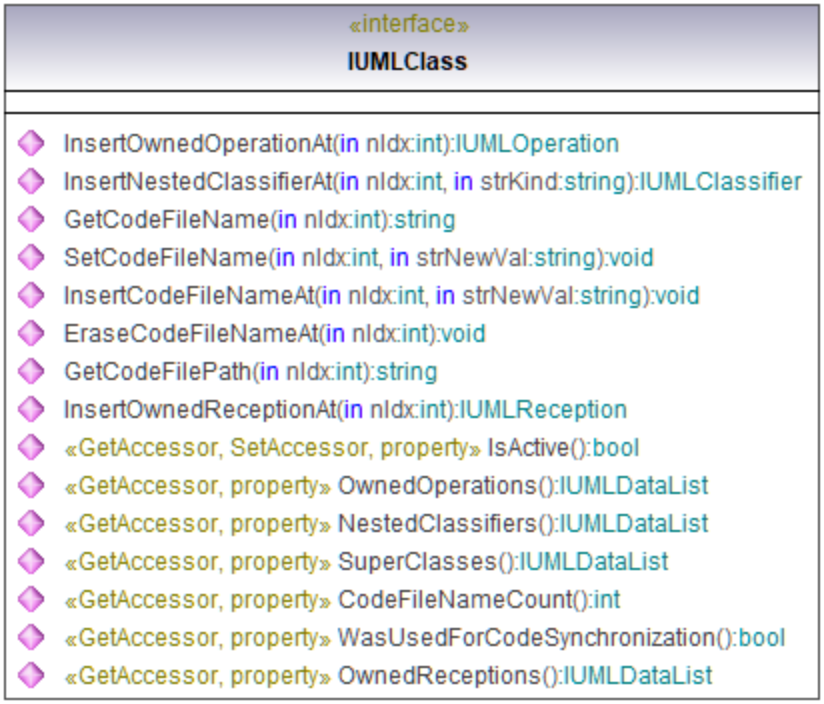
parameter	name	direction	type	type modifier	multiplicity	default
	strKind	**in**	**string**			
	return	**return**	**IUMLValueSpecification** 1195			

© 2019-2025 Altova GmbH

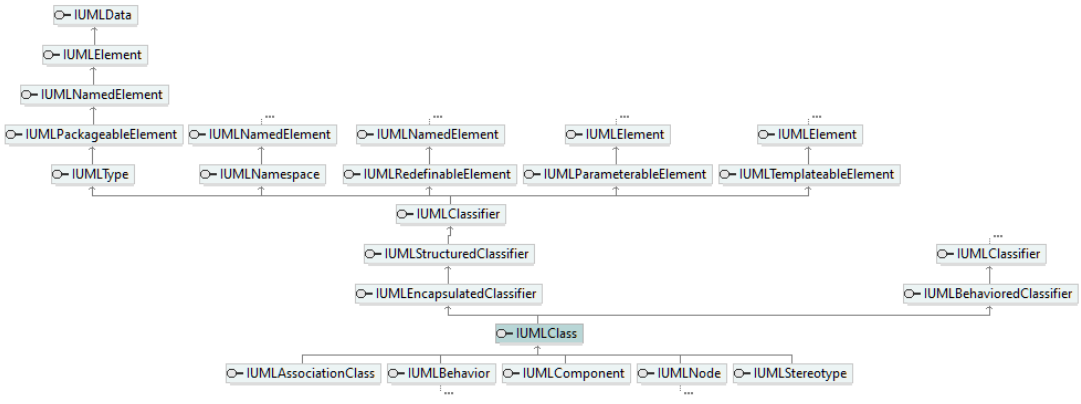
16.4.3.5.28 UModelAPI - IUMLClass

Interface **IUMLClass**

diagram



hierarchy



typedElements

Interface **IUMLClassifier** 1018	Operation **Class** 1019
Interface **IUMLDataAll** 911	Operation **Class** 917
Interface **IUMLOperation** 1131	Operation **Class** 1132
Interface **IUMLProperty** 1147	Operation **Class** 1148
Interface **IUMLReception** 1154	Operation **Class** 1155

Operation **IUMLClass::CodeFileNameCount**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**int**			

Operation **IUMLClass::EraseCodeFileNameAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	**void**			

Operation **IUMLClass::GetCodeFileName**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	**string**			

Operation **IUMLClass::GetCodeFilePath**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	**string**			
documentation	get the full code file path					

Operation **IUMLClass::InsertCodeFileNameAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	strNewVal	**in**	**string**			
	return	**return**	**void**			

Operation **IUMLClass::InsertNestedClassifierAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	strKind	**in**	**string**			
	return	**return**	**IUMLClassifier** 1018			

Operation **IUMLClass::InsertOwnedOperationAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	**IUMLOperation** 1131			

Operation **IUMLClass::InsertOwnedReceptionAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	**IUMLReception** 1154			

Operation **IUMLClass::IsActive**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IUMLClass::NestedClassifiers**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			

© 2019-2025 Altova GmbH

documentation	A list of elements of type IUMLClassifier[1018].

Operation **IUMLClass::OwnedOperations**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList**[906]			
documentation	A list of elements of type IUMLOperation[1131].					

Operation **IUMLClass::OwnedReceptions**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList**[906]			

Operation **IUMLClass::SetCodeFileName**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	strNewVal	**in**	**string**			
	return	**return**	**void**			

Operation **IUMLClass::SuperClasses**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList**[906]			
documentation	A list of elements of type IUMLClass[1015].					

Operation **IUMLClass::WasUsedForCodeSynchronization**

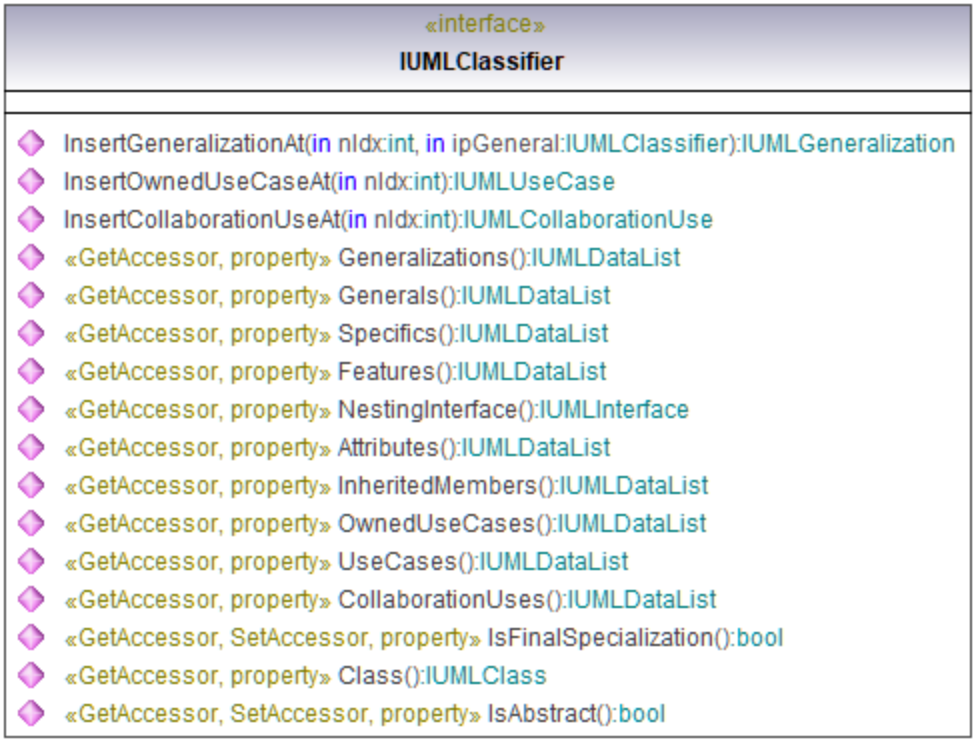
parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

© 2019-2025 Altova GmbH

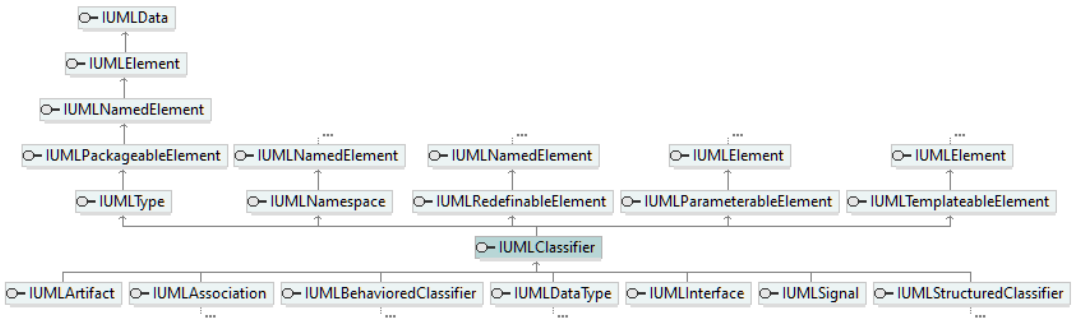
16.4.3.5.29 UModelAPI - IUMLClassifier

Interface **IUMLClassifier**

diagram



hierarchy



typedElements

Interface **IUMLAction** 983
Interface **IUMLClass** 1015
Interface **IUMLClassifier** 1018
Interface **IUMLClassifierTemplateParameter** 1021
Interface **IUMLComponent** 1028
Interface **IUMLComponentRealization** 1029
Interface **IUMLDataAll** 911

Operation **ActionContext** 983
Operation **InsertNestedClassifierAt** 1016
Operation **InsertGeneralizationAt** 1020
Operation **InsertConstrainingClassifierAt** 1022
Operation **InsertRealizationAt** 1029
Operation **RealizingClassifier** 1030
Operation **ActionContext** 911 **Classifier** 917 **General** 927 **InsertConstrainingClassifierAt** 933 **InsertConveyedAt** 933

© 2019-2025 Altova GmbH

	Interface **IUMLExceptionHandler** 1059 Interface **IUMLGeneralization** 1074 Interface **IUMLInformationFlow** 1080 Interface **IUMLInstanceSpecification** 1085 Interface **IUMLInterface** 1094 Interface **IUMLProperty** 1147 Interface **IUMLUseCase** 1192	**InsertExceptionTypeAt** 934 **InsertGeneralizationAt** 935 **InsertNestedClassifierAt** 937 **InsertRealizationAt** 942 **InsertSubjectAt** 943 **RealizingClassifier** 963 **Specific** 973 Operation **InsertExceptionTypeAt** 1060 Operation **General** 1075 **Specific** 1075 Operation **InsertConveyedAt** 1081 Operation **Classifier** 1085 Operation **InsertNestedClassifierAt** 1095 Operation **Classifier** 1148 Operation **InsertSubjectAt** 1193

Operation **IUMLClassifier::Attributes**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			
documentation	A list of elements of type IUMLProperty 1147.					

Operation **IUMLClassifier::Class**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLClass** 1015			

Operation **IUMLClassifier::CollaborationUses**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			
documentation	A list of elements of type IUMLCollaborationUse 1023.					

Operation **IUMLClassifier::Features**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			
documentation	A list of elements of type IUMLFeature 1069.					

Operation **IUMLClassifier::Generalizations**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			
documentation	A list of elements of type IUMLGeneralization 1074.					

Operation **IUMLClassifier::Generals**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			
documentation	A list of elements of type IUMLClassifier 1018.					

Operation **IUMLClassifier::InheritedMembers**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			
documentation	A list of elements of type IUMLNamedElement 1117.					

Operation **IUMLClassifier::InsertCollaborationUseAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	**IUMLCollaborationUse** 1023			

Operation **IUMLClassifier::InsertGeneralizationAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	ipGeneral	**in**	**IUMLClassifier** 1018			
	return	**return**	**IUMLGeneralization** 1074			

Operation **IUMLClassifier::InsertOwnedUseCaseAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	**IUMLUseCase** 1192			

Operation **IUMLClassifier::IsAbstract**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IUMLClassifier::IsFinalSpecialization**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IUMLClassifier::NestingInterface**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLInterface** 1094			

Operation **IUMLClassifier::OwnedUseCases**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			
documentation	A list of elements of type IUMLUseCase 1192.					

Operation **IUMLClassifier::Specifics**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			

© 2019-2025 Altova GmbH

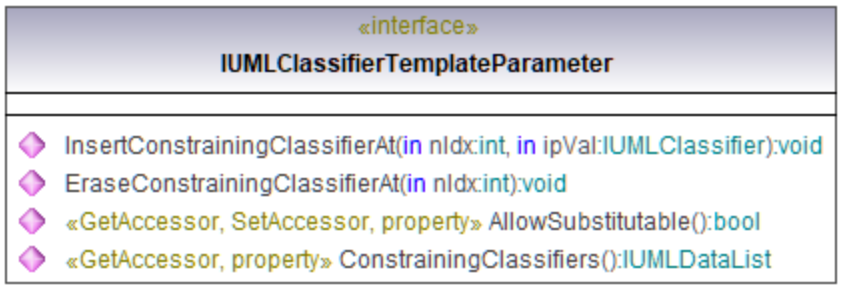
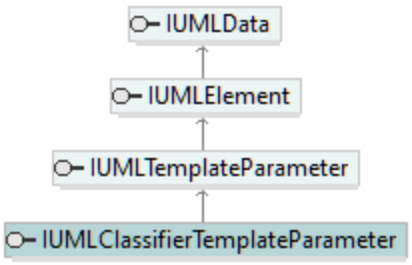
documentation	A list of elements of type IUMLClassifier[1018].

Operation **IUMLClassifier::UseCases**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList**[906]			
documentation	A list of elements of type IUMLUseCase[1192].					

16.4.3.5.30 UModelAPI - IUMLClassifierTemplateParameter

Interface **IUMLClassifierTemplateParameter**

diagram	
hierarchy	
typedElements	Interface **IUMLDataAll**[911] Interface **IUMLTemplateSignature**[1181] Operation **InsertOwnedTemplateParameterAt**[941] Operation **InsertOwnedTemplateParameterAt**[1181]

Operation **IUMLClassifierTemplateParameter::AllowSubstitutable**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IUMLClassifierTemplateParameter::ConstrainingClassifiers**

parameter	name	direction	type	type modifier	multiplicity	default

	return	**return**	**IUMLDataList** 906
documentation	A list of elements of type IUMLClassifier 1018.		

Operation **IUMLClassifierTemplateParameter::EraseConstrainingClassifierAt**

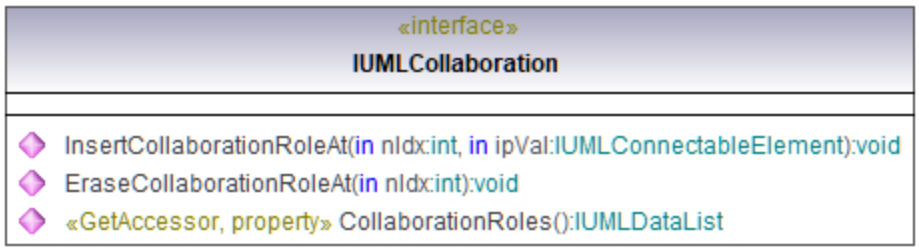
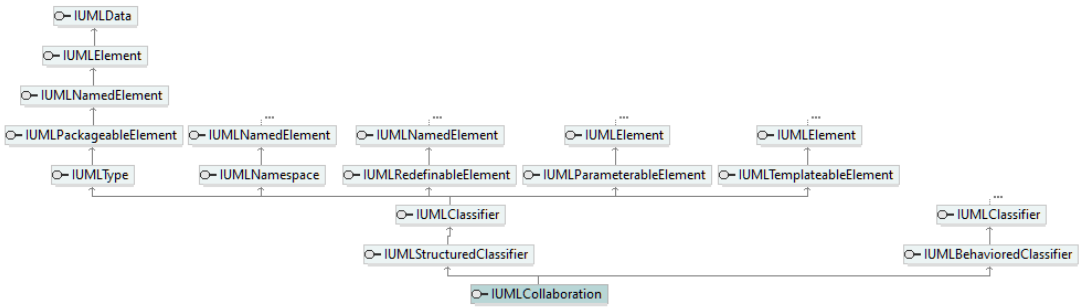
parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	**void**			

Operation **IUMLClassifierTemplateParameter::InsertConstrainingClassifierAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	ipVal	**in**	**IUMLClassifier** 1018			
	return	**return**	**void**			

16.4.3.5.31 UModelAPI - IUMLCollaboration

Interface **IUMLCollaboration**

diagram	
hierarchy	
typedElements	Interface **IUMLCollaborationUse** 1023 Interface **IUMLDataAll** 911 Operation **CollaborationType** 1023 Operation **CollaborationType** 918		

Operation **IUMLCollaboration::CollaborationRoles**

parameter	name	direction	type	type modifier	multiplicity	default

 © 2019-2025 Altova GmbH

	return	**return**	**IUMLDataList** 906
documentation	A list of elements of type IUMLCollaboration 1022.		

Operation **IUMLCollaboration::EraseCollaborationRoleAt**

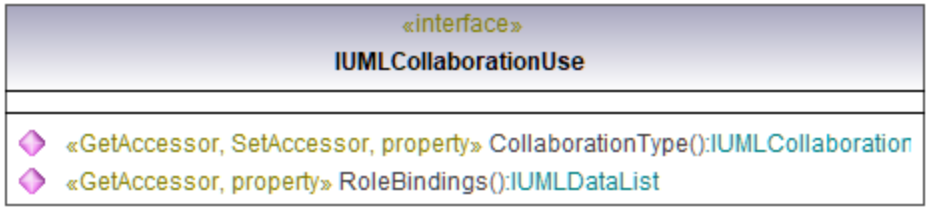
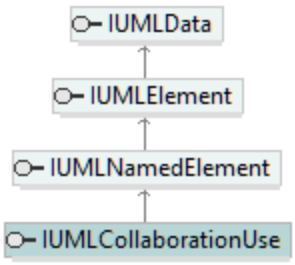
parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	**void**			

Operation **IUMLCollaboration::InsertCollaborationRoleAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	ipVal	**in**	**IUMLConnectableElement** 1030			
	return	**return**	**void**			

16.4.3.5.32 UModelAPI - IUMLCollaborationUse

Interface **IUMLCollaborationUse**

diagram	
hierarchy	
typedElements	Interface **IUMLClassifier** 1018 Interface **IUMLDataAll** 911	Operation **InsertCollaborationUseAt** 1020 Operation **InsertCollaborationUseAt** 933

Operation **IUMLCollaborationUse::CollaborationType**

parameter	name	direction	type	type modifier	multiplicity	default

	return	**return**	**IUMLCollaboration** 1022

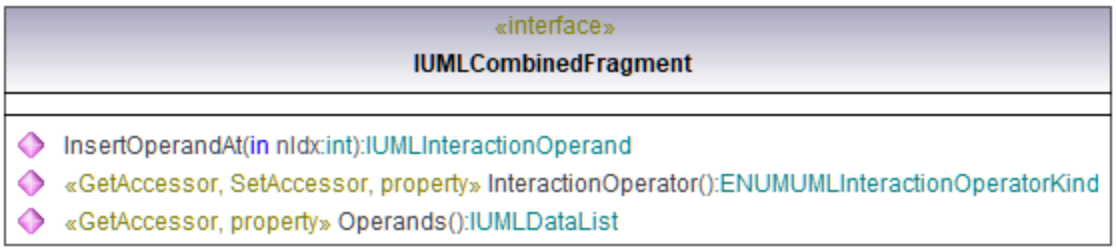
Operation **IUMLCollaborationUse::RoleBindings**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			
documentation	A list of elements of type IUMLDependency 1041.					

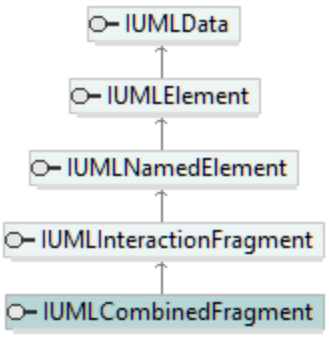
16.4.3.5.33 UModelAPI - IUMLCombinedFragment

Interface **IUMLCombinedFragment**

diagram



hierarchy



Operation **IUMLCombinedFragment::InsertOperandAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	**IUMLInteractionOperand** 1091			

Operation **IUMLCombinedFragment::InteractionOperator**

parameter	name	direction	type	type modifier	multiplicity	default

 © 2019-2025 Altova GmbH

	return	**return**	**ENUMUMLInteractionOperatorKind** 1269

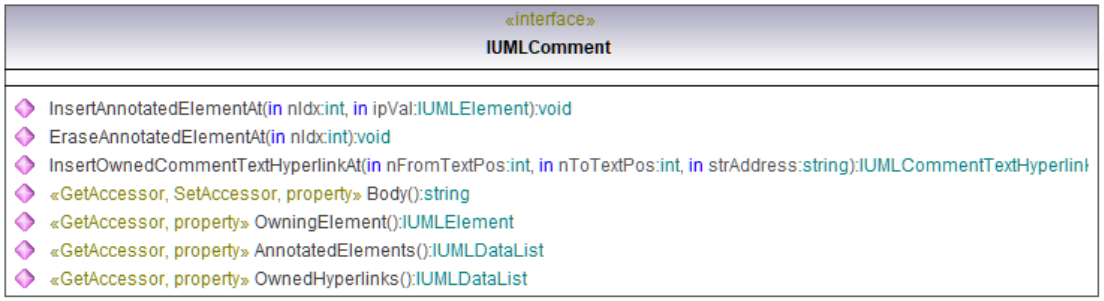
Operation **IUMLCombinedFragment::Operands**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			
documentation	A list of elements of type IUMLInteractionOperand 1091.					

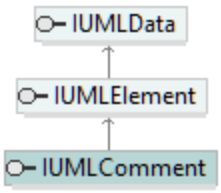
16.4.3.5.34 UModelAPI - IUMLComment

Interface **IUMLComment**

diagram



hierarchy



typedElements

Interface **IUMLDataAll** 911

Interface **IUMLElement** 1050

Operation **InsertOwnedCommentAt** 939
OwnedDocComment 957

Operation **InsertOwnedCommentAt** 1052
OwnedDocComment 1053

Operation **IUMLComment::AnnotatedElements**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			
documentation	A list of elements of type IUMLElement 1050.					

Operation **IUMLComment::Body**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**string**			

Operation **IUMLComment::EraseAnnotatedElementAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	**void**			

Operation **IUMLComment::InsertAnnotatedElementAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	ipVal	**in**	**IUMLElement** 1050			
	return	**return**	**void**			

Operation **IUMLComment::InsertOwnedCommentTextHyperlinkAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nFromTextPos	**in**	**int**			
	nToTextPos	**in**	**int**			
	strAddress	**in**	**string**			
	return	**return**	**IUMLCommentTextHyperlink** 1026			

Operation **IUMLComment::OwnedHyperlinks**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			

Operation **IUMLComment::OwningElement**

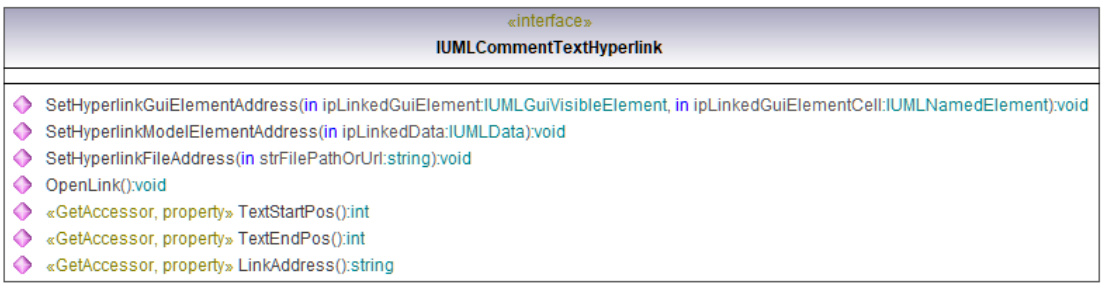
parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLElement** 1050			

UML documentation generated by **UModel** UML Editor **http://www.altova.com/umodel** Wed Sep 25 10:33:04 2024

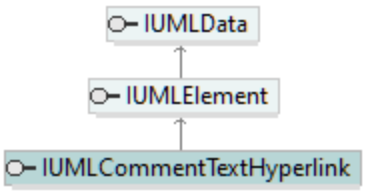
16.4.3.5.35 UModelAPI - IUMLCommentTextHyperlink

Interface **IUMLCommentTextHyperlink**

diagram



 © 2019-2025 Altova GmbH

hierarchy	
typedElements	Interface **IUMLComment** 1025 Interface **IUMLDataAll** 911 Operation **InsertOwnedCommentTextHyperlinkAt** 1026 Operation **InsertOwnedCommentTextHyperlinkAt** 939

Operation **IUMLCommentTextHyperlink::LinkAddress**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**string**			

Operation **IUMLCommentTextHyperlink::OpenLink**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**void**			

Operation **IUMLCommentTextHyperlink::SetHyperlinkFileAddress**

parameter	name	direction	type	type modifier	multiplicity	default
	strFilePathOrUrl	**in**	**string**			
	return	**return**	**void**			

Operation **IUMLCommentTextHyperlink::SetHyperlinkGuiElementAddress**

parameter	name	direction	type	type modifier	multiplicity	default
	ipLinkedGuiElemein nt		**IUMLGuiVisibleElement** 1260			
	ipLinkedGuiElemein ntCell		**IUMLNamedElement** 1117			
	return	**return**	**void**			

Operation **IUMLCommentTextHyperlink::SetHyperlinkModelElementAddress**

parameter	name	direction	type	type modifier	multiplicity	default
	ipLinkedData	**in**	**IUMLData** 904			
	return	**return**	**void**			

Operation **IUMLCommentTextHyperlink::TextEndPos**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**int**			

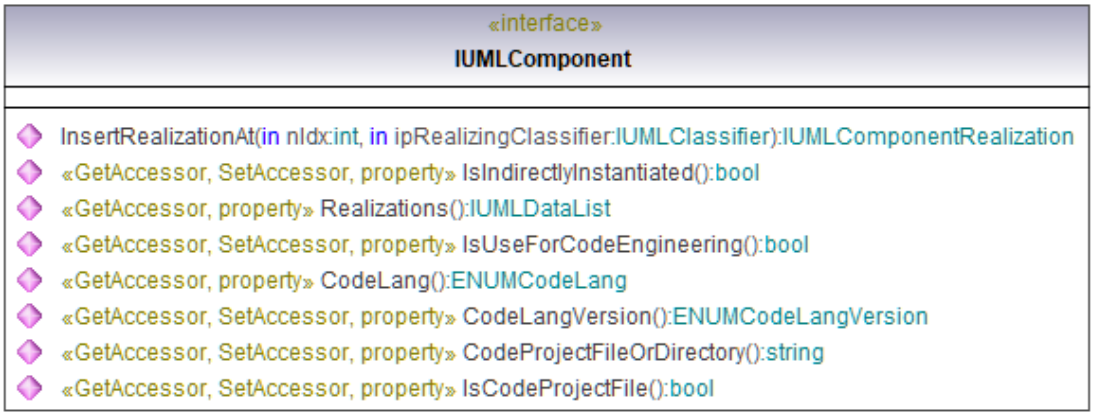
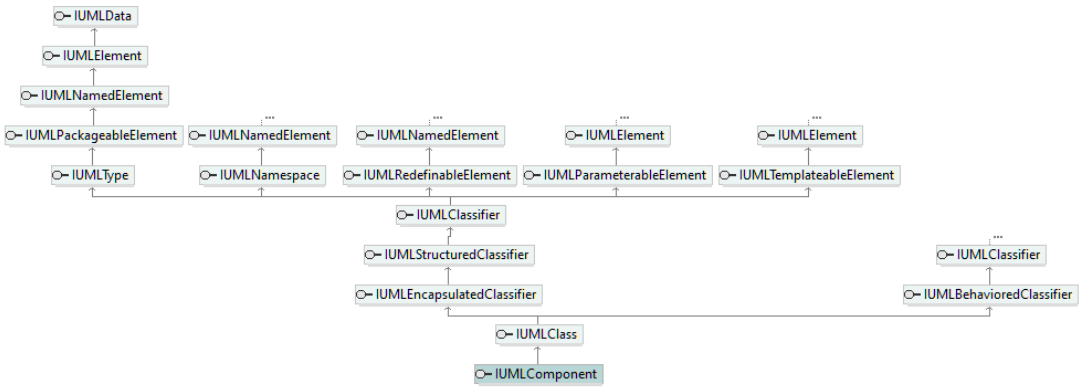
Operation **IUMLCommentTextHyperlink::TextStartPos**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**int**			

UML documentation generated by UModel UML Editor http://www.altova.com/umodel Wed Sep 25 10:33:04 2024

16.4.3.5.36 UModelAPI - IUMLComponent

Interface **IUMLComponent**

diagram	
hierarchy	
typedElements	Interface **IUMLComponentRealization** 1029 Interface **IUMLDataAll** 911 Operation **Abstraction** 1030 Operation **Abstraction** 911

Operation **IUMLComponent::CodeLang**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**ENUMCodeLang** 897			

Operation **IUMLComponent::CodeLangVersion**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**ENUMCodeLangVersion** 898			

Operation **IUMLComponent::CodeProjectFileOrDirectory**

parameter	name	direction	type	type modifier	multiplicity	default

 © 2019-2025 Altova GmbH

	return	**return**	**string**

Operation **IUMLComponent::InsertRealizationAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	ipRealizingClassifier	**in**	**IUMLClassifier** 1018			
	return	**return**	**IUMLComponentRealization** 1029			

Operation **IUMLComponent::IsCodeProjectFile**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IUMLComponent::IsIndirectlyInstantiated**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IUMLComponent::IsUseForCodeEngineering**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IUMLComponent::Realizations**

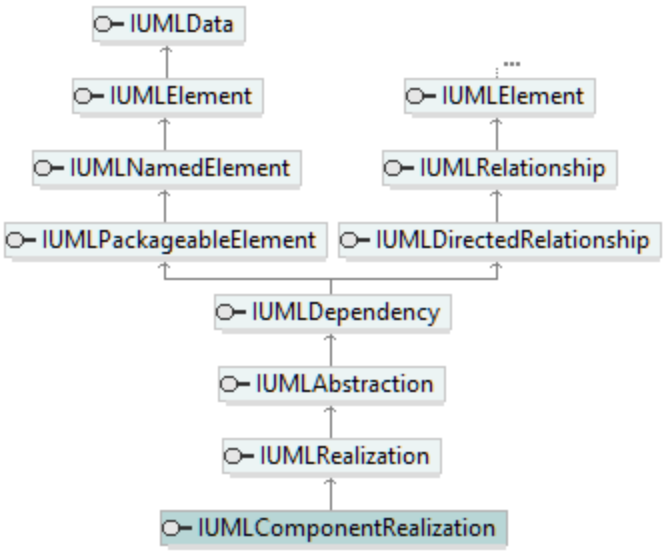
parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			
documentation	A list of elements of type IUMLComponentRealization 1029.					

UML documentation generated by **UModel** UML Editor **http://www.altova.com/umodel** Wed Sep 25 10:33:04 2024

16.4.3.5.37 UModelAPI - IUMLComponentRealization

Interface **IUMLComponentRealization**

diagram

hierarchy	
typedElements	Interface IUMLComponent 1028 Interface IUMLDataAll 911 Operation InsertRealizationAt 1029 Operation InsertRealizationAt 942

Operation **IUMLComponentRealization::Abstraction**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLComponent 1028			

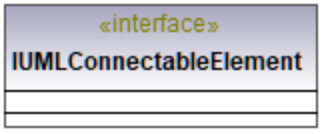
Operation **IUMLComponentRealization::RealizingClassifier**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLClassifier 1018			

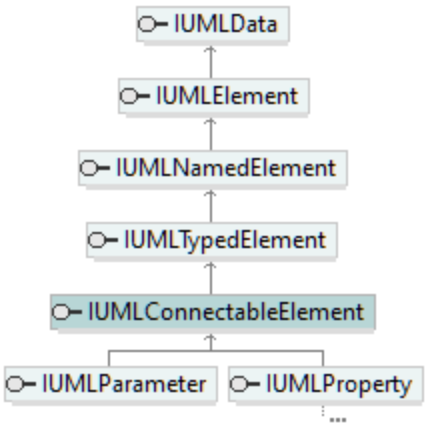
UML documentation generated by UModel UML Editor http://www.altova.com/umodel Wed Sep 25 10:33:04 2024

16.4.3.5.38 UModelAPI - IUMLConnectableElement

Interface **IUMLConnectableElement**

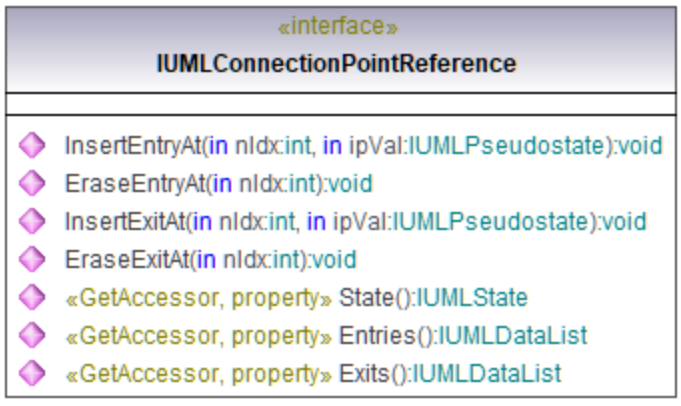
diagram	

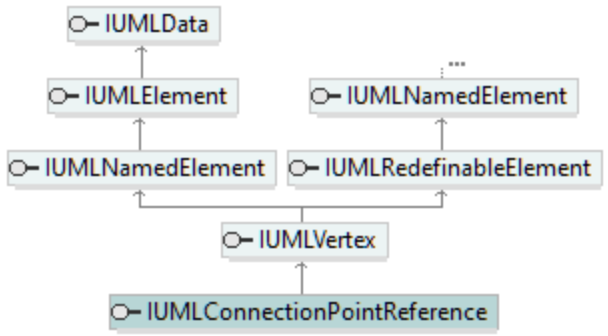
© 2019-2025 Altova GmbH

hierarchy	
typedElements	Interface IUMLCollaboration 1022 Interface IUMLConnectorEnd 1034 Interface IUMLDataAll 911 Interface IUMLLifeline 1104 Interface IUMLStructuredClassifier 1174 Operation InsertCollaborationRoleAt 1023 Operation Role 1034 Operation InsertCollaborationRoleAt 932 InsertOwnedConnectorAt 939 Represents 964 Role 965 Operation Represents 1104 Operation InsertOwnedConnectorAt 1175

16.4.3.5.39 UModelAPI - IUMLConnectionPointReference

Interface **IUMLConnectionPointReference**

diagram	

hierarchy	
typedElements	Interface **IUMLDataAll** 911 Interface **IUMLState** 1163 Operation **InsertConnectionAt** 933 Operation **InsertConnectionAt** 1165

Operation **IUMLConnectionPointReference::Entries**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			
documentation	A list of elements of type IUMLPseudostate 1153.					

Operation **IUMLConnectionPointReference::EraseEntryAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	**void**			

Operation **IUMLConnectionPointReference::EraseExitAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	**void**			

Operation **IUMLConnectionPointReference::Exits**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			
documentation	A list of elements of type IUMLPseudostate 1153.					

Operation **IUMLConnectionPointReference::InsertEntryAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	ipVal	**in**	**IUMLPseudostate** 1153			
	return	**return**	**void**			

Operation **IUMLConnectionPointReference::InsertExitAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			

© 2019-2025 Altova GmbH

	ipVal	**in**	IUMLPseudostate 1153
	return	**return**	**void**

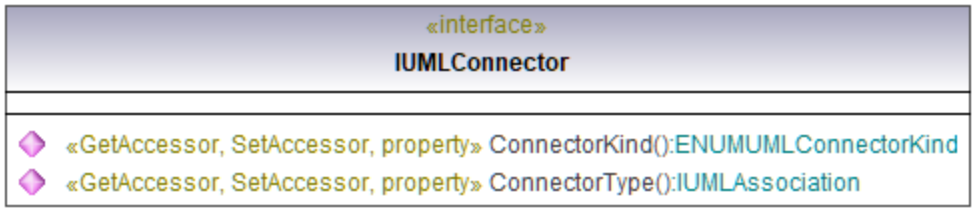
Operation **IUMLConnectionPointReference::State**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLState 1163			

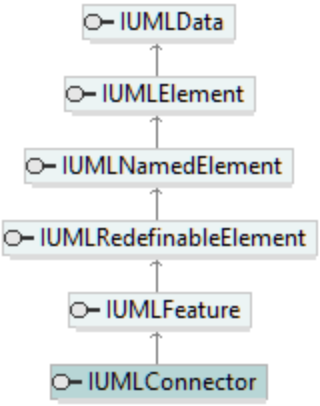
16.4.3.5.40 UModelAPI - IUMLConnector

Interface **IUMLConnector**

diagram



hierarchy



typedElements

Interface IUMLDataAll 911
Interface IUMLInformationFlow 1080
Interface IUMLStructuredClassifier 1174
Operation InsertOwnedConnectorAt 939
Operation InsertRealizingConnectorAt 943
Operation InsertRealizingConnectorAt 1082
Operation InsertOwnedConnectorAt 1175

Operation **IUMLConnector::ConnectorKind**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	ENUMUMLConnectorKind 1265			

documentation	**Deprecated:** Since UML2.3 (UModel2010r2) 'ConnectorKind' is derived and cannot be set anymore.

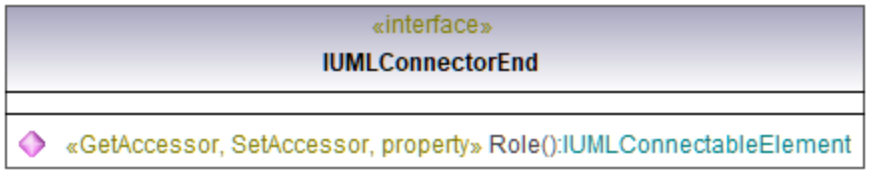
Operation **IUMLConnector::ConnectorType**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLAssociation 1000			

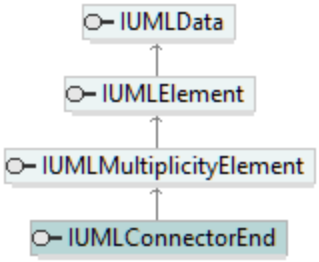
16.4.3.5.41 UModelAPI - IUMLConnectorEnd

Interface **IUMLConnectorEnd**

diagram



hierarchy



Operation **IUMLConnectorEnd::Role**

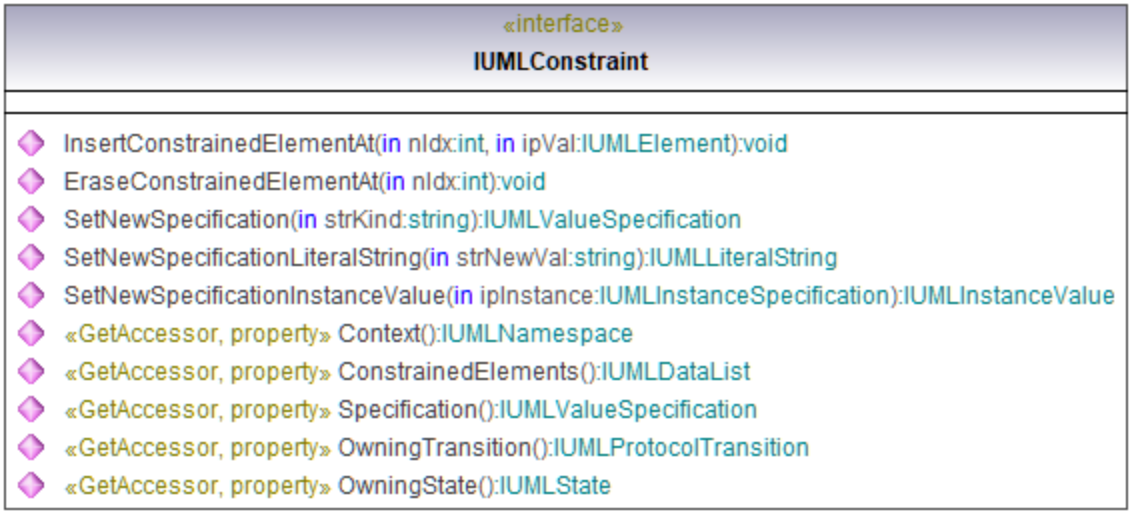
parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLConnectableElement 1030			

 © 2019-2025 Altova GmbH

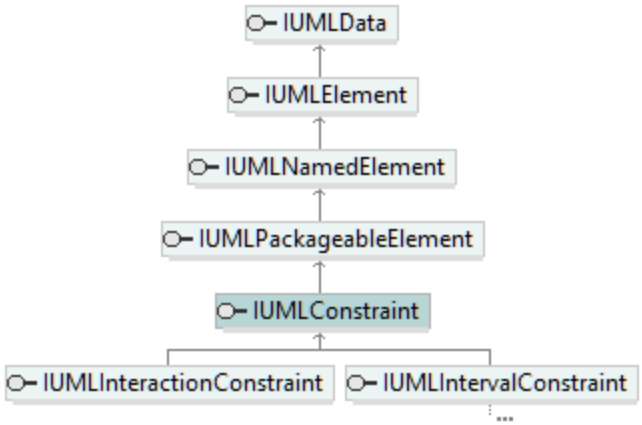
16.4.3.5.42 UModelAPI - IUMLConstraint

Interface **IUMLConstraint**

diagram



hierarchy



typed Elements

Interface **IUMLAction** 983

Interface **IUMLBehavior** 1002

Interface **IUMLDataAll** 911

Operation **InsertLocalPostConditionAt** 984
InsertLocalPreConditionAt 984

Operation **InsertPostconditionAt** 1003
InsertPreconditionAt 1003

Operation **InsertLocalPostConditionAt** 937
InsertLocalPreConditionAt 937
InsertOwnedRuleAt 941
InsertPostconditionAt 942
InsertPreconditionAt 942 **Invariant** 945
OwningConstraint 959
PostCondition 962 **PreCondition** 962
SetNewInvariant 968
SetNewPostCondition 969
SetNewPreCondition 969
SetNewStateInvariant 970
SetNewTransitionGuard 970

	Interface IUMLNamespace 1120 Interface IUMLProtocolTransition 1151 Interface IUMLState 1163 Interface IUMLStateInvariant 1166 Interface IUMLTransition 1187 Interface IUMLValueSpecification 1195 Operation StateInvariant 973 Operation TransitionGuard 977 Operation InsertOwnedRuleAt 1121 Operation PostCondition 1152 PreCondition 1152 Operation SetNewPostCondition 1152 Operation SetNewPreCondition 1152 Operation SetNewStateInvariant 1166 Operation StateInvariant 1166 Operation Invariant 1167 SetNewInvariant 1167 Operation SetNewTransitionGuard 1188 Operation TransitionGuard 1188 Operation OwningConstraint 1196

Operation **IUMLConstraint::ConstrainedElements**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLDataList 906			
documentation	A list of elements of type IUMLElement 1050.					

Operation **IUMLConstraint::Context**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLNamespace 1120			

Operation **IUMLConstraint::EraseConstrainedElementAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	**void**			

Operation **IUMLConstraint::InsertConstrainedElementAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	ipVal	**in**	IUMLElement 1050			
	return	**return**	**void**			

Operation **IUMLConstraint::OwningState**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLState 1163			

Operation **IUMLConstraint::OwningTransition**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLProtocolTransition 1151			

Operation **IUMLConstraint::SetNewSpecification**

parameter	name	direction	type	type modifier	multiplicity	default
	strKind	**in**	**string**			
	return	**return**	IUMLValueSpecification 1195			

© 2019-2025 Altova GmbH

Operation **IUMLConstraint::SetNewSpecificationInstanceValue**

parameter	name	direction	type	type modifier	multiplicity	default
	ipInstance	**in**	IUMLInstanceSpecification 1085			
	return	**return**	IUMLInstanceValue 1087			

Operation **IUMLConstraint::SetNewSpecificationLiteralString**

parameter	name	direction	type	type modifier	multiplicity	default
	strNewVal	**in**	**string**			
	return	**return**	IUMLLiteralString 1108			

Operation **IUMLConstraint::Specification**

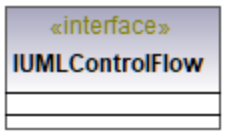
parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLValueSpecification 1195			

UML documentation generated by **UModel** UML Editor **http://www.altova.com/umodel** Wed Sep 25 10:33:04 2024

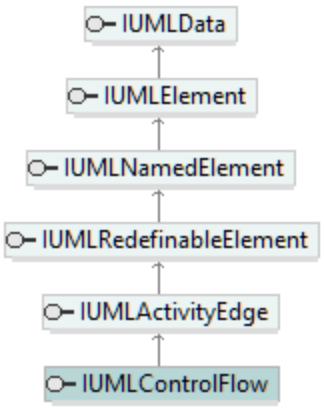
16.4.3.5.43 UModelAPI - IUMLControlFlow

Interface **IUMLControlFlow**

diagram



hierarchy

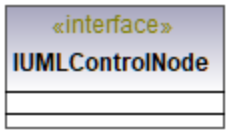


UML documentation generated by UModel UML Editor http://www.altova.com/umodel Wed Sep 25 10:33:04 2024

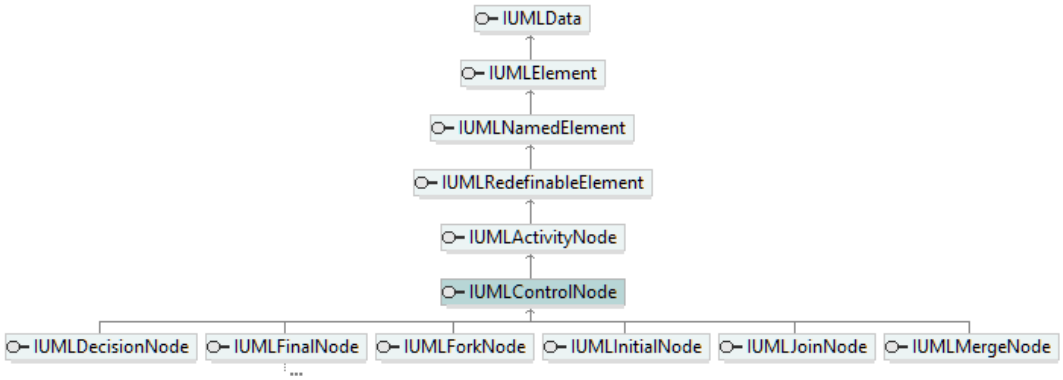
16.4.3.5.44 UModelAPI - IUMLControlNode

Interface **IUMLControlNode**

diagram



hierarchy

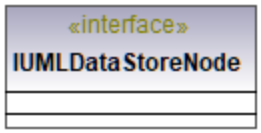


UML documentation generated by UModel UML Editor http://www.altova.com/umodel Wed Sep 25 10:33:04 2024

16.4.3.5.45 UModelAPI - IUMLDataStoreNode

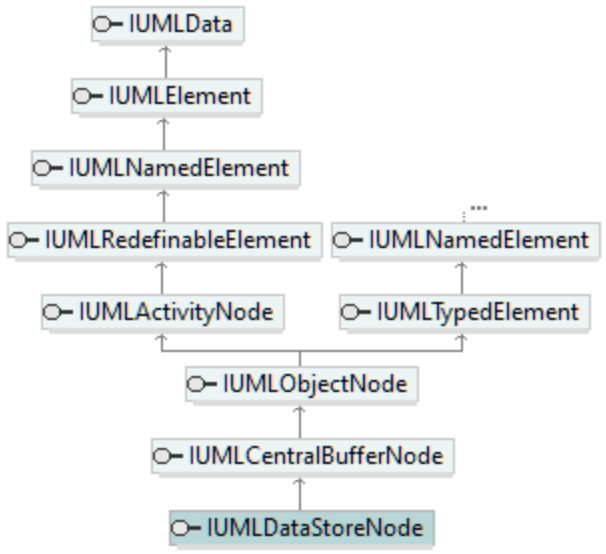
Interface **IUMLDataStoreNode**

diagram



 © 2019-2025 Altova GmbH

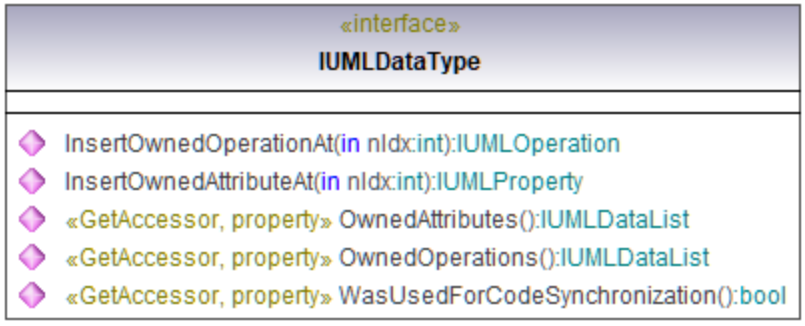
hierarchy

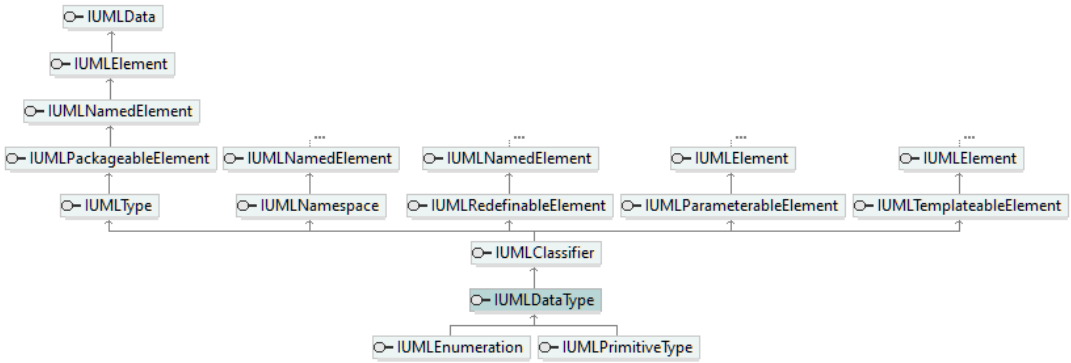


16.4.3.5.46 UModelAPI - IUMLDataType

Interface **IUMLDataType**

diagram



hierarchy	
typedElements	Interface **IUMLDataAll** 911 Interface **IUMLOperation** 1131 Interface **IUMLProperty** 1147 Operation **Datatype** 920 Operation **Datatype** 1132 Operation **Datatype** 1148

Operation **IUMLDataType::InsertOwnedAttributeAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	**IUMLProperty** 1147			

Operation **IUMLDataType::InsertOwnedOperationAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	**IUMLOperation** 1131			

Operation **IUMLDataType::OwnedAttributes**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			
documentation	A list of elements of type IUMLProperty 1147.					

Operation **IUMLDataType::OwnedOperations**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			
documentation	A list of elements of type IUMLOperation 1131.					

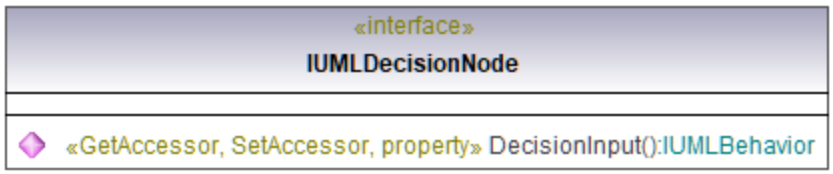
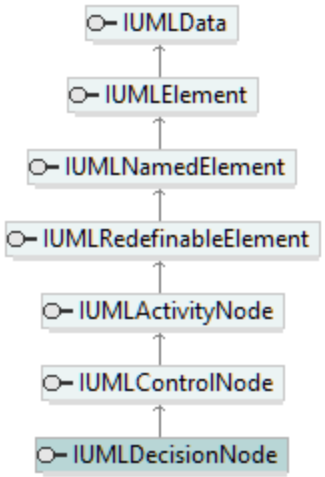
Operation **IUMLDataType::WasUsedForCodeSynchronization**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

© 2019-2025 Altova GmbH

16.4.3.5.47 UModelAPI - IUMLDecisionNode

Interface **IUMLDecisionNode**

diagram	
hierarchy	

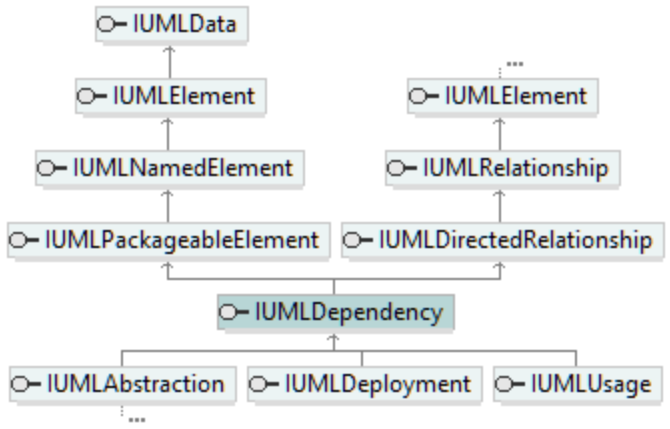
Operation **IUMLDecisionNode::DecisionInput**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLBehavior 1002			

16.4.3.5.48 UModelAPI - IUMLDependency

Interface **IUMLDependency**

diagram	

hierarchy	

Operation **IUMLDependency::Clients**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			
documentation	A list of elements of type IUMLNamedElement 1117.					

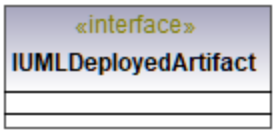
Operation **IUMLDependency::Suppliers**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			
documentation	A list of elements of type IUMLNamedElement 1117.					

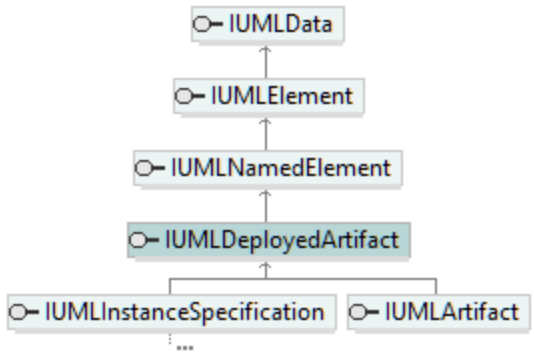
UML documentation generated by **UModel** UML Editor **http://www.altova.com/umodel** Wed Sep 25 10:33:04 2024

16.4.3.5.49 UModelAPI - IUMLDeployedArtifact

Interface **IUMLDeployedArtifact**

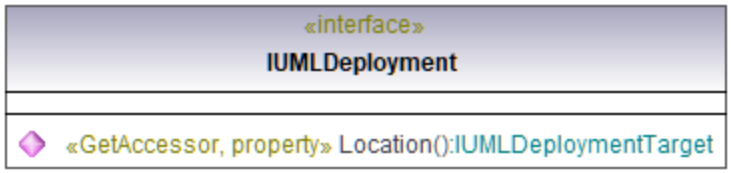
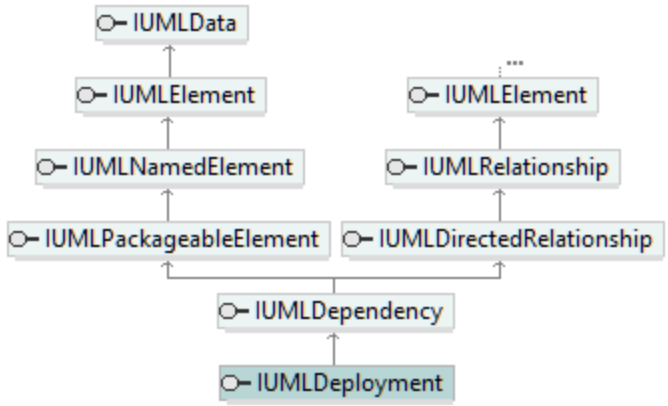
diagram	

© 2019-2025 Altova GmbH

hierarchy	
typedElements	Interface **IUMLDataAll** 911 Interface **IUMLDeploymentTarget** 1044 Operation **InsertDeploymentAt** 933 Operation **InsertDeploymentAt** 1045

16.4.3.5.50 UModelAPI - IUMLDeployment

Interface **IUMLDeployment**

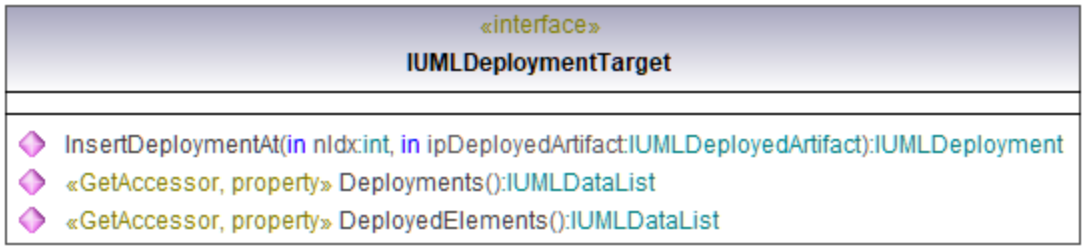
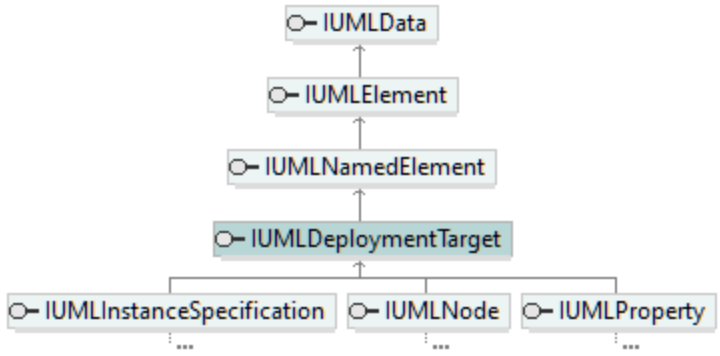
diagram	
hierarchy	
typedElements	Interface **IUMLDataAll** 911 Interface **IUMLDeploymentTarget** 1044 Operation **InsertDeploymentAt** 933 Operation **InsertDeploymentAt** 1045

Operation **IUMLDeployment::Location**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLDeployment Target 1044			

16.4.3.5.51 UModelAPI - IUMLDeploymentTarget

Interface **IUMLDeploymentTarget**

diagram	
hierarchy	
typedElements	Interface IUMLDataAll 911 Interface IUMLDeployment 1043 Operation Location 952 Operation Location 1044

Operation **IUMLDeploymentTarget::DeployedElements**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLDataList 906			
documentation	A list of elements of type IUMLPackageableElement 1137.					

Operation **IUMLDeploymentTarget::Deployments**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLDataList 906			

© 2019-2025 Altova GmbH

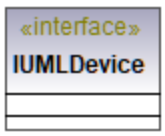
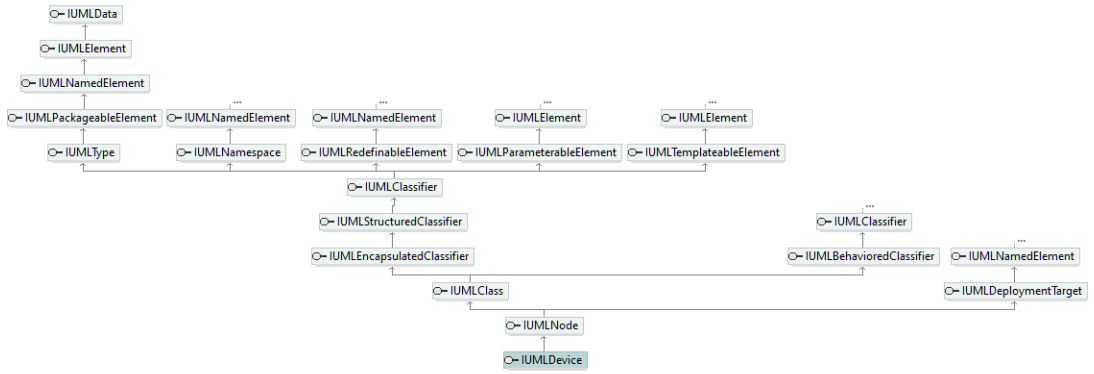
documentation	A list of elements of type IUMLDeployment[1043].

Operation **IUMLDeploymentTarget::InsertDeploymentAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	ipDeployedArtifact	**in**	**IUMLDeployedArtifact**[1042]			
	return	**return**	**IUMLDeployment**[1043]			

16.4.3.5.52 UModelAPI - IUMLDevice

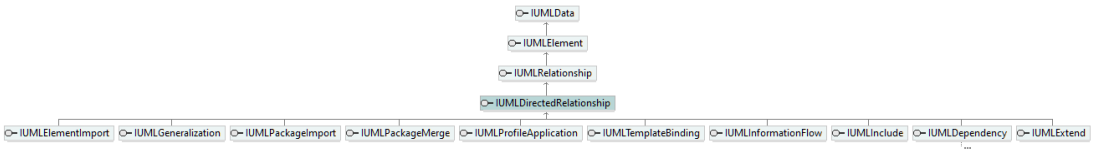
Interface **IUMLDevice**

diagram	
hierarchy	

© 2019-2025 Altova GmbH

16.4.3.5.53 UModelAPI - IUMLDirectedRelationship

Interface **IUMLDirectedRelationship**

diagram	
hierarchy	

Operation **IUMLDirectedRelationship::Sources**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			
documentation	A list of elements of type IUMLElement 1050.					

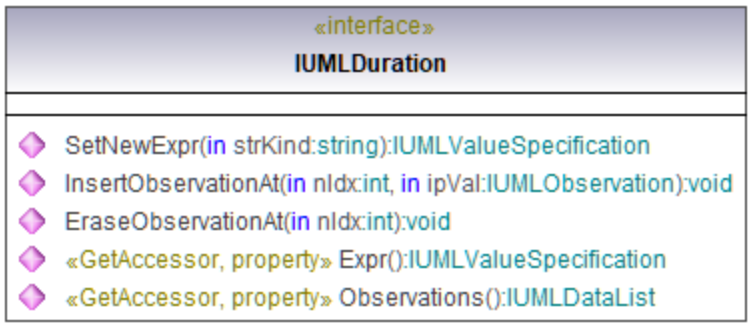
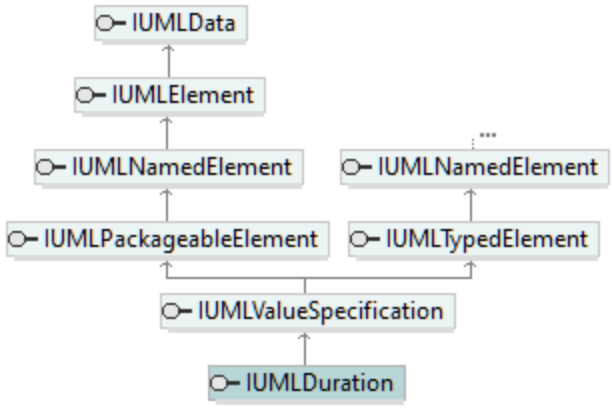
Operation **IUMLDirectedRelationship::Targets**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			
documentation	A list of elements of type IUMLElement 1050.					

© 2019-2025 Altova GmbH

16.4.3.5.54 UModelAPI - IUMLDuration

Interface **IUMLDuration**

diagram	
hierarchy	

Operation **IUMLDuration::EraseObservationAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	**void**			

Operation **IUMLDuration::Expr**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLValueSpecification** 1195			

Operation **IUMLDuration::InsertObservationAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	ipVal	**in**	**IUMLObservation** 1126			
	return	**return**	**void**			

Operation **IUMLDuration::Observations**

parameter	name	direction	type	type modifier	multiplicity	default

	return	**return**	**IUMLDataList** 906
documentation	A list of elements of type IUMLObservation 1126.		

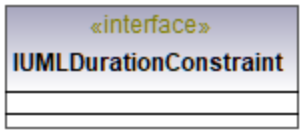
Operation **IUMLDuration::SetNewExpr**

parameter	name	direction	type	type modifier	multiplicity	default
	strKind	**in**	**string**			
	return	**return**	**IUMLValueSpecification** 1195			

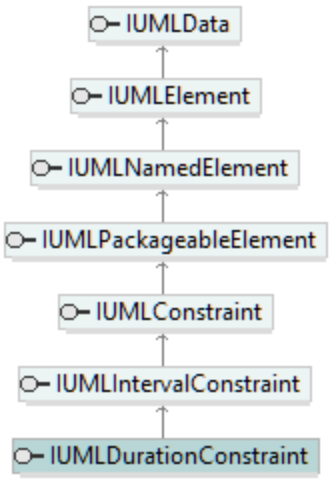
16.4.3.5.55 UModelAPI - IUMLDurationConstraint

Interface **IUMLDurationConstraint**

diagram



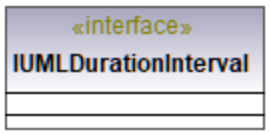
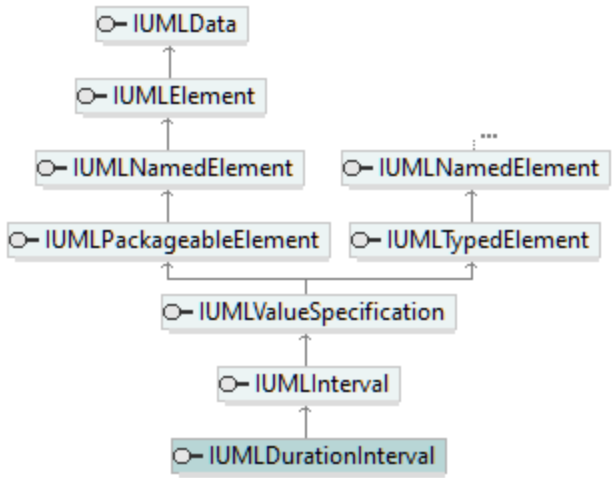
hierarchy



 © 2019-2025 Altova GmbH

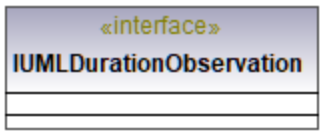
16.4.3.5.56 UModelAPI - IUMLDurationInterval

Interface **IUMLDurationInterval**

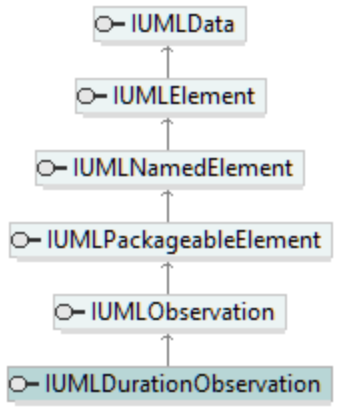
diagram	
hierarchy	

16.4.3.5.57 UModelAPI - IUMLDurationObservation

Interface **IUMLDurationObservation**

diagram	

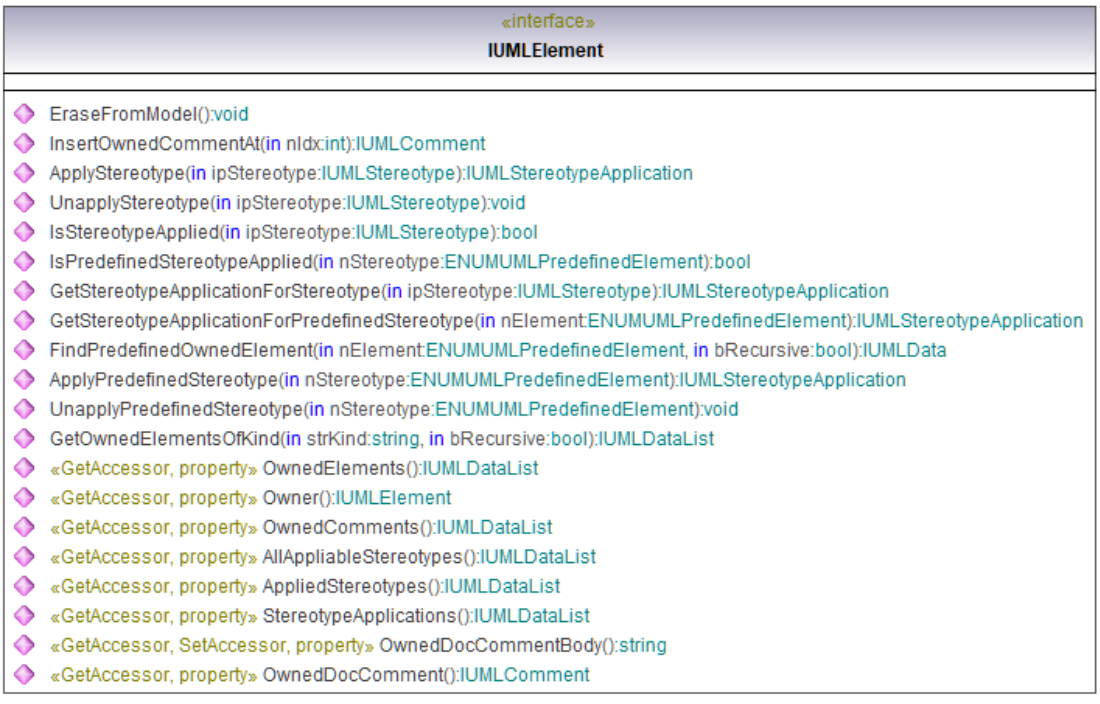
hierarchy



16.4.3.5.58 UModelAPI - IUMLElement

Interface **IUMLElement**

diagram



hierarchy

 © 2019-2025 Altova GmbH

typedElements	Interface **IGenerateSequenceDiagramDlg** 848	Operation **DiagramOwner** 848
	Interface **IUMLComment** 1025	Operation **InsertAnnotatedElementAt** 1026 **OwningElement** 1026
	Interface **IUMLConstraint** 1035	Operation **InsertConstrainedElementAt** 1036
	Interface **IUMLDataAll** 911	Operation **AppliedElement** 915 **Element** 921 **InsertAnnotatedElementAt** 932 **InsertConstrainedElementAt** 933 **InsertPackagedElementRelationshipAt** 941 **IsElementVisible** 947 **LinkedOwner** 952 **Owner** 959 **OwningElement** 959 **SetElementVisible** 966 **TextLabelElement** 976
	Interface **IUMLElement** 1050	Operation **Owner** 1053
	Interface **IUMLGuiDiagram** 1211	Operation **LinkedOwner** 1214
	Interface **IUMLGuiLink** 1224	Operation **Element** 1224
	Interface **IUMLGuiNodeLink** 1226	Operation **IsElementVisible** 1227 **SetElementVisible** 1227
	Interface **IUMLGuiTextLabel** 1251	Operation **TextLabelElement** 1252
	Interface **IUMLPackage** 1134	Operation **InsertPackagedElementRelationshipAt** 1135
	Interface **IUMLStereotypeApplication** 1170	Operation **AppliedElement** 1171

Operation **IUMLElement::AllAppliableStereotypes**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			
documentation	A list of elements of type IUMLStereotype 1169.					

Operation **IUMLElement::AppliedStereotypes**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			
documentation	A list of elements of type IUMLStereotype 1169.					

Operation **IUMLElement::ApplyPredefinedStereotype**

parameter	name	direction	type	type modifier	multiplicity	default
	nStereotype	**in**	**ENUMUMLPredefinedElement** 1271			
	return	**return**	**IUMLStereotypeApplication** 1170			

Operation **IUMLElement::ApplyStereotype**

parameter	name	direction	type	type modifier	multiplicity	default
	ipStereotype	**in**	**IUMLStereotype** 1169			
	return	**return**	**IUMLStereotypeApplication** 1170			

Operation **IUMLElement::EraseFromModel**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**void**			
documentation	Use this function to erase the element from the model and all diagrams. Use IUMLGuiDiagram[1211]::EraseFromDiagram[1213] to erase from diagram only.					

Operation **IUMLElement::FindPredefinedOwnedElement**

parameter	name	direction	type	type modifier	multiplicity	default
	nElement	**in**	**ENUMUMLPredefinedElement**[1271]			
	bRecursive	**in**	**bool**			
	return	**return**	**IUMLData**[904]			

Operation **IUMLElement::GetOwnedElementsOfKind**

parameter	name	direction	type	type modifier	multiplicity	default
	strKind	**in**	**string**			
	bRecursive	**in**	**bool**			
	return	**return**	**IUMLDataList**[906]			
documentation	get all owned elements of the specified kind (*strKind*)					

Operation **IUMLElement::GetStereotypeApplicationForPredefinedStereotype**

parameter	name	direction	type	type modifier	multiplicity	default
	nElement	**in**	**ENUMUMLPredefinedElement**[1271]			
	return	**return**	**IUMLStereotypeApplication**[1170]			

Operation **IUMLElement::GetStereotypeApplicationForStereotype**

parameter	name	direction	type	type modifier	multiplicity	default
	ipStereotype	**in**	**IUMLStereotype**[1169]			
	return	**return**	**IUMLStereotypeApplication**[1170]			

Operation **IUMLElement::InsertOwnedCommentAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	**IUMLComment**[1025]			

Operation **IUMLElement::IsPredefinedStereotypeApplied**

parameter	name	direction	type	type modifier	multiplicity	default
	nStereotype	**in**	**ENUMUMLPredefinedElement**[1271]			
	return	**return**	**bool**			

Operation **IUMLElement::IsStereotypeApplied**

parameter	name	direction	type	type modifier	multiplicity	default
	ipStereotype	**in**	**IUMLStereotype**[1169]			

© 2019-2025 Altova GmbH

	return	**return**	**bool**

Operation **IUMLElement::OwnedComments**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			
documentation	A list of elements of type IUMLComment 1025.					

Operation **IUMLElement::OwnedDocComment**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLComment** 1025			

Operation **IUMLElement::OwnedDocCommentBody**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**string**			

Operation **IUMLElement::OwnedElements**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			
documentation	A list of elements of type IUMLElement 1050.					

Operation **IUMLElement::Owner**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLElement** 1050			

Operation **IUMLElement::StereotypeApplications**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			
documentation	A list of elements of type IUMLStereotypeApplication 1170.					

Operation **IUMLElement::UnapplyPredefinedStereotype**

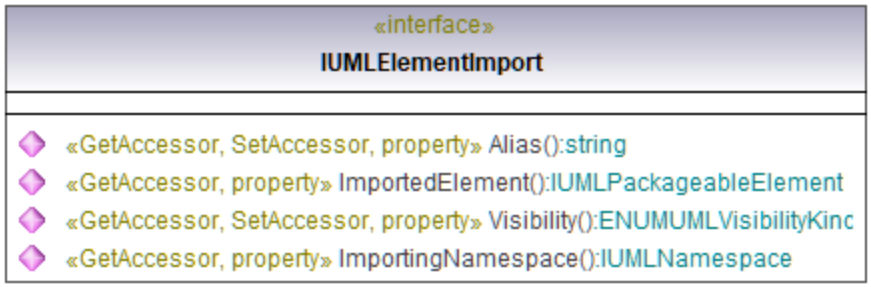
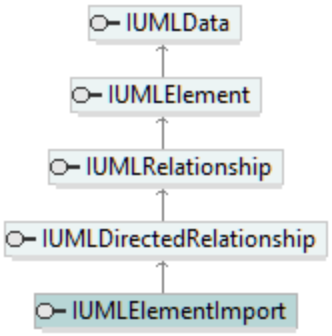
parameter	name	direction	type	type modifier	multiplicity	default
	nStereotype	**in**	**ENUMUMLPredefinedElement** 1271			
	return	**return**	**void**			

Operation **IUMLElement::UnapplyStereotype**

parameter	name	direction	type	type modifier	multiplicity	default
	ipStereotype	**in**	**IUMLStereotype** 1169			
	return	**return**	**void**			

16.4.3.5.59 UModelAPI - IUMLElementImport

Interface **IUMLElementImport**

diagram	
hierarchy	
typedElements	Interface IUMLDataAll 911 Interface IUMLNamespace 1120 Operation InsertElementImportAt 934 Operation InsertElementImportAt 1121

Operation **IUMLElementImport::Alias**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**string**			

Operation **IUMLElementImport::ImportedElement**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLPackageableElement 1137			

Operation **IUMLElementImport::ImportingNamespace**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLNamespace 1120			

Operation **IUMLElementImport::Visibility**

parameter	name	direction	type	type modifier	multiplicity	default

 © 2019-2025 Altova GmbH

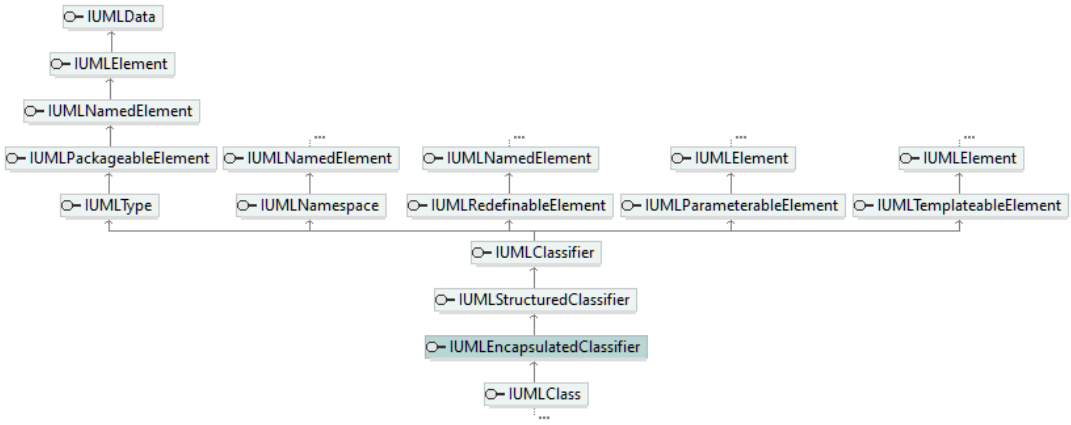
	return	**return**	**ENUMUMLVisibilityKind** 1273

16.4.3.5.60 UModelAPI - IUMLEncapsulatedClassifier

Interface **IUMLEncapsulatedClassifier**

diagram

hierarchy



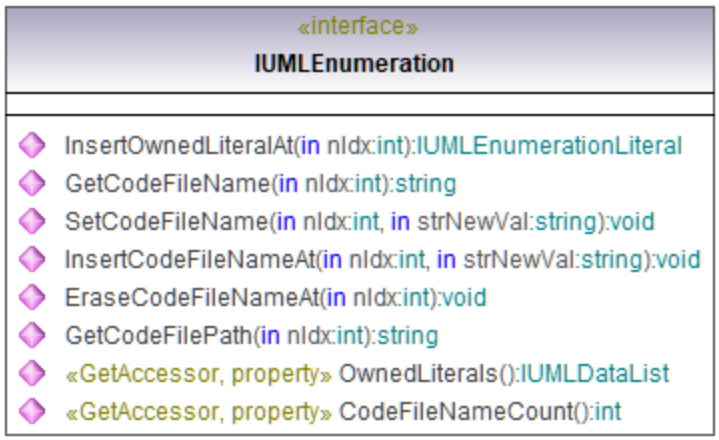
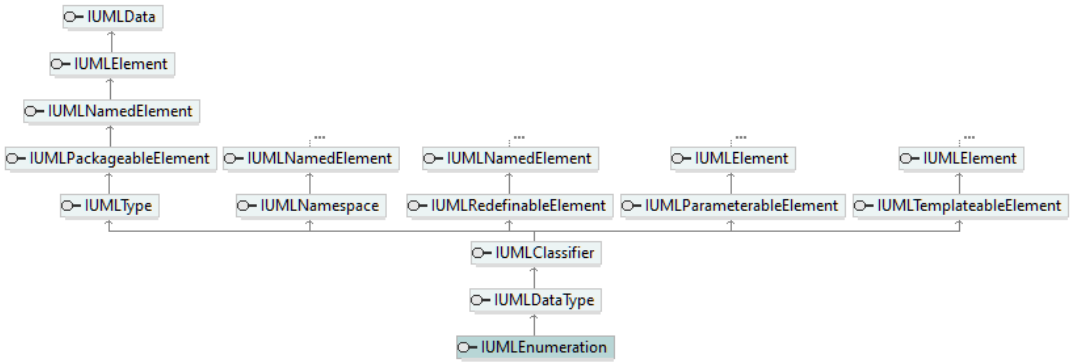
Operation **IUMLEncapsulatedClassifier::OwnedPorts**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			

© 2019-2025 Altova GmbH

16.4.3.5.61 UModelAPI - IUMLEnumeration

Interface **IUMLEnumeration**

diagram	
hierarchy	
typedElements	Interface **IUMLDataAll** 911 Interface **IUMLEnumerationLiteral** 1057 Operation **Enumeration** 922 Operation **Enumeration** 1058

Operation **IUMLEnumeration::CodeFileNameCount**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**int**			

Operation **IUMLEnumeration::EraseCodeFileNameAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	**void**			

Operation **IUMLEnumeration::GetCodeFileName**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	**string**			

© 2019-2025 Altova GmbH

Operation **IUMLEnumeration::GetCodeFilePath**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	**string**			
documentation	get the full code file path					

Operation **IUMLEnumeration::InsertCodeFileNameAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	strNewVal	**in**	**string**			
	return	**return**	**void**			

Operation **IUMLEnumeration::InsertOwnedLiteralAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	**IUMLEnumerationLiteral** 1057			

Operation **IUMLEnumeration::OwnedLiterals**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			
documentation	A list of elements of type IUMLEnumerationLiteral 1057.					

Operation **IUMLEnumeration::SetCodeFileName**

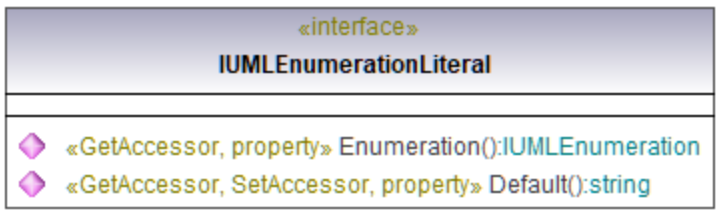
parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	strNewVal	**in**	**string**			
	return	**return**	**void**			

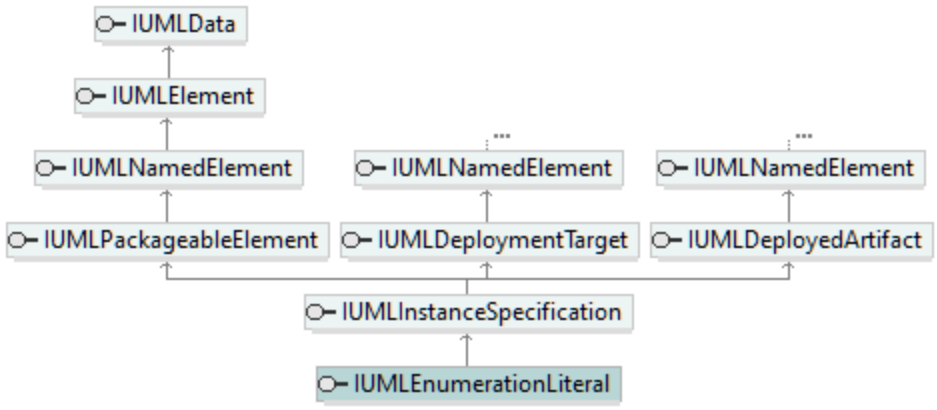
UML documentation generated by **UModel** UML Editor **http://www.altova.com/umodel** Wed Sep 25 10:33:04 2024

16.4.3.5.62 UModelAPI - IUMLEnumerationLiteral

Interface **IUMLEnumerationLiteral**

diagram



hierarchy	
typedElements	Interface IUMLDataAll 911 Interface IUMLEnumeration 1056 Operation InsertOwnedLiteralAt 940 Operation InsertOwnedLiteralAt 1057

Operation **IUMLEnumerationLiteral::Default**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**string**			

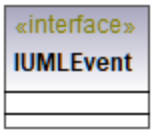
Operation **IUMLEnumerationLiteral::Enumeration**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLEnumeration 1056			

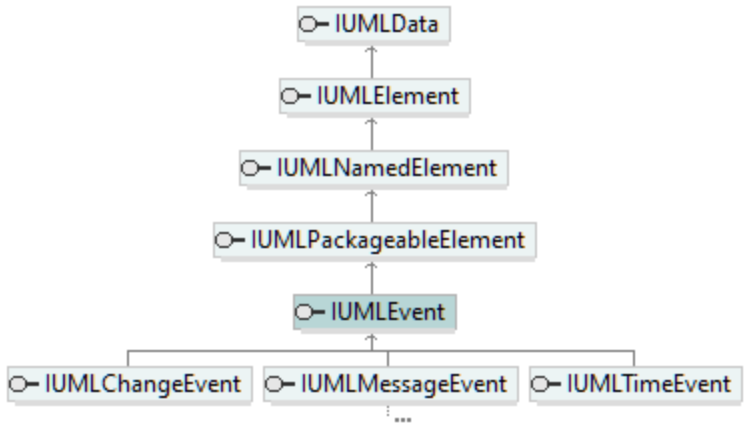
UML documentation generated by **UModel** UML Editor **http://www.altova.com/umodel** Wed Sep 25 10:33:04 2024

16.4.3.5.63 UModelAPI - IUMLEvent

Interface **IUMLEvent**

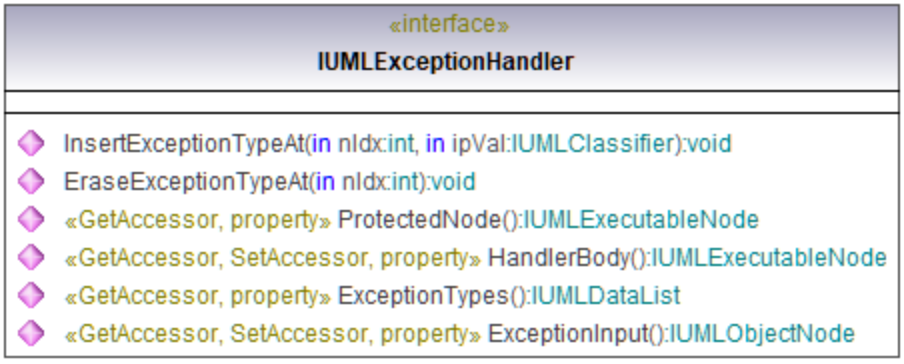
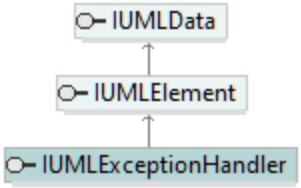
diagram	

© 2019-2025 Altova GmbH

hierarchy	
typedElements	Interface **IUMLDataAll** 911 Interface **IUMLOccurrenceSpecification** 1127 Interface **IUMLTrigger** 1188 Operation **Event** 924 **OccurringEvent** 955 Operation **OccurringEvent** 1127 Operation **Event** 1189

16.4.3.5.64 UModelAPI - IUMLExceptionHandler

Interface **IUMLExceptionHandler**

diagram	
hierarchy	

typedElements	Interface IUMLDataAll 911 Interface IUMLExecutableNode 1061	Operation InsertHandlerAt 935 Operation InsertHandlerAt 1061

Operation **IUMLExceptionHandler::EraseExceptionTypeAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	**void**			

Operation **IUMLExceptionHandler::ExceptionInput**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLObjectNode** 1124			

Operation **IUMLExceptionHandler::ExceptionTypes**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			
documentation	A list of elements of type IUMLClassifier 1018.					

Operation **IUMLExceptionHandler::HandlerBody**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLExecutableNode** 1061			

Operation **IUMLExceptionHandler::InsertExceptionTypeAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	ipVal	**in**	**IUMLClassifier** 1018			
	return	**return**	**void**			

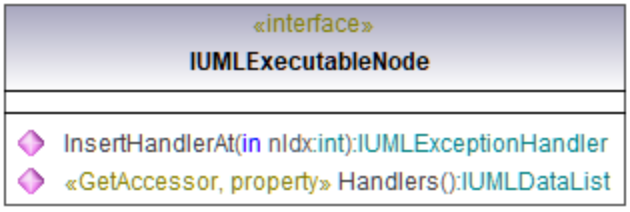
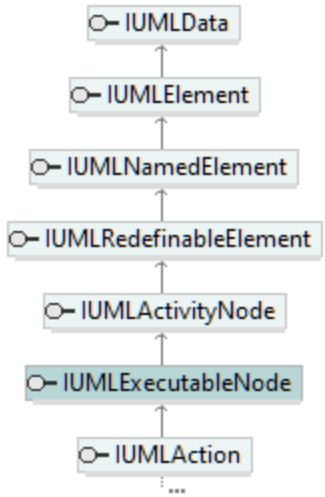
Operation **IUMLExceptionHandler::ProtectedNode**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLExecutableNode** 1061			

© 2019-2025 Altova GmbH

16.4.3.5.65 UModelAPI - IUMLExecutableNode

Interface **IUMLExecutableNode**

diagram	
hierarchy	
typedElements	Interface **IUMLDataAll** 911 Interface **IUMLExceptionHandler** 1059 Operation **HandlerBody** 929 **ProtectedNode** 962 Operation **HandlerBody** 1060 **ProtectedNode** 1060

Operation **IUMLExecutableNode::Handlers**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			
documentation	A list of elements of type IUMLExceptionHandler 1059.					

Operation **IUMLExecutableNode::InsertHandlerAt**

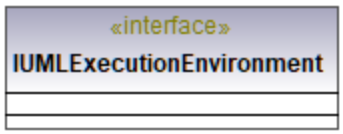
parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	**IUMLExceptionHandler** 1059			

© 2019-2025 Altova GmbH

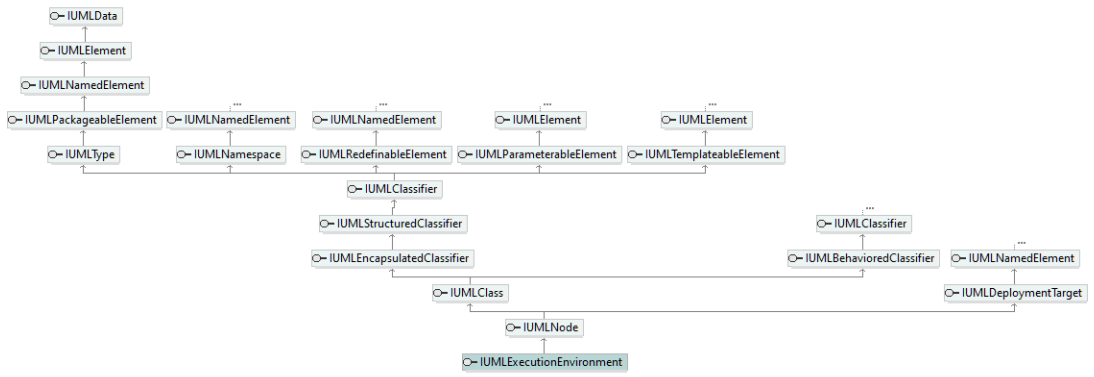
16.4.3.5.66 UModelAPI - IUMLExecutionEnvironment

Interface **IUMLExecutionEnvironment**

diagram



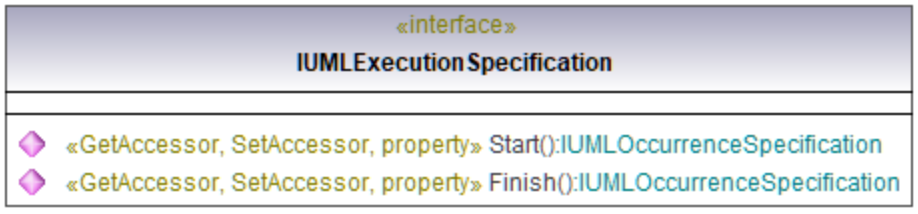
hierarchy



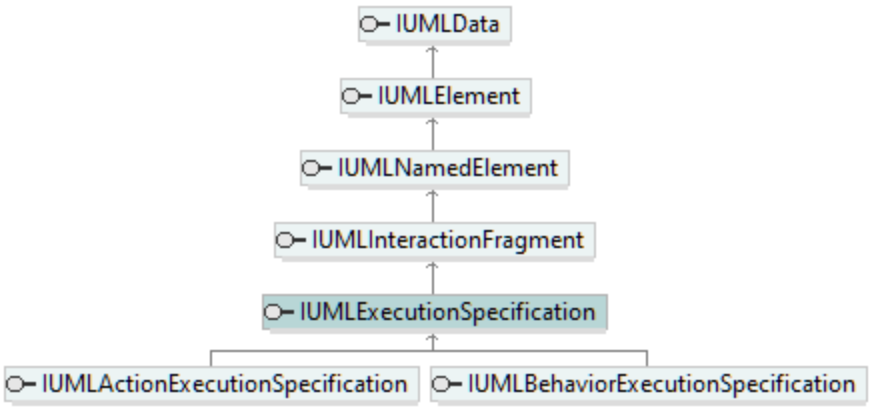
16.4.3.5.67 UModelAPI - IUMLExecutionSpecification

Interface **IUMLExecutionSpecification**

diagram



 © 2019-2025 Altova GmbH

hierarchy	
typedElements	Interface IUMLDataAll 911 Interface IUMLOccurrenceSpecification 1127 Operation ExecutionSpecificationFinish 925 ExecutionSpecificationStart 925 Operation ExecutionSpecificationFinish 1127 ExecutionSpecificationStart 1127

Operation **IUMLExecutionSpecification::Finish**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLOccurrenceSpecification 1127			

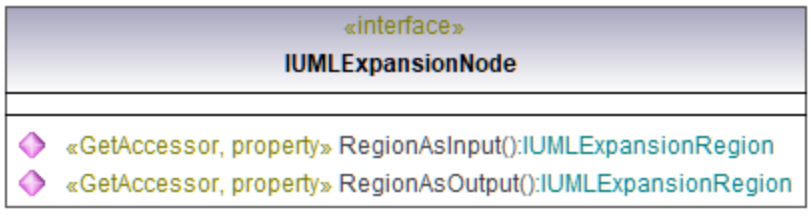
Operation **IUMLExecutionSpecification::Start**

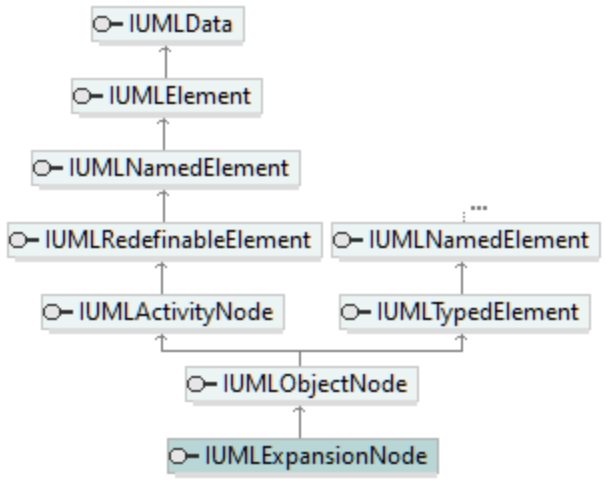
parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLOccurrenceSpecification 1127			

UML documentation generated by UModel UML Editor http://www.altova.com/umodel Wed Sep 25 10:33:04 2024

16.4.3.5.68 UModelAPI - IUMLExpansionNode

Interface **IUMLExpansionNode**

diagram	

hierarchy	
typedElements	Interface **IUMLDataAll** 911 Interface **IUMLExpansionRegion** 1065 Operation **InsertInputElementAt** 936 **InsertOutputElementAt** 938 Operation **InsertInputElementAt** 1066 **InsertOutputElementAt** 1066

Operation **IUMLExpansionNode::RegionAsInput**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLExpansionRegion** 1065			

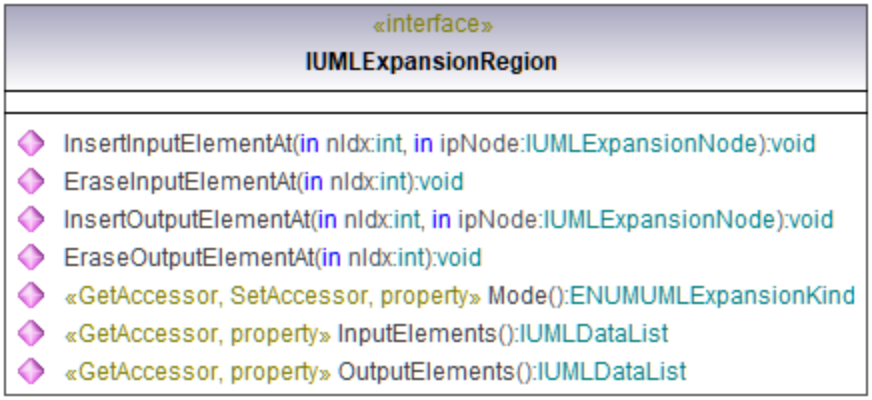
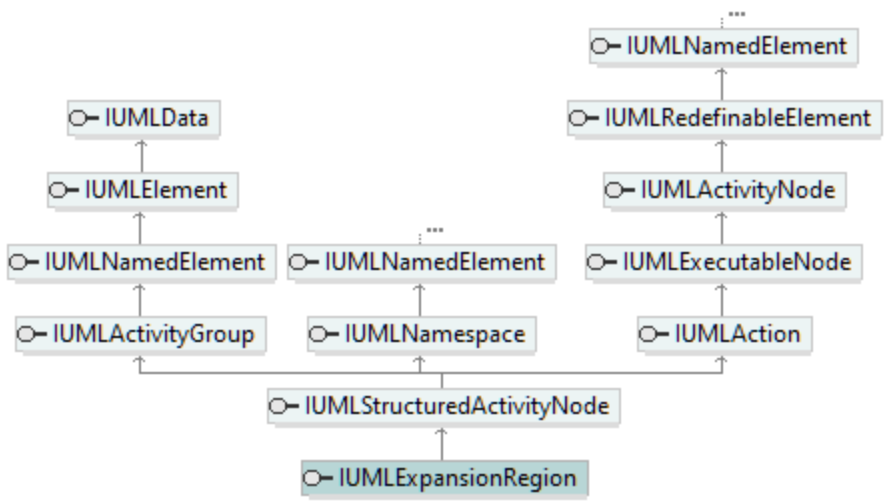
Operation **IUMLExpansionNode::RegionAsOutput**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLExpansionRegion** 1065			

© 2019-2025 Altova GmbH

16.4.3.5.69 UModelAPI - IUMLExpansionRegion

Interface **IUMLExpansionRegion**

diagram	
hierarchy	
typedElements	Interface **IUMLDataAll** 911 Interface **IUMLExpansionNode** 1063 Operation **RegionAsInput** 964 **RegionAsOutput** 964 Operation **RegionAsInput** 1064 **RegionAsOutput** 1064

Operation **IUMLExpansionRegion::EraseInputElementAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	**void**			

Operation **IUMLExpansionRegion::EraseOutputElementAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	**void**			

Operation **IUMLExpansionRegion::InputElements**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			
documentation	A list of elements of type IUMLExpansionNode 1063.					

Operation **IUMLExpansionRegion::InsertInputElementAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	ipNode	**in**	**IUMLExpansionNode** 1063			
	return	**return**	**void**			

Operation **IUMLExpansionRegion::InsertOutputElementAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	ipNode	**in**	**IUMLExpansionNode** 1063			
	return	**return**	**void**			

Operation **IUMLExpansionRegion::Mode**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**ENUMUMLExpansionKind** 1266			

Operation **IUMLExpansionRegion::OutputElements**

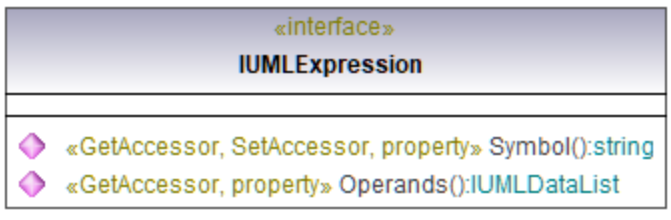
parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			
documentation	A list of elements of type IUMLExpansionNode 1063.					

UML documentation generated by **UModel** UML Editor **http://www.altova.com/umodel** Wed Sep 25 10:33:04 2024

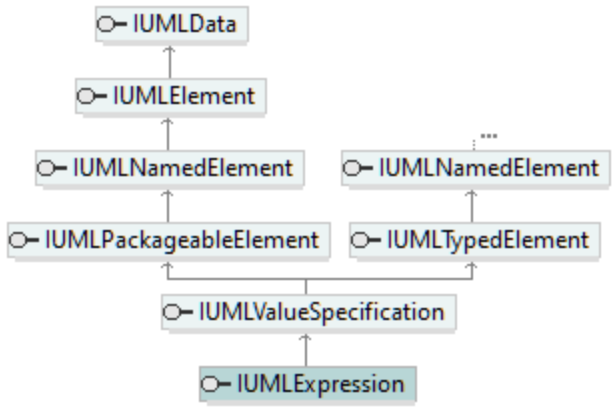
16.4.3.5.70 UModelAPI - IUMLExpression

Interface **IUMLExpression**

diagram



 © 2019-2025 Altova GmbH

hierarchy	
typedElements	Interface **IUMLDataAll** 911 Interface **IUMLValueSpecification** 1195 Operation **Expression** 925 Operation **Expression** 1196

Operation **IUMLExpression::Operands**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			
documentation	A list of elements of type IUMLValueSpecification 1195.					

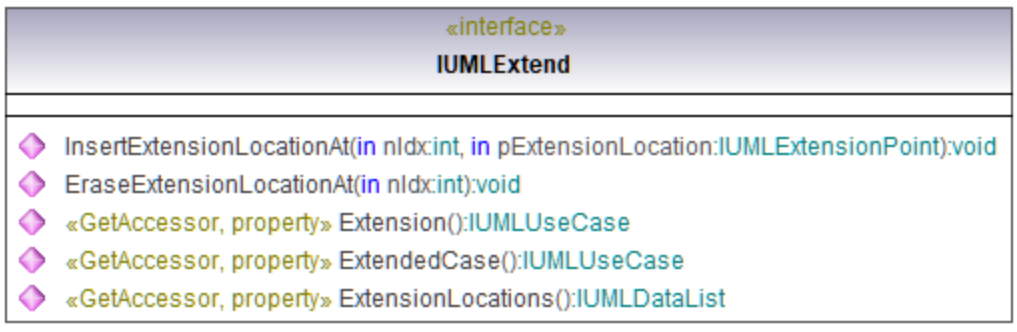
Operation **IUMLExpression::Symbol**

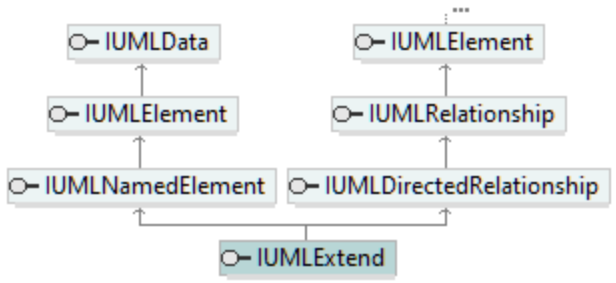
parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**string**			

UML documentation generated by **UModel** UML Editor **http://www.altova.com/umodel** Wed Sep 25 10:33:04 2024

16.4.3.5.71 UModelAPI - IUMLExtend

Interface **IUMLExtend**

diagram	

hierarchy	
typedElements	Interface **IUMLDataAll** 911 Interface **IUMLUseCase** 1192 Operation **InsertExtendAt** 934 Operation **InsertExtendAt** 1193

Operation **IUMLExtend::EraseExtensionLocationAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	**void**			

Operation **IUMLExtend::ExtendedCase**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLUseCase** 1192			

Operation **IUMLExtend::Extension**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLUseCase** 1192			

Operation **IUMLExtend::ExtensionLocations**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			
documentation	A list of elements of type IUMLExtensionPoint 1069.					

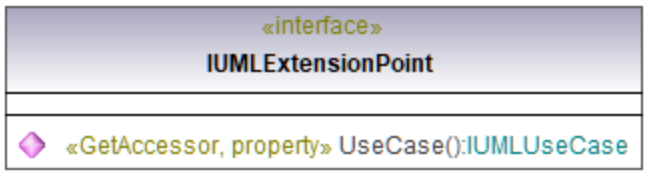
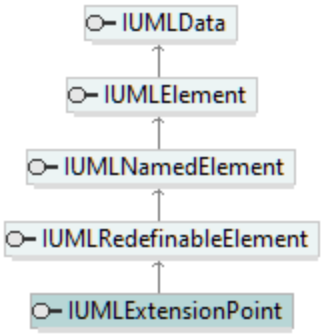
Operation **IUMLExtend::InsertExtensionLocationAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	pExtensionLocation	**in**	**IUMLExtensionPoint** 1069			
	return	**return**	**void**			

© 2019-2025 Altova GmbH

16.4.3.5.72 UModelAPI - IUMLExtensionPoint

Interface **IUMLExtensionPoint**

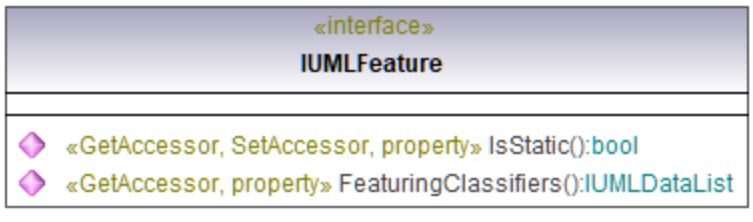
diagram	
hierarchy	
typedElements	Interface **IUMLDataAll** 911 Interface **IUMLExtend** 1067 Interface **IUMLUseCase** 1192 Operation **InsertExtensionLocationAt** 935 **InsertExtensionPointAt** 935 Operation **InsertExtensionLocationAt** 1068 Operation **InsertExtensionPointAt** 1193

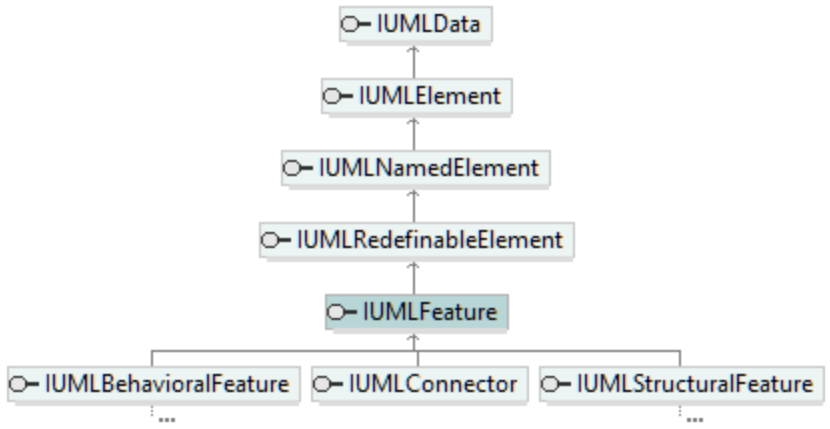
Operation **IUMLExtensionPoint::UseCase**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLUseCase** 1192			

16.4.3.5.73 UModelAPI - IUMLFeature

Interface **IUMLFeature**

diagram	

hierarchy	

Operation **IUMLFeature::FeaturingClassifiers**

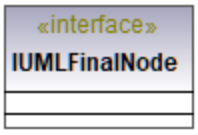
parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			
documentation	A list of elements of type IUMLClassifier 1018.					

Operation **IUMLFeature::IsStatic**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

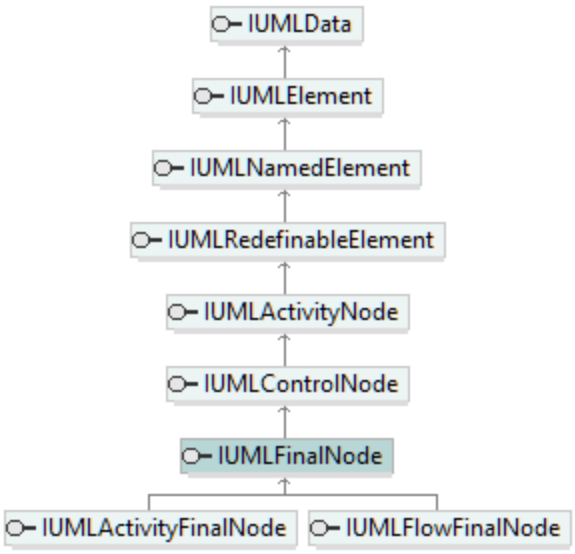
16.4.3.5.74 UModelAPI - IUMLFinalNode

Interface **IUMLFinalNode**

diagram	

© 2019-2025 Altova GmbH

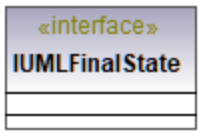
hierarchy



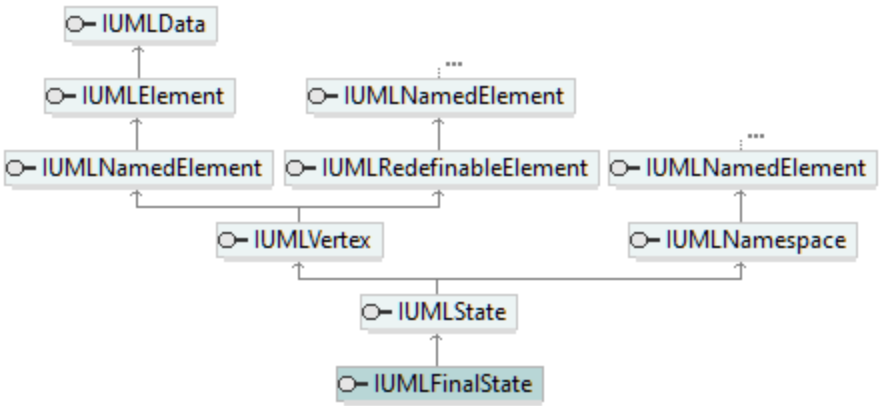
16.4.3.5.75 UModelAPI - IUMLFinalState

Interface **IUMLFinalState**

diagram



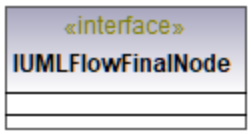
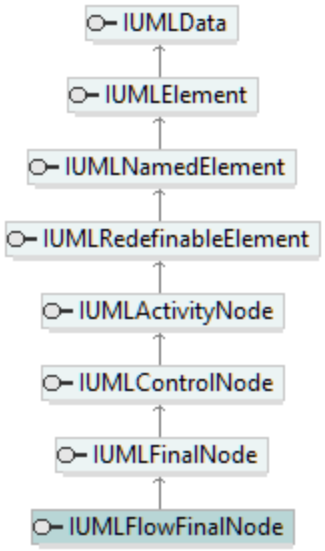
hierarchy



UML documentation generated by UModel UML Editor http://www.altova.com/umodel Wed Sep 25 10:33:04 2024

16.4.3.5.76 UModelAPI - IUMLFlowFinalNode

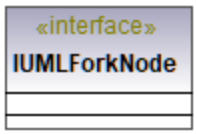
Interface **IUMLFlowFinalNode**

diagram	
hierarchy	

UML documentation generated by UModel UML Editor http://www.altova.com/umodel Wed Sep 25 10:33:04 2024

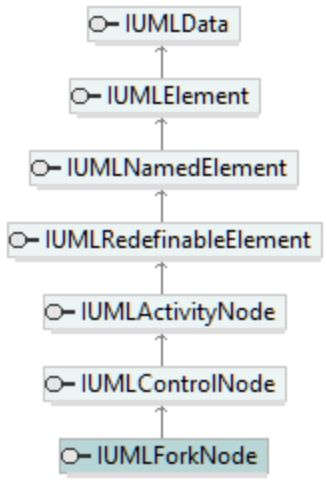
16.4.3.5.77 UModelAPI - IUMLForkNode

Interface **IUMLForkNode**

diagram	

 © 2019-2025 Altova GmbH

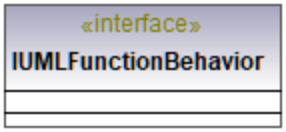
hierarchy



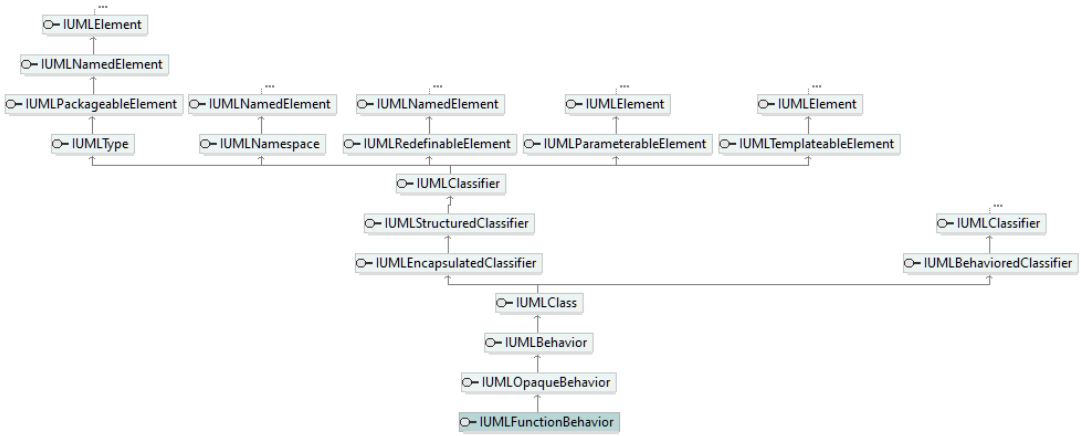
16.4.3.5.78 UModelAPI - IUMLFunctionBehavior

Interface **IUMLFunctionBehavior**

diagram



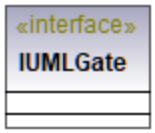
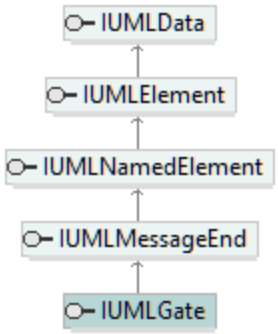
hierarchy



UML documentation generated by UModel UML Editor http://www.altova.com/umodel Wed Sep 25 10:33:04 2024

16.4.3.5.79 UModelAPI - IUMLGate

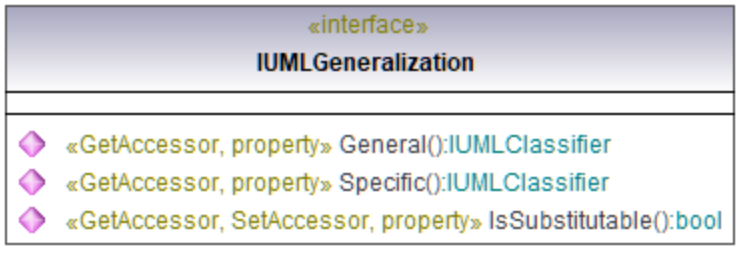
Interface **IUMLGate**

diagram	
hierarchy	
typedElements	Interface IUMLDataAll 911 Interface IUMLInteraction 1088 Interface IUMLInteractionUse 1092 Operation InsertActualGateAt 932 InsertFormalGateAt 935 Operation InsertFormalGateAt 1089 Operation InsertActualGateAt 1093

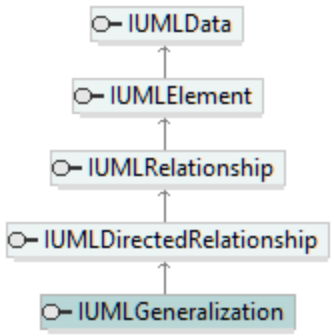
UML documentation generated by UModel UML Editor http://www.altova.com/umodel Wed Sep 25 10:33:04 2024

16.4.3.5.80 UModelAPI - IUMLGeneralization

Interface **IUMLGeneralization**

diagram	

 © 2019-2025 Altova GmbH

hierarchy	
typedElements	Interface **IUMLClassifier** 1018 Interface **IUMLDataAll** 911 Operation **InsertGeneralizationAt** 1020 Operation **InsertGeneralizationAt** 935

Operation **IUMLGeneralization::General**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLClassifier** 1018			

Operation **IUMLGeneralization::IsSubstitutable**

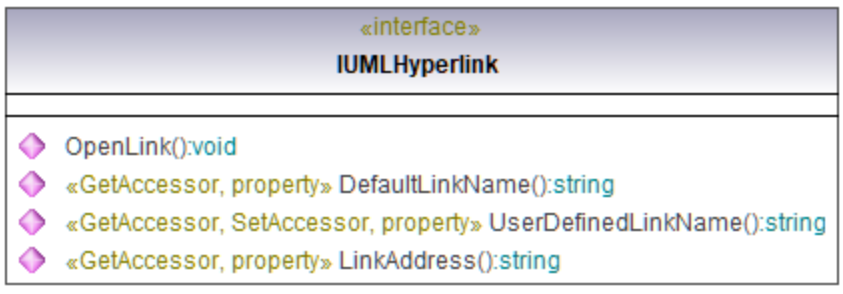
parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

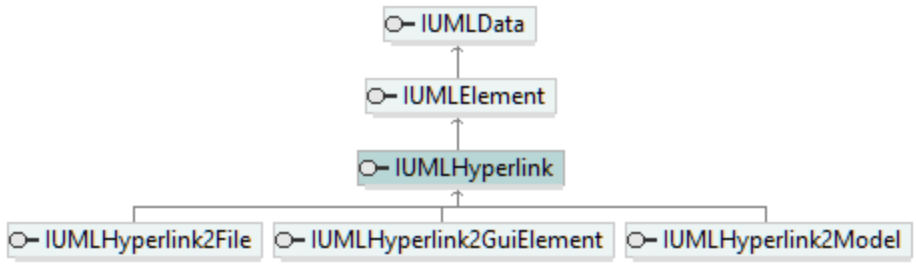
Operation **IUMLGeneralization::Specific**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLClassifier** 1018			

16.4.3.5.81 UModelAPI - IUMLHyperlink

Interface **IUMLHyperlink**

diagram	

hierarchy	

Operation **IUMLHyperlink::DefaultLinkName**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**string**			

Operation **IUMLHyperlink::LinkAddress**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**string**			

Operation **IUMLHyperlink::OpenLink**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**void**			

Operation **IUMLHyperlink::UserDefinedLinkName**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**string**			

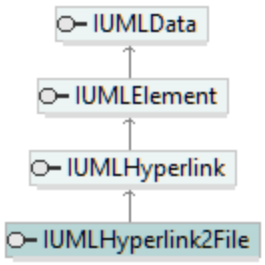
UML documentation generated by **UModel** UML Editor **http://www.altova.com/umodel** Wed Sep 25 10:33:04 2024

16.4.3.5.82 UModelAPI - IUMLHyperlink2File

Interface **IUMLHyperlink2File**

diagram	

 © 2019-2025 Altova GmbH

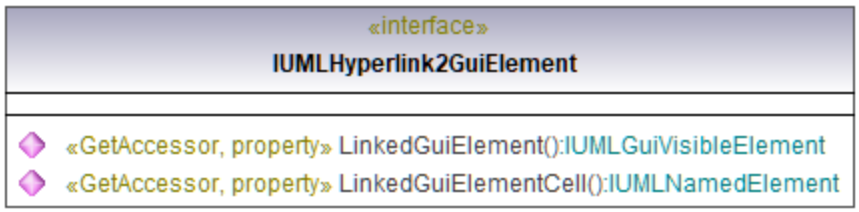
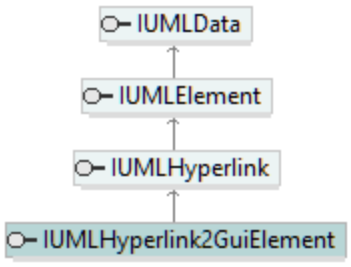
hierarchy	
typedElements	Interface **IUMLDataAll** 911 Interface **IUMLNamedElement** 1117 Operation **InsertOwnedHyperlink2FileAt** 939 Operation **InsertOwnedHyperlink2FileAt** 1118

Operation **IUMLHyperlink2File::SetHyperlinkFileAddress**

parameter	name	direction	type	type modifier	multiplicity	default
	strNewVal	**in**	**string**			
	return	**return**	**void**			

16.4.3.5.83 UModelAPI - IUMLHyperlink2GuiElement

Interface **IUMLHyperlink2GuiElement**

diagram	
hierarchy	
typedElements	Interface **IUMLDataAll** 911 Interface **IUMLNamedElement** 1117 Operation **InsertOwnedHyperlink2GuiElementAt** 940 Operation **InsertOwnedHyperlink2GuiElementAt** 1118

Operation **IUMLHyperlink2GuiElement::LinkedGuiElement**

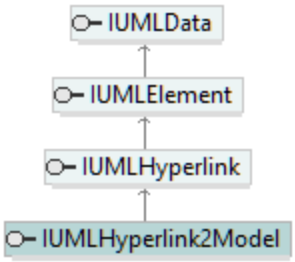
parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLGuiVisibleElement 1260			

Operation **IUMLHyperlink2GuiElement::LinkedGuiElementCell**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLNamedElement 1117			

16.4.3.5.84 UModelAPI - IUMLHyperlink2Model

Interface **IUMLHyperlink2Model**

diagram	
hierarchy	
typedElements	Interface IUMLDataAll 911 Interface IUMLNamedElement 1117 Operation InsertOwnedHyperlink2ModelAt 940 Operation InsertOwnedHyperlink2ModelAt 1118

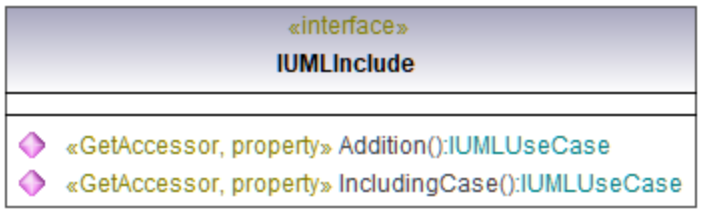
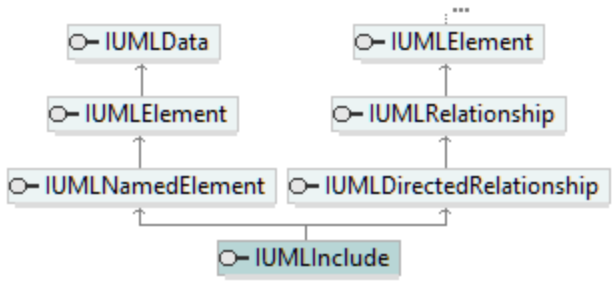
Operation **IUMLHyperlink2Model::LinkedModelElement**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLData 904			

 © 2019-2025 Altova GmbH

16.4.3.5.85 UModelAPI - IUMLInclude

Interface **IUMLInclude**

diagram	
hierarchy	
typedElements	Interface **IUMLDataAll** 911 Interface **IUMLUseCase** 1192 Operation **InsertIncludeAt** 935 Operation **InsertIncludeAt** 1193

Operation **IUMLInclude::Addition**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLUseCase** 1192			

Operation **IUMLInclude::IncludingCase**

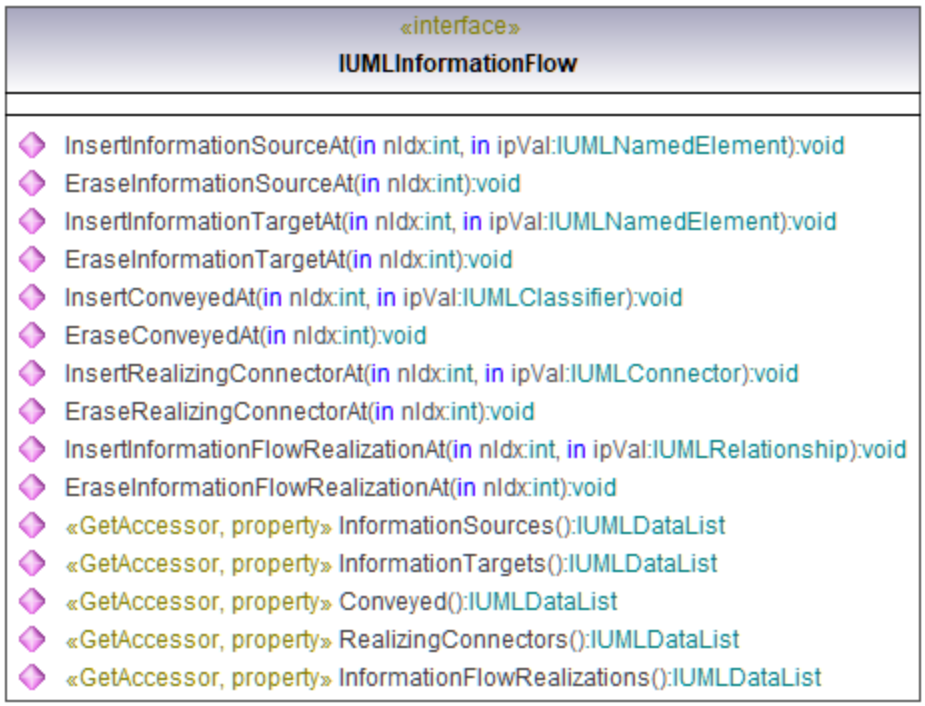
parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLUseCase** 1192			

© 2019-2025 Altova GmbH

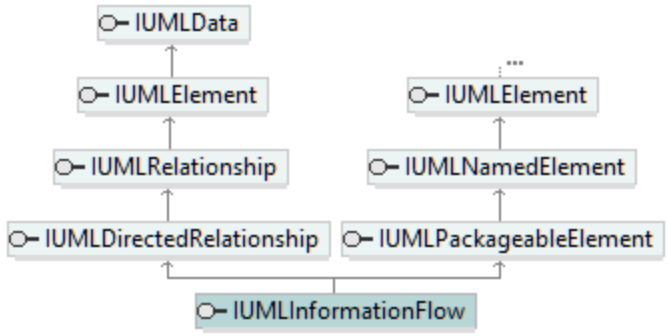
16.4.3.5.86 UModelAPI - IUMLInformationFlow

Interface **IUMLInformationFlow**

diagram



hierarchy



Operation **IUMLInformationFlow::Conveyed**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			

Operation **IUMLInformationFlow::EraseConveyedAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	**void**			

Operation **IUMLInformationFlow::EraseInformationFlowRealizationAt**

© 2019-2025 Altova GmbH

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	**void**			

Operation **IUMLInformationFlow::EraseInformationSourceAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	**void**			

Operation **IUMLInformationFlow::EraseInformationTargetAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	**void**			

Operation **IUMLInformationFlow::EraseRealizingConnectorAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	**void**			

Operation **IUMLInformationFlow::InformationFlowRealizations**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			

Operation **IUMLInformationFlow::InformationSources**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			

Operation **IUMLInformationFlow::InformationTargets**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			

Operation **IUMLInformationFlow::InsertConveyedAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	ipVal	**in**	**IUMLClassifier** 1018			
	return	**return**	**void**			

Operation **IUMLInformationFlow::InsertInformationFlowRealizationAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	ipVal	**in**	**IUMLRelationship** 1158			
	return	**return**	**void**			

Operation **IUMLInformationFlow::InsertInformationSourceAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			

	ipVal	**in**	**IUMLNamedElement** 1117
	return	**return**	**void**

Operation **IUMLInformationFlow::InsertInformationTargetAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	ipVal	**in**	**IUMLNamedElement** 1117			
	return	**return**	**void**			

Operation **IUMLInformationFlow::InsertRealizingConnectorAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	ipVal	**in**	**IUMLConnector** 1033			
	return	**return**	**void**			

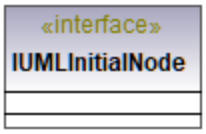
Operation **IUMLInformationFlow::RealizingConnectors**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			

UML documentation generated by **UModel** UML Editor **http://www.altova.com/umodel** Wed Sep 25 10:33:04 2024

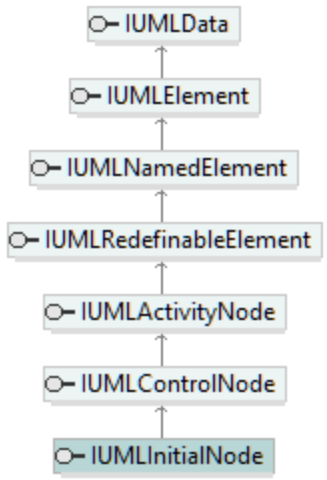
16.4.3.5.87 UModelAPI - IUMLInitialNode

Interface **IUMLInitialNode**

diagram	
	

 © 2019-2025 Altova GmbH

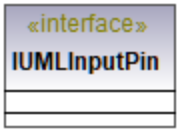
hierarchy



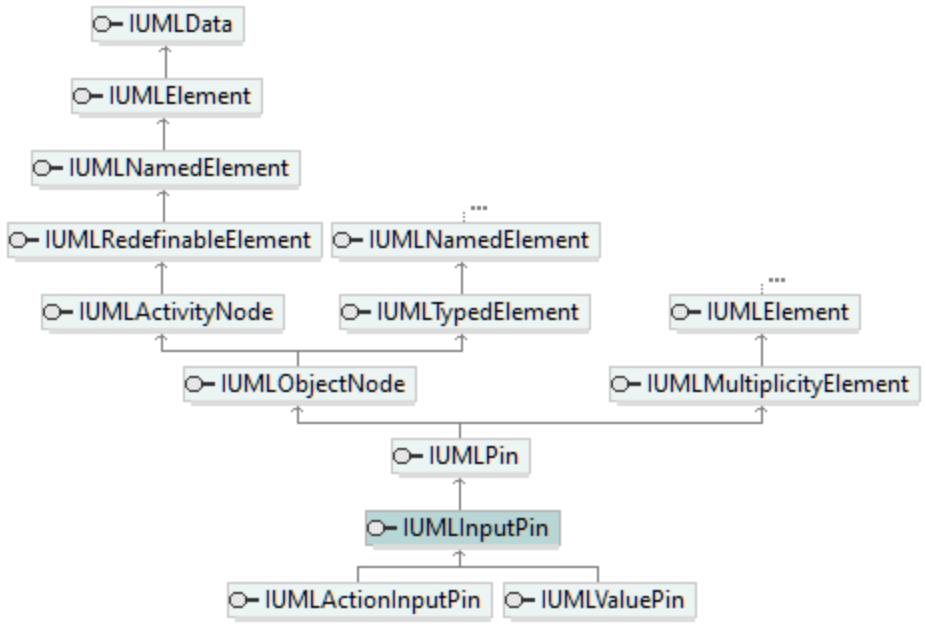
16.4.3.5.88 UModelAPI - IUMLInputPin

Interface **IUMLInputPin**

diagram



hierarchy



typedElements

Interface **IUMLCallOperationAction** 1011
Operation **CallTarget** 1012 **SetNewCallTarget** 1012

Interface **IUMLDataAll** 911
Operation **CallTarget** 917 **InsertArgumentAt** 932 **InsertArgumentOfKindAt** 932 **InsertInputValueAt** 936 **SetNewCallTarget** 966 **SetNewSignalTarget** 969 **SignalTarget** 972

Interface **IUMLInvocationAction** 1101
Operation **InsertArgumentAt** 1102 **InsertArgumentOfKindAt** 1102

Interface **IUMLOpaqueAction** 1128
Operation **InsertInputValueAt** 1129

Interface **IUMLSendSignalAction** 1159
Operation **SetNewSignalTarget** 1160 **SignalTarget** 1160

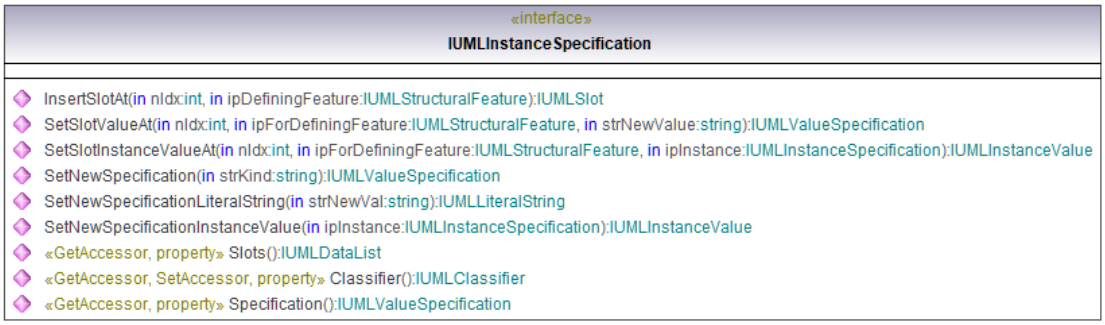
UML documentation generated by **UModel** UML Editor **http://www.altova.com/umodel** Wed Sep 25 10:33:04 2024

© 2019-2025 Altova GmbH

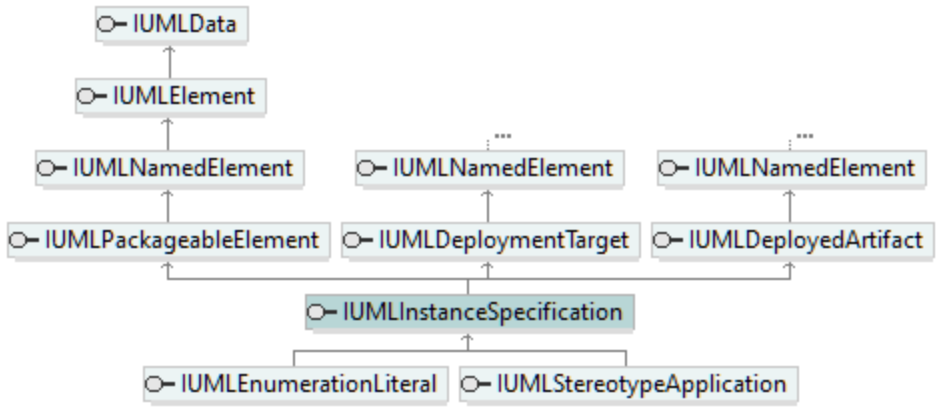
16.4.3.5.89 UModelAPI - IUMLInstanceSpecification

Interface **IUMLInstanceSpecification**

diagram



hierarchy



typedElements

Interface **IUMLConstraint** 1035
Interface **IUMLDataAll** 911
Interface **IUMLInstanceSpecification** 1085
Interface **IUMLInstanceValue** 1087
Interface **IUMLParameter** 1140
Interface **IUMLProperty** 1147
Interface **IUMLSlot** 1162
Interface **IUMLValueSpecification** 1195

Operation **SetNewSpecificationInstanceValue** 1037
Operation **InsertSlotInstanceValueAt** 943 **Instance** 944 **OwningInstance** 960 **OwningInstanceSpec** 960 **SetNewDefaultValueInstanceValue** 967 **SetNewSpecificationInstanceValue** 969 **SetSlotInstanceValueAt** 971
Operation **SetNewSpecificationInstanceValue** 1086 **SetSlotInstanceValueAt** 1086
Operation **Instance** 1088
Operation **SetNewDefaultValueInstanceValue** 1141
Operation **SetNewDefaultValueInstanceValue** 1150
Operation **InsertSlotInstanceValueAt** 1162 **OwningInstance** 1163
Operation **OwningInstanceSpec** 1196

Operation **IUMLInstanceSpecification::Classifier**

parameter	name	direction	type	type modifier	multiplicity	default

	return	**return**	**IUMLClassifier** 1018

Operation **IUMLInstanceSpecification::InsertSlotAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	ipDefiningFeature	**in**	**IUMLStructuralFeature** 1171			
	return	**return**	**IUMLSlot** 1162			

Operation **IUMLInstanceSpecification::SetNewSpecification**

parameter	name	direction	type	type modifier	multiplicity	default
	strKind	**in**	**string**			
	return	**return**	**IUMLValueSpecification** 1195			

Operation **IUMLInstanceSpecification::SetNewSpecificationInstanceValue**

parameter	name	direction	type	type modifier	multiplicity	default
	ipInstance	**in**	**IUMLInstanceSpecification** 1085			
	return	**return**	**IUMLInstanceValue** 1087			

Operation **IUMLInstanceSpecification::SetNewSpecificationLiteralString**

parameter	name	direction	type	type modifier	multiplicity	default
	strNewVal	**in**	**string**			
	return	**return**	**IUMLLiteralString** 1108			

Operation **IUMLInstanceSpecification::SetSlotInstanceValueAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	ipForDefiningFeature	**in**	**IUMLStructuralFeature** 1171			
	ipInstance	**in**	**IUMLInstanceSpecification** 1085			
	return	**return**	**IUMLInstanceValue** 1087			

Operation **IUMLInstanceSpecification::SetSlotValueAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	ipForDefiningFeature	**in**	**IUMLStructuralFeature** 1171			
	strNewValue	**in**	**string**			
	return	**return**	**IUMLValueSpecification** 1195			

Operation **IUMLInstanceSpecification::Slots**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			

© 2019-2025 Altova GmbH

documentation	A list of elements of type IUMLSlot[1162].

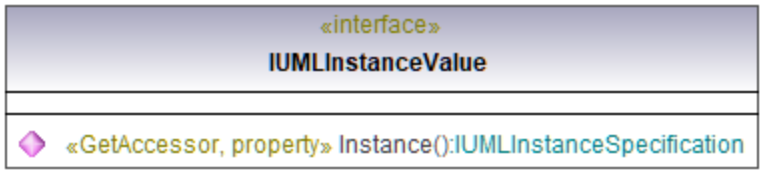
Operation **IUMLInstanceSpecification::Specification**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLValueSpecification**[1195]			

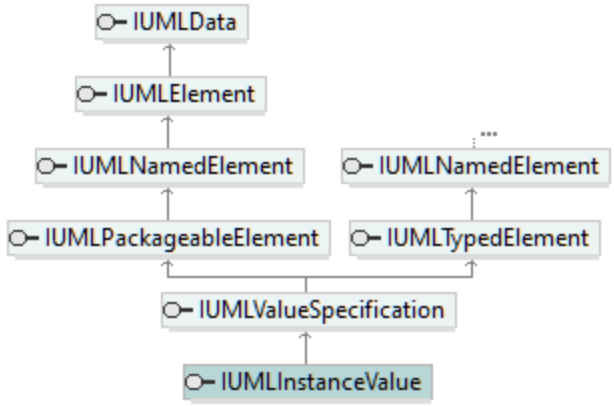
16.4.3.5.90 UModelAPI - IUMLInstanceValue

Interface **IUMLInstanceValue**

diagram



hierarchy



typedElements

Interface **IUMLConstraint**[1035]
Interface **IUMLDataAll**[911]
Interface **IUMLInstanceSpecification**[1085]
Interface **IUMLParameter**[1140]
Interface **IUMLProperty**[1147]
Interface **IUMLSlot**[1162]

Operation **SetNewSpecificationInstanceValue**[1037]
Operation **InsertSlotInstanceValueAt**[943]
SetNewDefaultValueInstanceValue[967]
SetNewSpecificationInstanceValue[969]
SetSlotInstanceValueAt[971]
Operation **SetNewSpecificationInstanceValue**[1086]
SetSlotInstanceValueAt[1086]
Operation **SetNewDefaultValueInstanceValue**[1141]
Operation **SetNewDefaultValueInstanceValue**[1150]
Operation **InsertSlotInstanceValueAt**[1162]

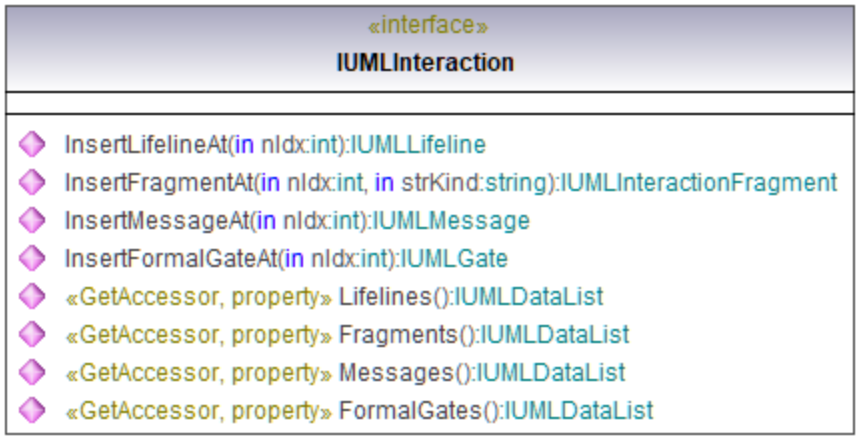
Operation **IUMLInstanceValue::Instance**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLInstanceSpecification 1085			

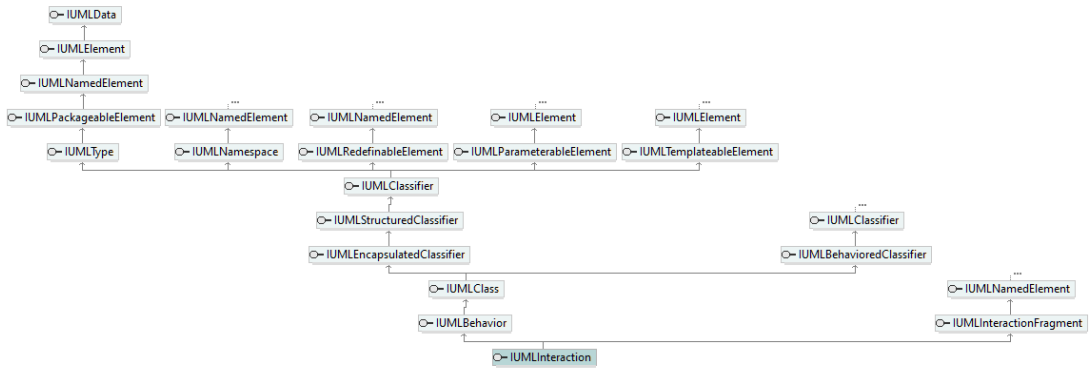
16.4.3.5.91 UModelAPI - IUMLInteraction

Interface **IUMLInteraction**

diagram



hierarchy



typedElements

Interface IUMLDataAll 911
Interface IUMLInteractionUse 1092
Operation RefersTo 964
Operation RefersTo 1093

Operation **IUMLInteraction::FormalGates**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLDataList 906			
documentation	A list of elements of type IUMLGate 1074.					

© 2019-2025 Altova GmbH

Operation **IUMLInteraction::Fragments**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			
documentation	A list of elements of type IUMLInteractionFragment 1091.					

Operation **IUMLInteraction::InsertFormalGateAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	**IUMLGate** 1074			

Operation **IUMLInteraction::InsertFragmentAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	strKind	**in**	**string**			
	return	**return**	**IUMLInteractionFragment** 1091			

Operation **IUMLInteraction::InsertLifelineAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	**IUMLLifeline** 1104			

Operation **IUMLInteraction::InsertMessageAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	**IUMLMessage** 1111			

Operation **IUMLInteraction::Lifelines**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			
documentation	A list of elements of type IUMLLifeline 1104.					

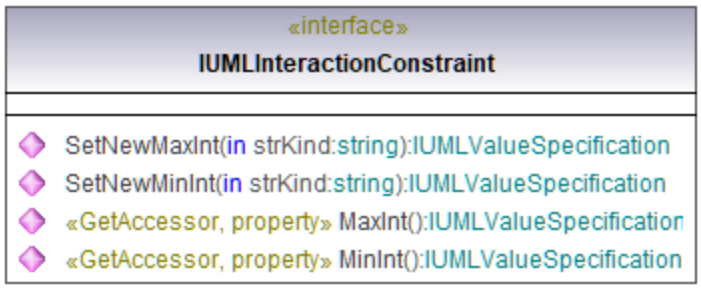
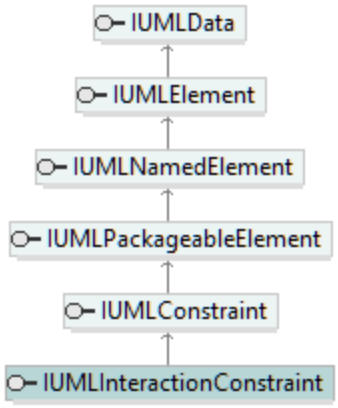
Operation **IUMLInteraction::Messages**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			
documentation	A list of elements of type IUMLMessage 1111.					

© 2019-2025 Altova GmbH

16.4.3.5.92 UModelAPI - IUMLInteractionConstraint

Interface **IUMLInteractionConstraint**

diagram	
hierarchy	
typedElements	Interface IUMLDataAll 911 Interface IUMLInteractionOperand 1091 Operation OperandGuard 956 SetNewOperandGuard 968 Operation OperandGuard 1092 SetNewOperandGuard 1092

Operation **IUMLInteractionConstraint::MaxInt**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLValueSpecification 1195			

Operation **IUMLInteractionConstraint::MinInt**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLValueSpecification 1195			

Operation **IUMLInteractionConstraint::SetNewMaxInt**

parameter	name	direction	type	type modifier	multiplicity	default
	strKind	**in**	**string**			
	return	**return**	IUMLValueSpecification 1195			

© 2019-2025 Altova GmbH

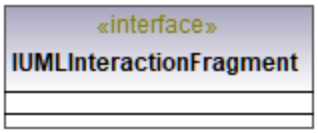
Operation **IUMLInteractionConstraint::SetNewMinInt**

parameter	name	direction	type	type modifier	multiplicity	default
	strKind	**in**	**string**			
	return	**return**	**IUMLValueSpecification** 1195			

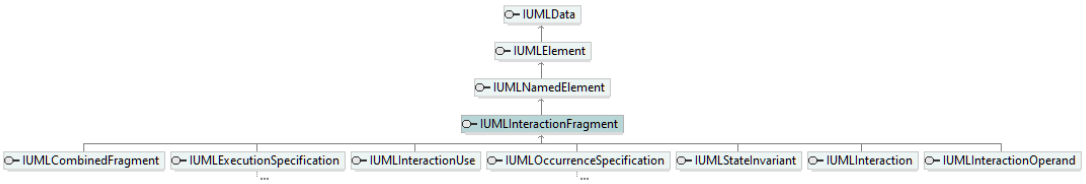
16.4.3.5.93 UModelAPI - IUMLInteractionFragment

Interface **IUMLInteractionFragment**

diagram



hierarchy



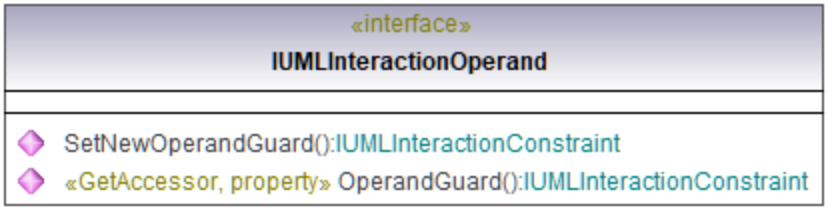
typedElements

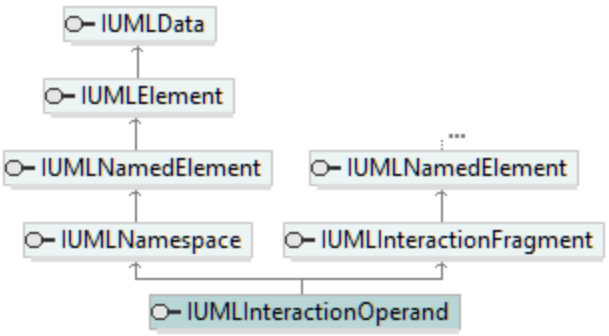
Interface **IUMLDataAll** 911
Interface **IUMLInteraction** 1088
Interface **IUMLLifeline** 1104
Operation **InsertCoveredByAt** 933
InsertFragmentAt 935
Operation **InsertFragmentAt** 1089
Operation **InsertCoveredByAt** 1104

16.4.3.5.94 UModelAPI - IUMLInteractionOperand

Interface **IUMLInteractionOperand**

diagram



hierarchy	
typedElements	Interface IUMLCombinedFragment 1024 Interface IUMLDataAll 911 Operation InsertOperandAt 1024 Operation InsertOperandAt 938

Operation **IUMLInteractionOperand::OperandGuard**

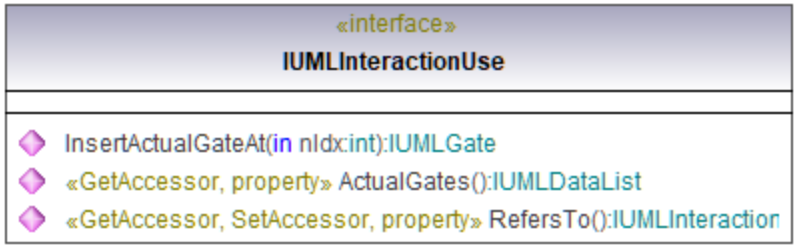
parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLInteractionConstraint 1090			

Operation **IUMLInteractionOperand::SetNewOperandGuard**

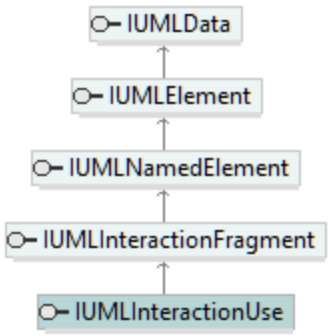
parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLInteractionConstraint 1090			

16.4.3.5.95 UModelAPI - IUMLInteractionUse

Interface **IUMLInteractionUse**

diagram	

 © 2019-2025 Altova GmbH

hierarchy	

Operation **IUMLInteractionUse::ActualGates**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			
documentation	A list of elements of type IUMLGate 1074.					

Operation **IUMLInteractionUse::InsertActualGateAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	**IUMLGate** 1074			

Operation **IUMLInteractionUse::RefersTo**

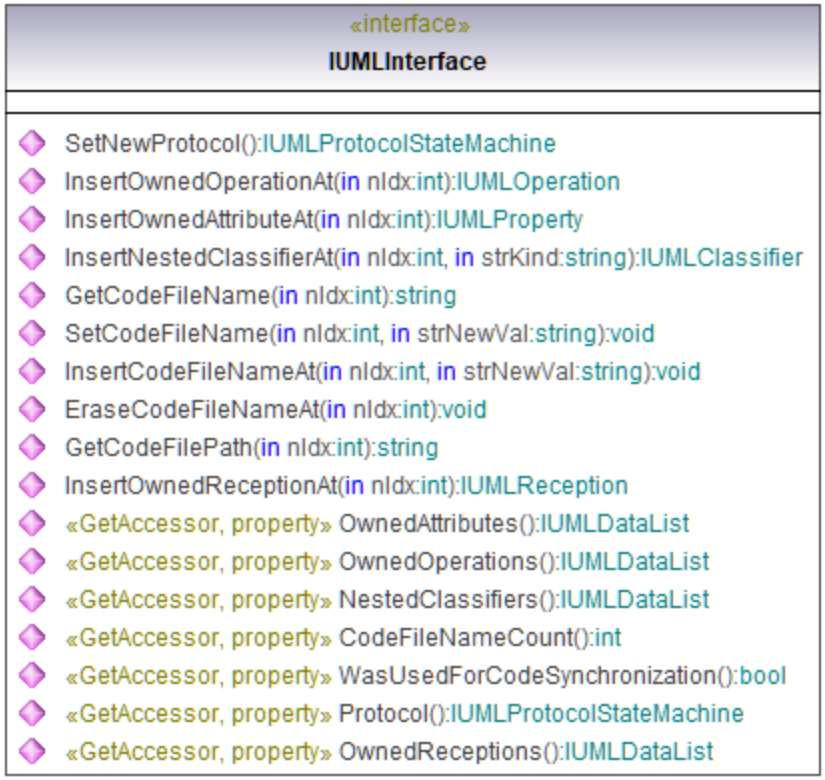
parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLInteraction** 1088			

© 2019-2025 Altova GmbH

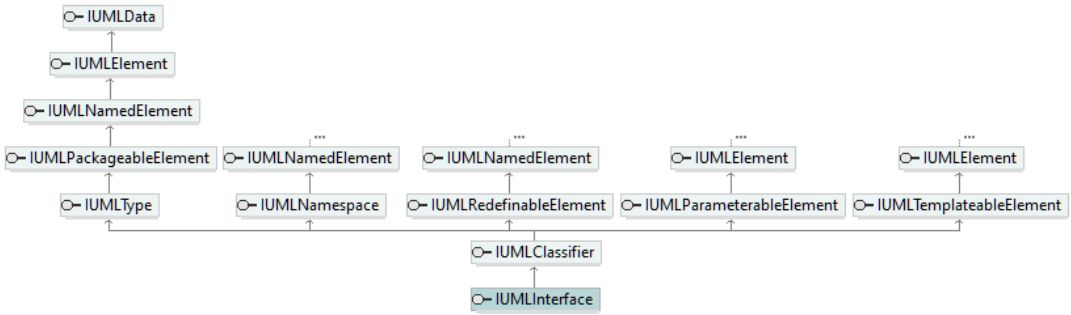
16.4.3.5.96 UModelAPI - IUMLInterface

Interface **IUMLInterface**

diagram



hierarchy



typedElements

Interface **IUMLBehavioredClassifier** 1006
Interface **IUMLClassifier** 1018
Interface **IUMLDataAll** 911

Interface **IUMLInterfaceRealization** 1097
Interface **IUMLOperation** 1131
Interface **IUMLProperty** 1147
Interface **IUMLProtocolStateMachine** 1150

Operation **InsertInterfaceRealizationAt** 1007

Operation **NestingInterface** 1020
Operation **Contract** 919
InsertInterfaceRealizationAt 936
Interface 945 **NestingInterface** 955

Operation **Contract** 1097

Operation **Interface** 1132
Operation **Interface** 1149
Operation **Interface** 1151

© 2019-2025 Altova GmbH

	Interface **IUMLReception** 1154	Operation **Interface** 1155

Operation **IUMLInterface::CodeFileNameCount**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**int**			

Operation **IUMLInterface::EraseCodeFileNameAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	**void**			

Operation **IUMLInterface::GetCodeFileName**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	**string**			

Operation **IUMLInterface::GetCodeFilePath**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	**string**			
documentation	get the full code file path					

Operation **IUMLInterface::InsertCodeFileNameAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	strNewVal	**in**	**string**			
	return	**return**	**void**			

Operation **IUMLInterface::InsertNestedClassifierAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	strKind	**in**	**string**			
	return	**return**	**IUMLClassifier** 1018			

Operation **IUMLInterface::InsertOwnedAttributeAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	**IUMLProperty** 1147			

Operation **IUMLInterface::InsertOwnedOperationAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	**IUMLOperation** 1131			

Operation **IUMLInterface::InsertOwnedReceptionAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	**IUMLReception** 1154			

Operation **IUMLInterface::NestedClassifiers**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			
documentation	A list of elements of type IUMLClassifier 1018.					

Operation **IUMLInterface::OwnedAttributes**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			
documentation	A list of elements of type IUMLProperty 1147.					

Operation **IUMLInterface::OwnedOperations**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			
documentation	A list of elements of type IUMLOperation 1131.					

Operation **IUMLInterface::OwnedReceptions**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			

Operation **IUMLInterface::Protocol**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLProtocolStateMachine** 1150			

Operation **IUMLInterface::SetCodeFileName**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	strNewVal	**in**	**string**			
	return	**return**	**void**			

Operation **IUMLInterface::SetNewProtocol**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLProtocolStateMachine** 1150			

Operation **IUMLInterface::WasUsedForCodeSynchronization**

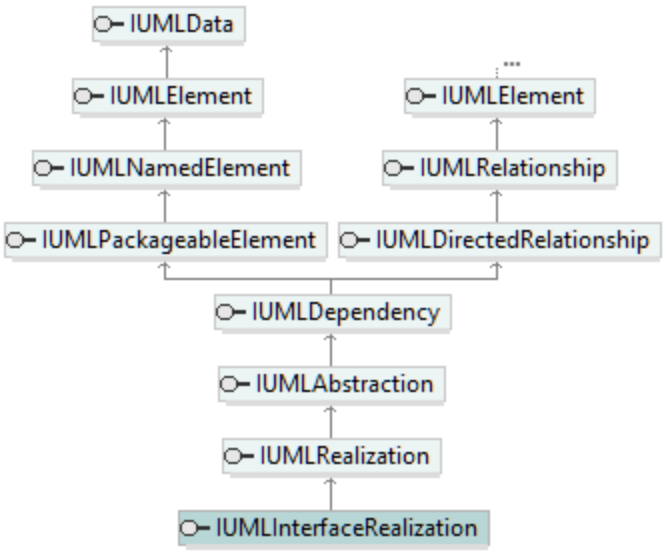
parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

© 2019-2025 Altova GmbH

UML documentation generated by **UModel** UML Editor **http://www.altova.com/umodel** Wed Sep 25 10:33:04 2024

16.4.3.5.97 UModelAPI - IUMLInterfaceRealization

Interface **IUMLInterfaceRealization**

diagram	
hierarchy	
typedElements	Interface **IUMLBehavioredClassifier** 1006 Interface **IUMLDataAll** 911 Operation **InsertInterfaceRealizationAt** 1007 Operation **InsertInterfaceRealizationAt** 936

Operation **IUMLInterfaceRealization::Contract**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLInterface** 1094			

Operation **IUMLInterfaceRealization::ImplementingClassifier**

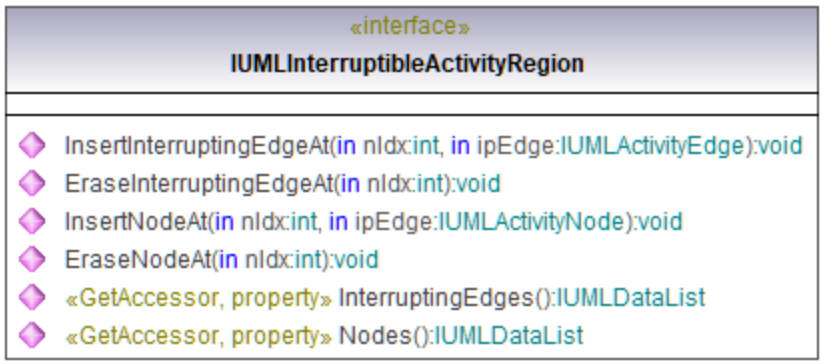
parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLBehavioredClassifier** 1006			

UML documentation generated by UModel UML Editor http://www.altova.com/umodel Wed Sep 25 10:33:04 2024

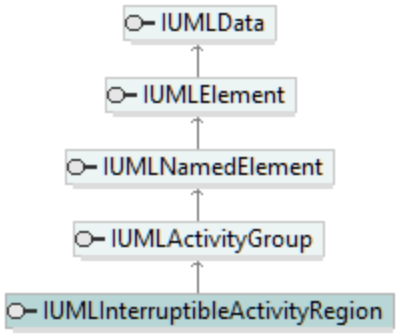
16.4.3.5.98 UModelAPI - IUMLInterruptibleActivityRegion

Interface **IUMLInterruptibleActivityRegion**

diagram



hierarchy



Operation **IUMLInterruptibleActivityRegion::EraseInterruptingEdgeAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	**void**			

Operation **IUMLInterruptibleActivityRegion::EraseNodeAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	**void**			

Operation **IUMLInterruptibleActivityRegion::InsertInterruptingEdgeAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	ipEdge	**in**	**IUMLActivityEdge** 988			
	return	**return**	**void**			

 © 2019-2025 Altova GmbH

Operation **IUMLInterruptibleActivityRegion::InsertNodeAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	ipEdge	**in**	**IUMLActivityNode** 992			
	return	**return**	**void**			

Operation **IUMLInterruptibleActivityRegion::InterruptingEdges**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			
documentation	A list of elements of type IUMLActivityEdge 988.					

Operation **IUMLInterruptibleActivityRegion::Nodes**

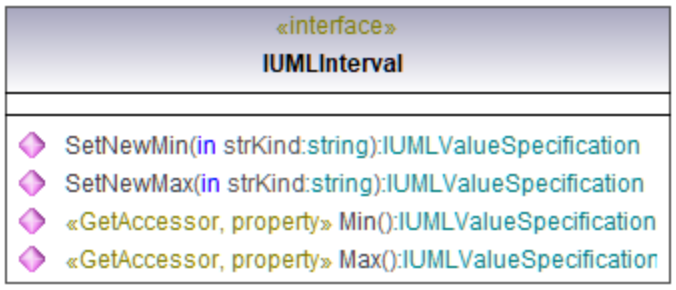
parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			
documentation	A list of elements of type IUMLActivityNode 992.					

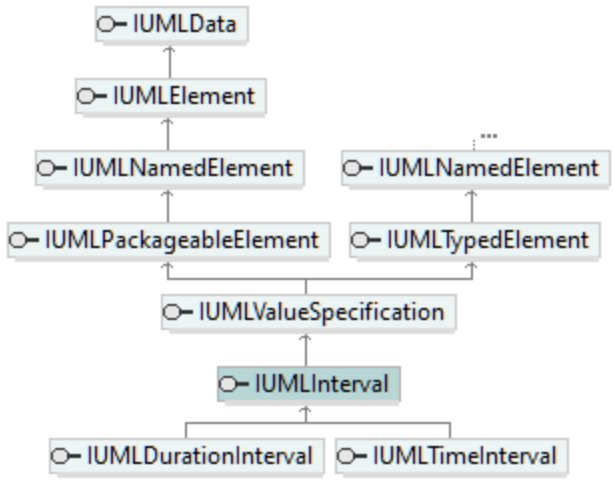
UML documentation generated by **UModel** UML Editor **http://www.altova.com/umodel** Wed Sep 25 10:33:04 2024

16.4.3.5.99 UModelAPI - IUMLInterval

Interface **IUMLInterval**

diagram



hierarchy	
	

Operation **IUMLInterval::Max**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLValueSpecification** 1195			

Operation **IUMLInterval::Min**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLValueSpecification** 1195			

Operation **IUMLInterval::SetNewMax**

parameter	name	direction	type	type modifier	multiplicity	default
	strKind	**in**	**string**			
	return	**return**	**IUMLValueSpecification** 1195			

Operation **IUMLInterval::SetNewMin**

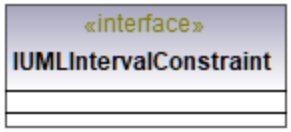
parameter	name	direction	type	type modifier	multiplicity	default
	strKind	**in**	**string**			
	return	**return**	**IUMLValueSpecification** 1195			

© 2019-2025 Altova GmbH

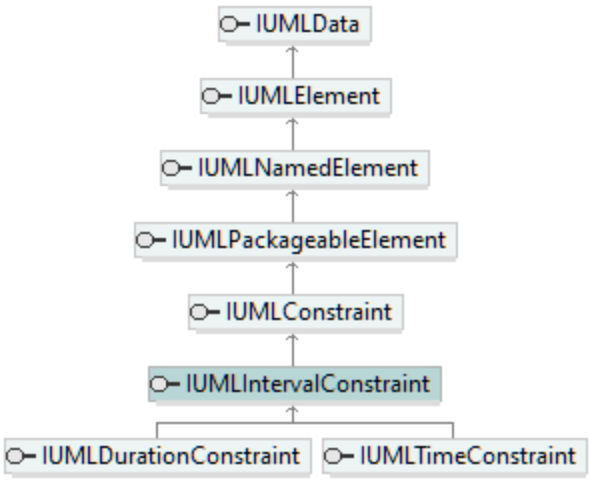
16.4.3.5.100 UModelAPI - IUMLIntervalConstraint

Interface **IUMLIntervalConstraint**

diagram



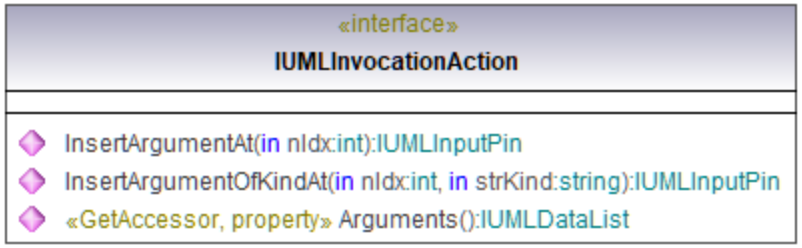
hierarchy

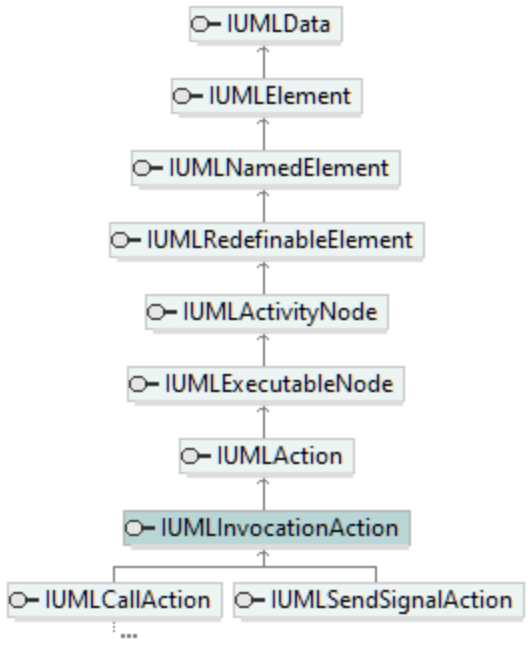


16.4.3.5.101 UModelAPI - IUMLInvocationAction

Interface **IUMLInvocationAction**

diagram



hierarchy	

Operation **IUMLInvocationAction::Arguments**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			
documentation	A list of elements of type IUMLInputPin 1083.					

Operation **IUMLInvocationAction::InsertArgumentAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	**IUMLInputPin** 1083			

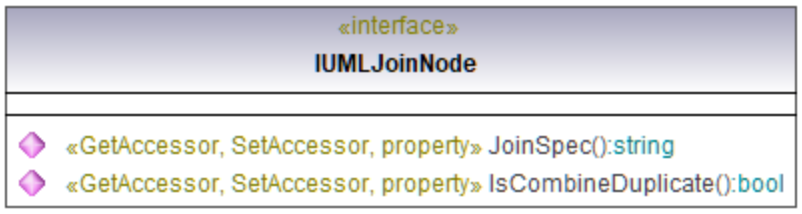
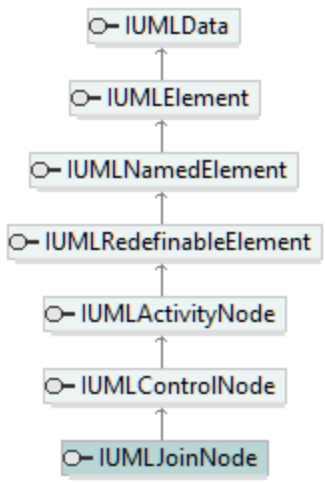
Operation **IUMLInvocationAction::InsertArgumentOfKindAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	strKind	**in**	**string**			
	return	**return**	**IUMLInputPin** 1083			

© 2019-2025 Altova GmbH

16.4.3.5.102 UModelAPI - IUMLJoinNode

Interface **IUMLJoinNode**

diagram	
hierarchy	

Operation **IUMLJoinNode::IsCombineDuplicate**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

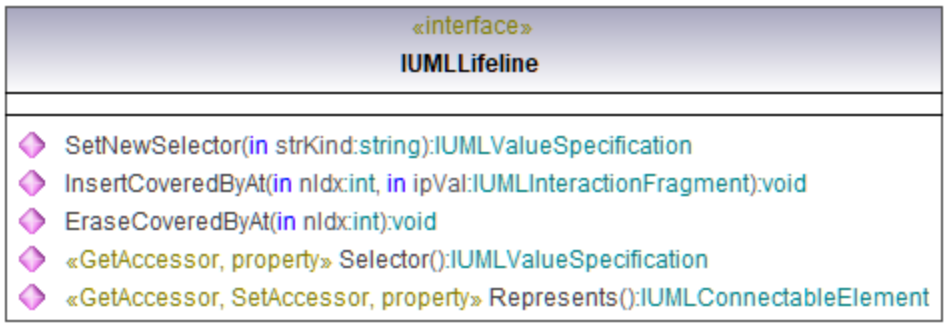
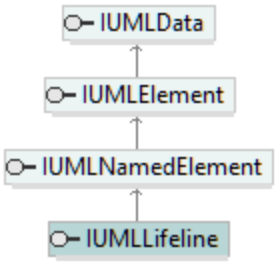
Operation **IUMLJoinNode::JoinSpec**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**string**			

© 2019-2025 Altova GmbH

16.4.3.5.103 UModelAPI - IUMLLifeline

Interface **IUMLLifeline**

diagram	
hierarchy	
typedElements	Interface IUMLDataAll 911 Interface IUMLInteraction 1088 Interface IUMLMessage 1111 Interface IUMLOccurrenceSpecification 1127 Interface IUMLStateInvariant 1166 Operation Covered 920 GetSourceLifeline 928 GetTargetLifeline 928 InsertLifelineAt 937 Operation InsertLifelineAt 1089 Operation GetSourceLifeline 1111 GetTargetLifeline 1111 Operation Covered 1127 Operation Covered 1167

Operation **IUMLLifeline::EraseCoveredByAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	**void**			

Operation **IUMLLifeline::InsertCoveredByAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	ipVal	**in**	**IUMLInteractionFragment** 1091			
	return	**return**	**void**			

Operation **IUMLLifeline::Represents**

parameter	name	direction	type	type modifier	multiplicity	default

© 2019-2025 Altova GmbH

	return	**return**	**IUMLConnectableElement** 1030

Operation **IUMLLifeline::Selector**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLValueSpecification** 1195			

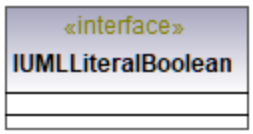
Operation **IUMLLifeline::SetNewSelector**

parameter	name	direction	type	type modifier	multiplicity	default
	strKind	**in**	**string**			
	return	**return**	**IUMLValueSpecification** 1195			

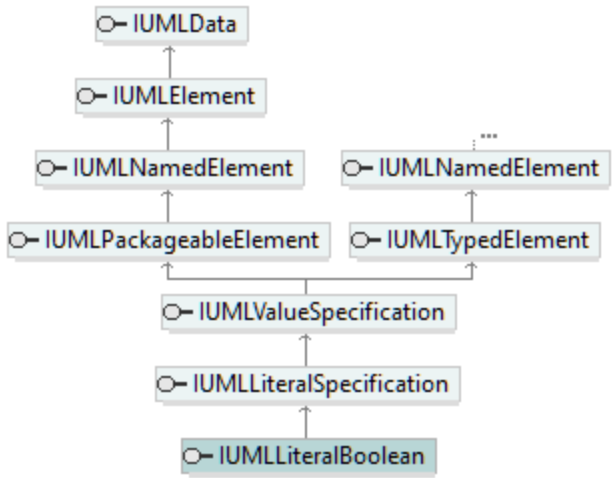
16.4.3.5.104 UModelAPI - IUMLLiteralBoolean

Interface **IUMLLiteralBoolean**

diagram



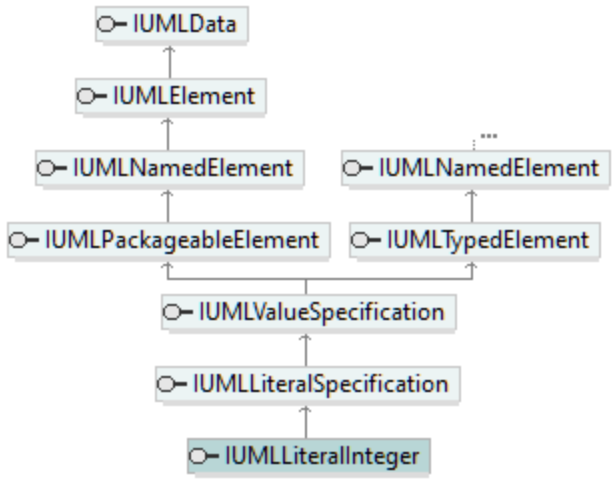
hierarchy



© 2019-2025 Altova GmbH

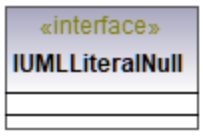
16.4.3.5.105 UModelAPI - IUMLLiteralInteger

Interface **IUMLLiteralInteger**

diagram	
hierarchy	

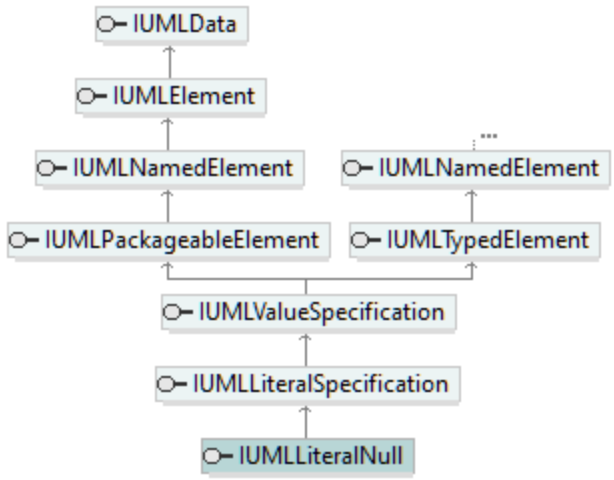
16.4.3.5.106 UModelAPI - IUMLLiteralNull

Interface **IUMLLiteralNull**

diagram	

 © 2019-2025 Altova GmbH

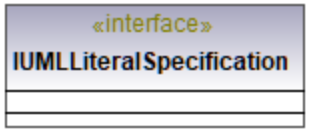
hierarchy



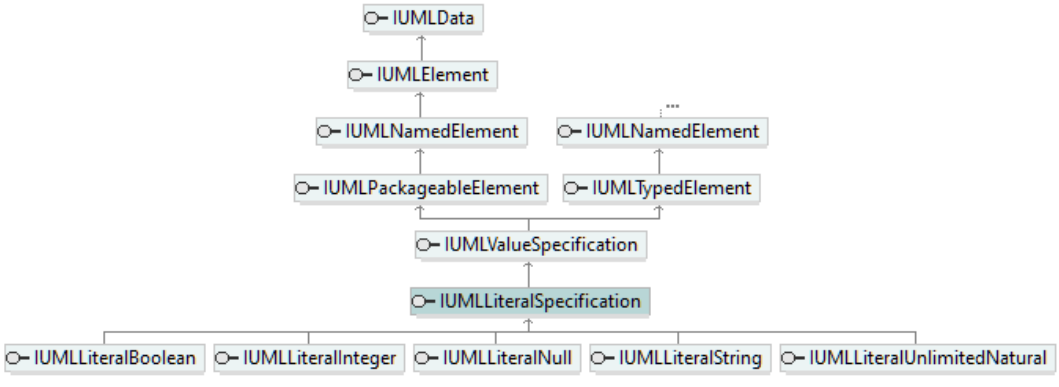
16.4.3.5.107 UModelAPI - IUMLLiteralSpecification

Interface **IUMLLiteralSpecification**

diagram



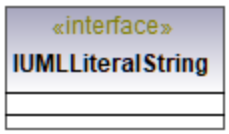
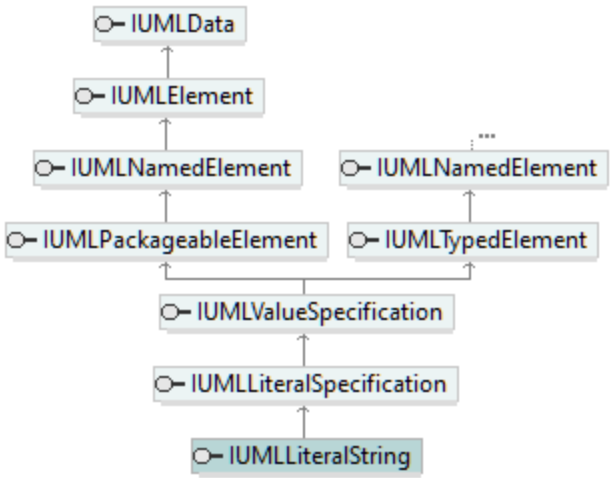
hierarchy



© 2019-2025 Altova GmbH

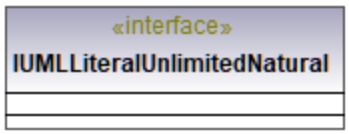
16.4.3.5.108 UModelAPI - IUMLLiteralString

Interface **IUMLLiteralString**

diagram	
hierarchy	
typedElements	Interface IUMLConstraint 1035 Interface IUMLDataAll 911 Interface IUMLInstanceSpecification 1085 Interface IUMLParameter 1140 Interface IUMLProperty 1147 Operation SetNewSpecificationLiteralString 1037 Operation SetNewDefaultValueLiteralString 967 SetNewSpecificationLiteralString 969 Operation SetNewSpecificationLiteralString 1086 Operation SetNewDefaultValueLiteralString 1141 Operation SetNewDefaultValueLiteralString 1150

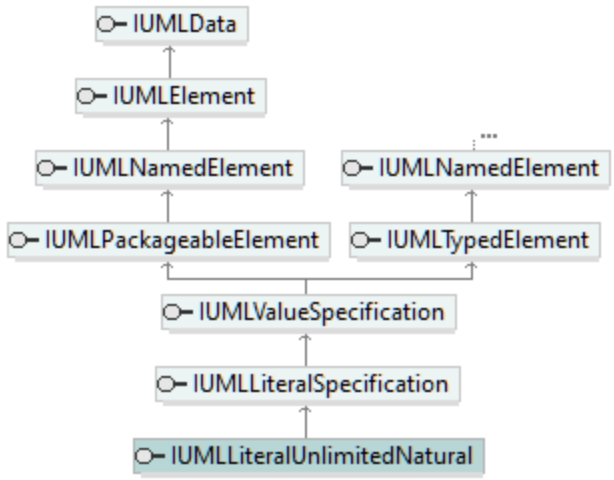
16.4.3.5.109 UModelAPI - IUMLLiteralUnlimitedNatural

Interface **IUMLLiteralUnlimitedNatural**

diagram	

 © 2019-2025 Altova GmbH

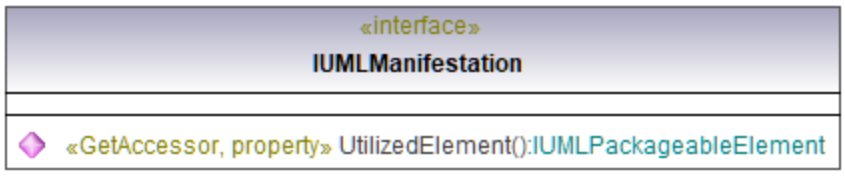
hierarchy



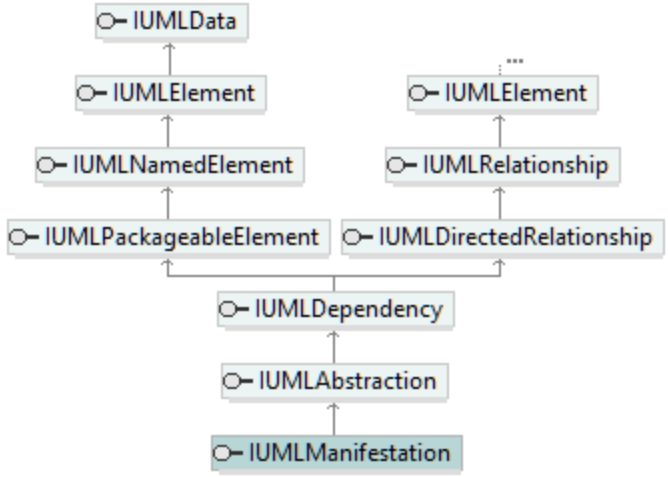
16.4.3.5.110 UModelAPI - IUMLManifestation

Interface **IUMLManifestation**

diagram



hierarchy



typedElements	Interface IUMLArtifact 998 Interface IUMLDataAll 911	Operation InsertManifestationAt 999 Operation InsertManifestationAt 937

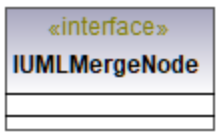
Operation **IUMLManifestation::UtilizedElement**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLPackageableElement 1137			

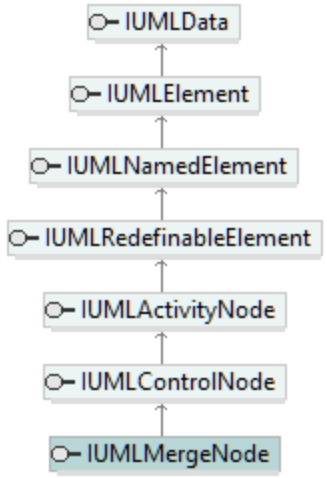
16.4.3.5.111 UModelAPI - IUMLMergeNode

Interface **IUMLMergeNode**

diagram



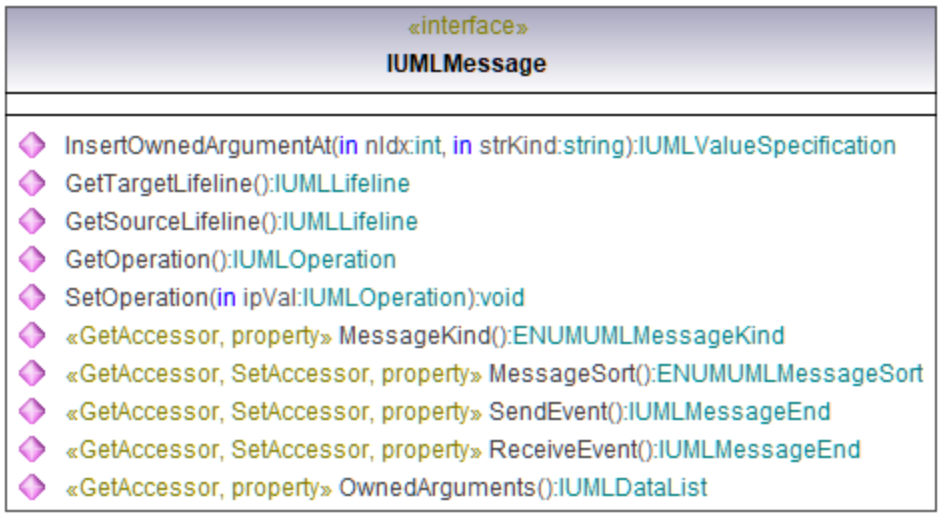
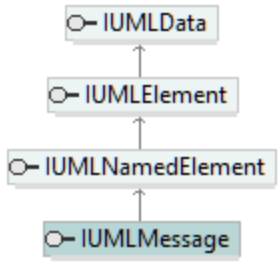
hierarchy



 © 2019-2025 Altova GmbH

16.4.3.5.112 UModelAPI - IUMLMessage

Interface **IUMLMessage**

diagram	
hierarchy	
typedElements	Interface **IUMLDataAll** 911 Interface **IUMLInteraction** 1088 Interface **IUMLMessageEnd** 1113 Operation **InsertMessageAt** 937 **Message** 953 Operation **InsertMessageAt** 1089 Operation **Message** 1113

Operation **IUMLMessage::GetOperation**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLOperation** 1131			

Operation **IUMLMessage::GetSourceLifeline**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLLifeline** 1104			

Operation **IUMLMessage::GetTargetLifeline**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLLifeline** 1104			

Operation **IUMLMessage::InsertOwnedArgumentAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	strKind	**in**	**string**			
	return	**return**	**IUMLValueSpecification** 1195			

Operation **IUMLMessage::MessageKind**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**ENUMUMLMessageKind** 1270			

Operation **IUMLMessage::MessageSort**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**ENUMUMLMessageSort** 1270			

Operation **IUMLMessage::OwnedArguments**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			
documentation	A list of elements of type IUMLValueSpecification 1195.					

Operation **IUMLMessage::ReceiveEvent**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLMessageEnd** 1113			

Operation **IUMLMessage::SendEvent**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLMessageEnd** 1113			

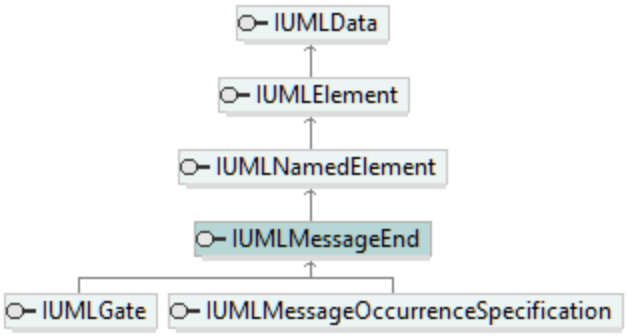
Operation **IUMLMessage::SetOperation**

parameter	name	direction	type	type modifier	multiplicity	default
	ipVal	**in**	**IUMLOperation** 1131			
	return	**return**	**void**			

© 2019-2025 Altova GmbH

16.4.3.5.113 UModelAPI - IUMLMessageEnd

Interface **IUMLMessageEnd**

diagram	
hierarchy	
typedElements	Interface **IUMLDataAll** 911 Interface **IUMLMessage** 1111 Operation **ReceiveEvent** 963 **SendEvent** 965 Operation **ReceiveEvent** 1112 **SendEvent** 1112

Operation **IUMLMessageEnd::Message**

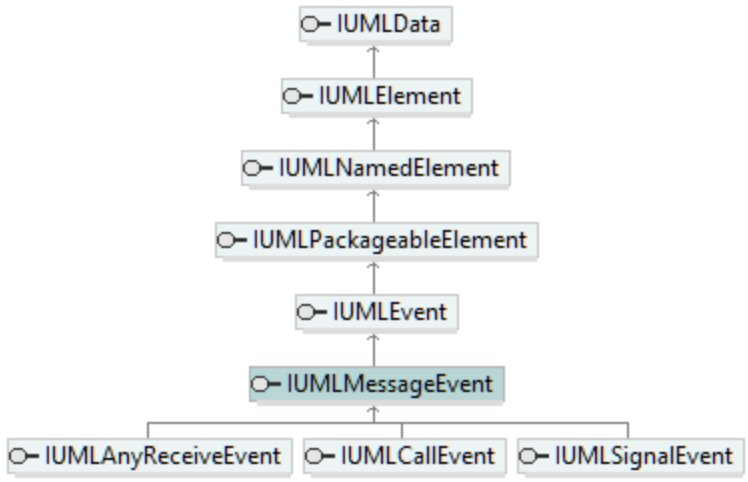
parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLMessage** 1111			

16.4.3.5.114 UModelAPI - IUMLMessageEvent

Interface **IUMLMessageEvent**

diagram	

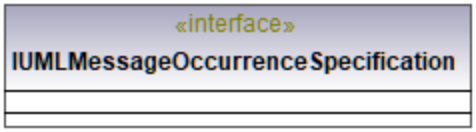
hierarchy



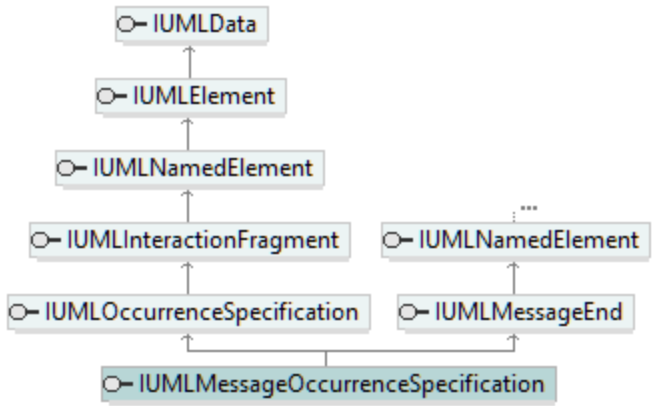
16.4.3.5.115 UModelAPI - IUMLMessageOccurrenceSpecification

Interface **IUMLMessageOccurrenceSpecification**

diagram



hierarchy

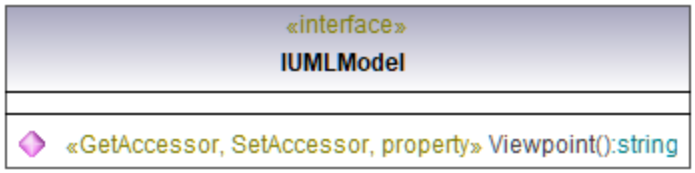
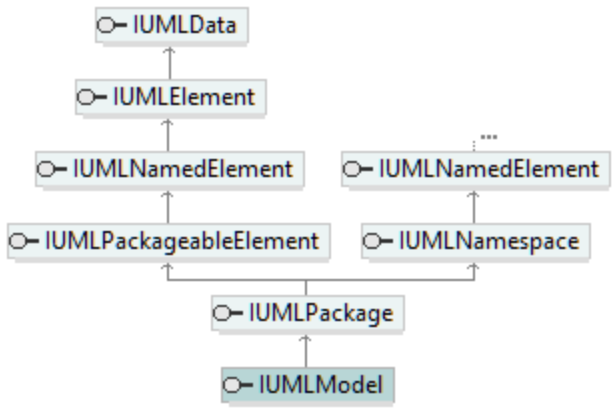


 © 2019-2025 Altova GmbH

UML documentation generated by **UModel** UML Editor **http://www.altova.com/umodel** Wed Sep 25 10:33:04 2024

16.4.3.5.116 UModelAPI - IUMLModel

Interface **IUMLModel**

diagram	
hierarchy	

Operation **IUMLModel::Viewpoint**

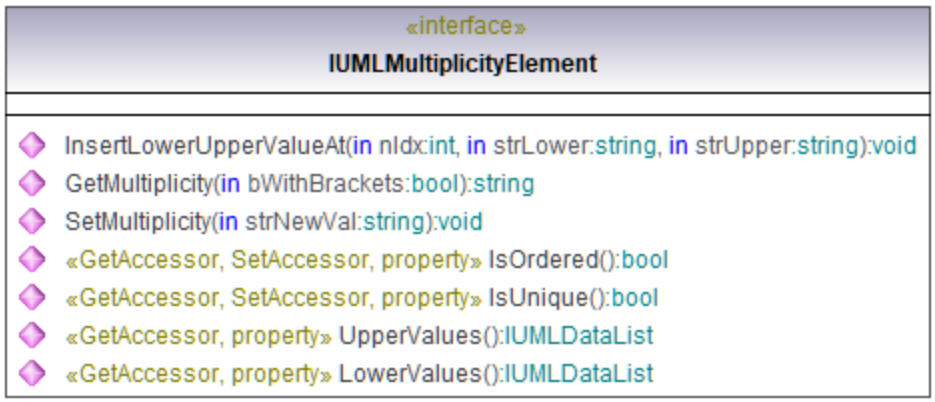
parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**string**			

UML documentation generated by **UModel** UML Editor **http://www.altova.com/umodel** Wed Sep 25 10:33:04 2024

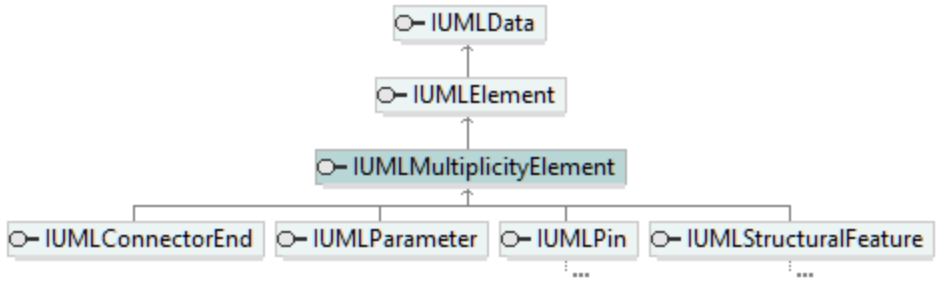
16.4.3.5.117 UModelAPI - IUMLMultiplicityElement

Interface **IUMLMultiplicityElement**

diagram



hierarchy



typedElements

Interface IUMLDataAll 911
Interface IUMLValueSpecification 1195
Operation OwningLower 960 OwningUpper 961
Operation OwningLower 1197 OwningUpper 1197

Operation **IUMLMultiplicityElement::GetMultiplicity**

parameter	name	direction	type	type modifier	multiplicity	default
	bWithBrackets	**in**	**bool**			
	return	**return**	**string**			

Operation **IUMLMultiplicityElement::InsertLowerUpperValueAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	strLower	**in**	**string**			
	strUpper	**in**	**string**			
	return	**return**	**void**			

Operation **IUMLMultiplicityElement::IsOrdered**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IUMLMultiplicityElement::IsUnique**

© 2019-2025 Altova GmbH

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IUMLMultiplicityElement::LowerValues**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			
documentation	A list of elements of type IUMLValueSpecification 1195.					

Operation **IUMLMultiplicityElement::SetMultiplicity**

parameter	name	direction	type	type modifier	multiplicity	default
	strNewVal	**in**	**string**			
	return	**return**	**void**			

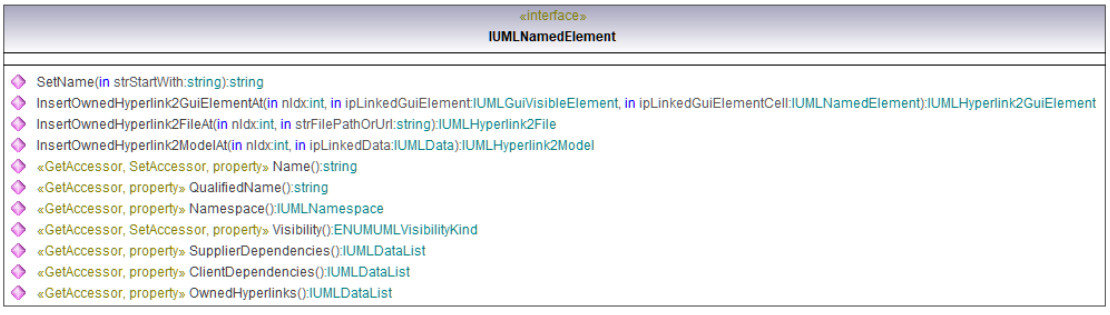
Operation **IUMLMultiplicityElement::UpperValues**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			
documentation	A list of elements of type IUMLValueSpecification 1195.					

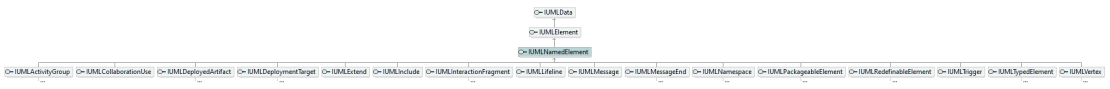
16.4.3.5.118 UModelAPI - IUMLNamedElement

Interface **IUMLNamedElement**

diagram



hierarchy



typedElements

Interface **IUMLCommentTextHyperlink** 1026
Interface **IUMLDataAll** 911

Operation **SetHyperlinkGuiElementAddress** 1027
Operation **FindOwnedMemberWithQualifiedName** 926
InsertInformationSourceAt 936
InsertInformationTargetAt 936
InsertOwnedHyperlink2GuiElementA

	t 940 LinkedGuiElementCell 951 SetHyperlinkGuiElementAddress 966 TimeObservationEvent 976 Interface IUMLGuiTextHyperlink 1250 — Operation SetHyperlinkGuiElementAddress 1251 Interface IUMLHyperlink2GuiElement 1077 — Operation LinkedGuiElementCell 1078 Interface IUMLInformationFlow 1080 — Operation InsertInformationSourceAt 1081 InsertInformationTargetAt 1082 Interface IUMLNamedElement 1117 — Operation InsertOwnedHyperlink2GuiElementAt 1118 Interface IUMLNamespace 1120 — Operation FindOwnedMemberWithQualifiedName 1120 Interface IUMLTimeObservation 1186 — Operation TimeObservationEvent 1186

Operation **IUMLNamedElement::ClientDependencies**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLDataList 906			
documentation	A list of elements of type IUMLDependency 1041.					

Operation **IUMLNamedElement::InsertOwnedHyperlink2FileAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	strFilePathOrUrl	**in**	**string**			
	return	**return**	IUMLHyperlink2File 1076			

Operation **IUMLNamedElement::InsertOwnedHyperlink2GuiElementAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	ipLinkedGuiElement	**in**	IUMLGuiVisibleElement 1260			
	ipLinkedGuiElementCell	**in**	IUMLNamedElement 1117			
	return	**return**	IUMLHyperlink2GuiElement 1077			

Operation **IUMLNamedElement::InsertOwnedHyperlink2ModelAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	ipLinkedData	**in**	IUMLData 904			
	return	**return**	IUMLHyperlink2Model 1078			

Operation **IUMLNamedElement::Name**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**string**			

Operation **IUMLNamedElement::Namespace**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLNamespace 1120			

© 2019-2025 Altova GmbH

Operation **IUMLNamedElement::OwnedHyperlinks**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			
documentation	A list of elements of type IUMLHyperlink 1075.					

Operation **IUMLNamedElement::QualifiedName**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**string**			

Operation **IUMLNamedElement::SetName**

parameter	name	direction	type	type modifier	multiplicity	default
	strStartWith	**in**	**string**			
	return	**return**	**string**			
documentation	This function will find and set a unique name (starting with 'strStartWith') that the element is distinguishable in its parent namespace.					

Operation **IUMLNamedElement::SupplierDependencies**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			
documentation	A list of elements of type IUMLDependency 1041.					

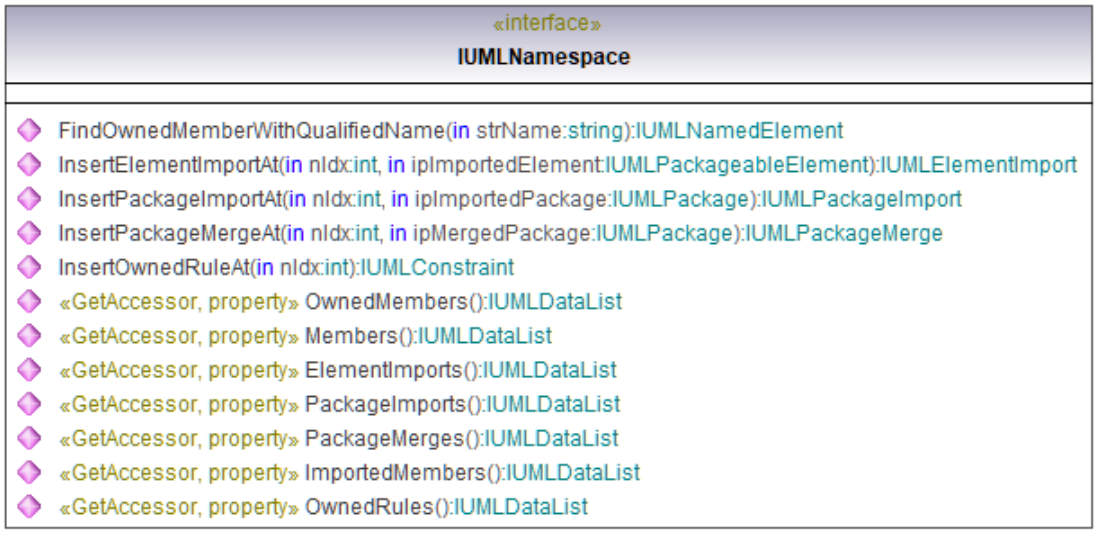
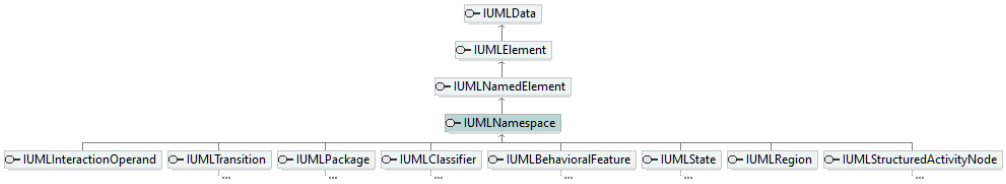
Operation **IUMLNamedElement::Visibility**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**ENUMUMLVisibilityKind** 1273			

© 2019-2025 Altova GmbH

16.4.3.5.119 UModelAPI - IUMLNamespace

Interface **IUMLNamespace**

diagram	
hierarchy	
typedElements	Interface **IUMLConstraint** 1035 Interface **IUMLDataAll** 911 Interface **IUMLElementImport** 1054 Interface **IUMLNamedElement** 1117 Interface **IUMLPackageImport** 1138 Operation **Context** 1036 Operation **Context** 919 **ImportingNamespace** 930 **Namespace** 954 Operation **ImportingNamespace** 1054 Operation **Namespace** 1118 Operation **ImportingNamespace** 1138

Operation **IUMLNamespace::ElementImports**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			
documentation	A list of elements of type IUMLElementImport 1054.					

Operation **IUMLNamespace::FindOwnedMemberWithQualifiedName**

parameter	name	direction	type	type modifier	multiplicity	default
	strName	**in**	**string**			
	return	**return**	**IUMLNamedElement** 1117			

Operation **IUMLNamespace::ImportedMembers**

parameter	name	direction	type	type modifier	multiplicity	default

© 2019-2025 Altova GmbH

	return	**return**	**IUMLDataList** 906
documentation	A list of elements of type IUMLPackageableElement 1137.		

Operation **IUMLNamespace::InsertElementImportAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	ipImportedElement	**in**	**IUMLPackageableElement** 1137			
	return	**return**	**IUMLElementImport** 1054			

Operation **IUMLNamespace::InsertOwnedRuleAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	**IUMLConstraint** 1035			

Operation **IUMLNamespace::InsertPackageImportAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	ipImportedPackage	**in**	**IUMLPackage** 1134			
	return	**return**	**IUMLPackageImport** 1138			

Operation **IUMLNamespace::InsertPackageMergeAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	ipMergedPackage	**in**	**IUMLPackage** 1134			
	return	**return**	**IUMLPackageMerge** 1139			

Operation **IUMLNamespace::Members**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			
documentation	A list of elements of type IUMLNamedElement 1117.					

Operation **IUMLNamespace::OwnedMembers**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			
documentation	A list of elements of type IUMLNamedElement 1117.					

Operation **IUMLNamespace::OwnedRules**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			

documentation	A list of elements of type IUMLConstraint[1035].

Operation **IUMLNamespace::PackageImports**

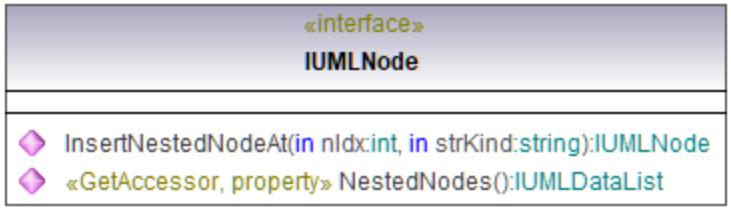
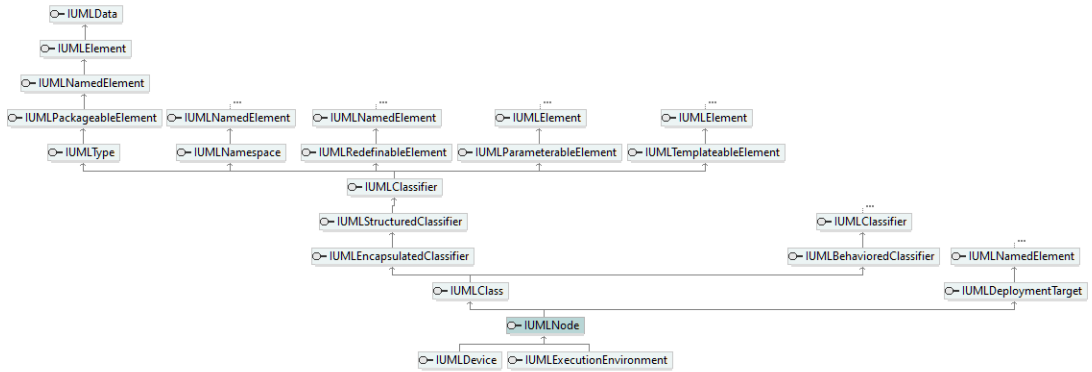
parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList**[906]			
documentation	A list of elements of type IUMLPackageImport[1138].					

Operation **IUMLNamespace::PackageMerges**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList**[906]			
documentation	A list of elements of type IUMLPackageMerge[1139].					

16.4.3.5.120 UModelAPI - IUMLNode

Interface **IUMLNode**

diagram	
hierarchy	
typedElements	Interface **IUMLDataAll**[911] Interface **IUMLNode**[1122]	Operation **InsertNestedNodeAt**[938] Operation **InsertNestedNodeAt**[1122]

Operation **IUMLNode::InsertNestedNodeAt**

 © 2019-2025 Altova GmbH

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	strKind	**in**	**string**			
	return	**return**	**IUMLNode** 1122			

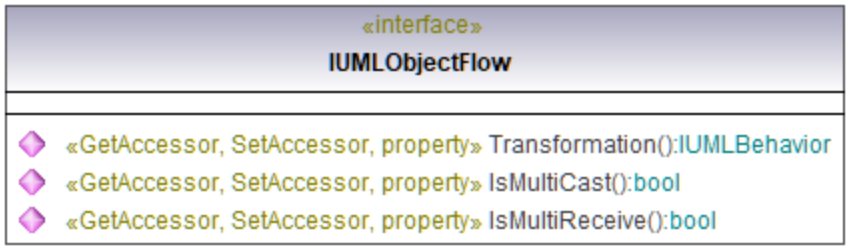
Operation **IUMLNode::NestedNodes**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			
documentation	A list of elements of type IUMLNode 1122.					

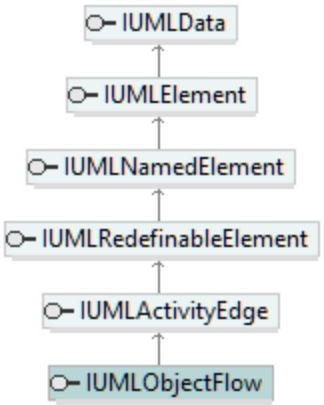
16.4.3.5.121 UModelAPI - IUMLObjectFlow

Interface **IUMLObjectFlow**

diagram



hierarchy



Operation **IUMLObjectFlow::IsMultiCast**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IUMLObjectFlow::IsMultiReceive**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

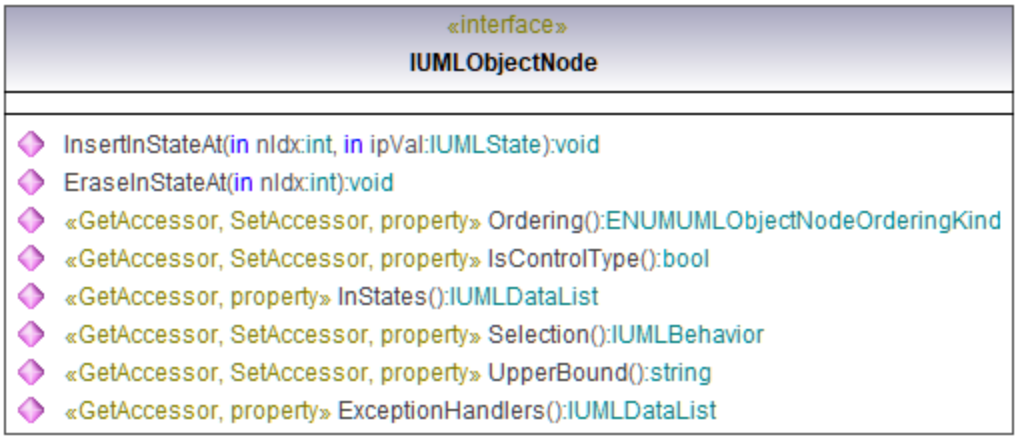
Operation **IUMLObjectFlow::Transformation**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLBehavior 1002			

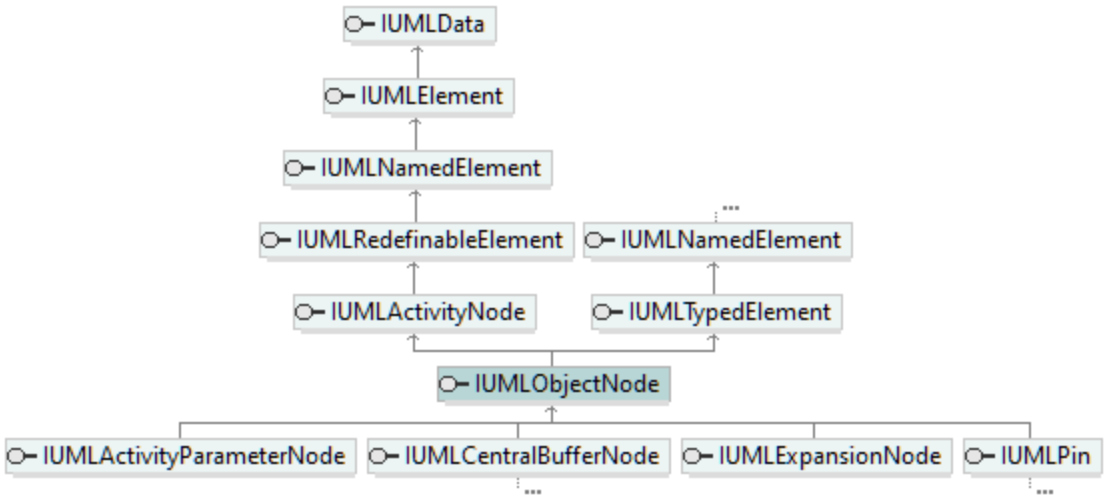
16.4.3.5.122 UModelAPI - IUMLObjectNode

Interface **IUMLObjectNode**

diagram



hierarchy



 © 2019-2025 Altova GmbH

typedElements	Interface **IUMLDataAll** 911 Interface **IUMLExceptionHandler** 1059	Operation **ExceptionInput** 925 Operation **ExceptionInput** 1060

Operation **IUMLObjectNode::EraseInStateAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	**void**			

Operation **IUMLObjectNode::ExceptionHandlers**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			
documentation	A list of elements of type IUMLExceptionHandler 1059.					

Operation **IUMLObjectNode::InsertInStateAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	ipVal	**in**	**IUMLState** 1163			
	return	**return**	**void**			

Operation **IUMLObjectNode::InStates**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			
documentation	A list of elements of type IUMLState 1163.					

Operation **IUMLObjectNode::IsControlType**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IUMLObjectNode::Ordering**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**ENUMUMLObjectNodeOrderingKind** 1271			

Operation **IUMLObjectNode::Selection**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLBehavior** 1002			

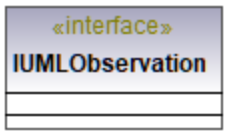
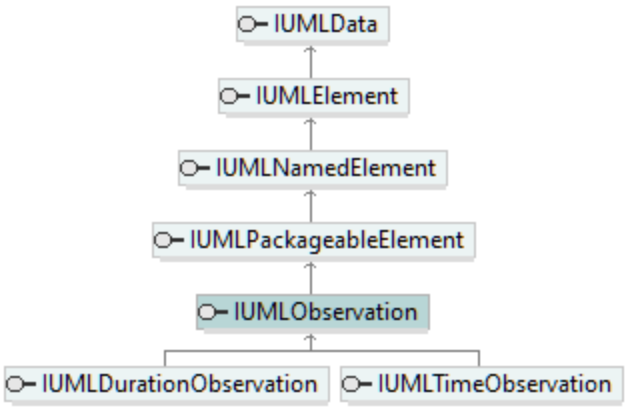
Operation **IUMLObjectNode::UpperBound**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**string**			

UML documentation generated by UModel UML Editor http://www.altova.com/umodel Wed Sep 25 10:33:04 2024

16.4.3.5.123 UModelAPI - IUMLObservation

Interface **IUMLObservation**

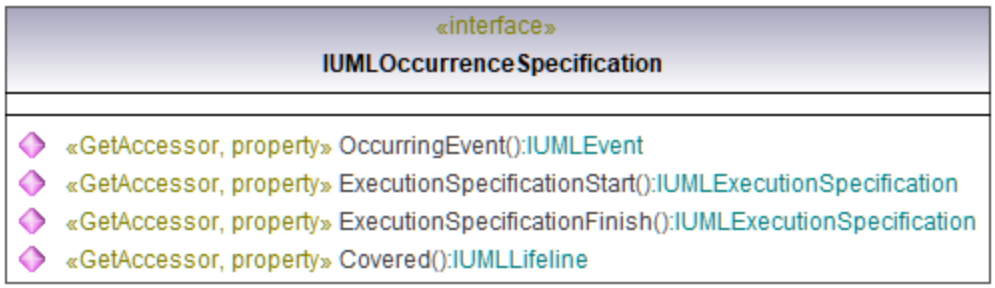
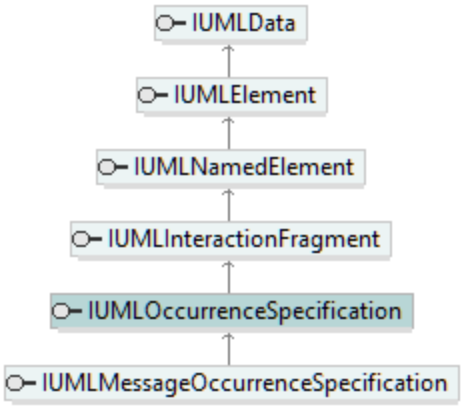
diagram	
hierarchy	
typedElements	Interface IUMLDataAll 911 Interface IUMLDuration 1047 Interface IUMLTimeExpression 1184 Operation InsertObservationAt 938 Operation InsertObservationAt 1047 Operation InsertObservationAt 1184

UML documentation generated by UModel UML Editor http://www.altova.com/umodel Wed Sep 25 10:33:04 2024

 © 2019-2025 Altova GmbH

16.4.3.5.124 UModelAPI - IUMLOccurrenceSpecification

Interface **IUMLOccurrenceSpecification**

diagram	
hierarchy	
typedElements	Interface **IUMLDataAll** 911 Interface **IUMLExecutionSpecification** 1062 Operation **Finish** 927 **Start** 973 Operation **Finish** 1063 **Start** 1063

Operation **IUMLOccurrenceSpecification::Covered**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLLifeline** 1104			

Operation **IUMLOccurrenceSpecification::ExecutionSpecificationFinish**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLExecutionSpecification** 1062			

Operation **IUMLOccurrenceSpecification::ExecutionSpecificationStart**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLExecutionSpecification** 1062			

Operation **IUMLOccurrenceSpecification::OccurringEvent**

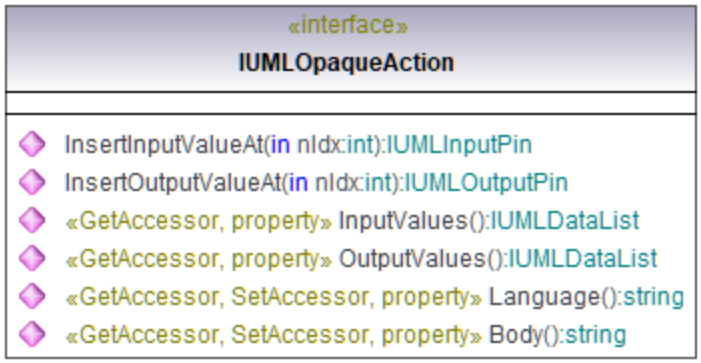
parameter	name	direction	type	type modifier	multiplicity	default

	return	**return**	**IUMLEvent**[1058]

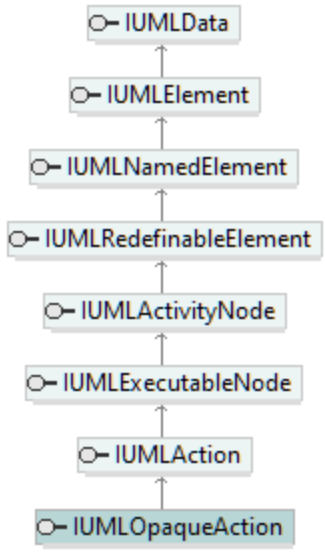
16.4.3.5.125 UModelAPI - IUMLOpaqueAction

Interface **IUMLOpaqueAction**

diagram



hierarchy



Operation **IUMLOpaqueAction::Body**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**string**			

Operation **IUMLOpaqueAction::InputValues**

parameter	name	direction	type	type modifier	multiplicity	default

 © 2019-2025 Altova GmbH

	return	**return**	IUMLDataList 906

Operation **IUMLOpaqueAction::InsertInputValueAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	IUMLInputPin 1083			

Operation **IUMLOpaqueAction::InsertOutputValueAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	IUMLOutputPin 1133			

Operation **IUMLOpaqueAction::Language**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**string**			

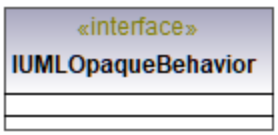
Operation **IUMLOpaqueAction::OutputValues**

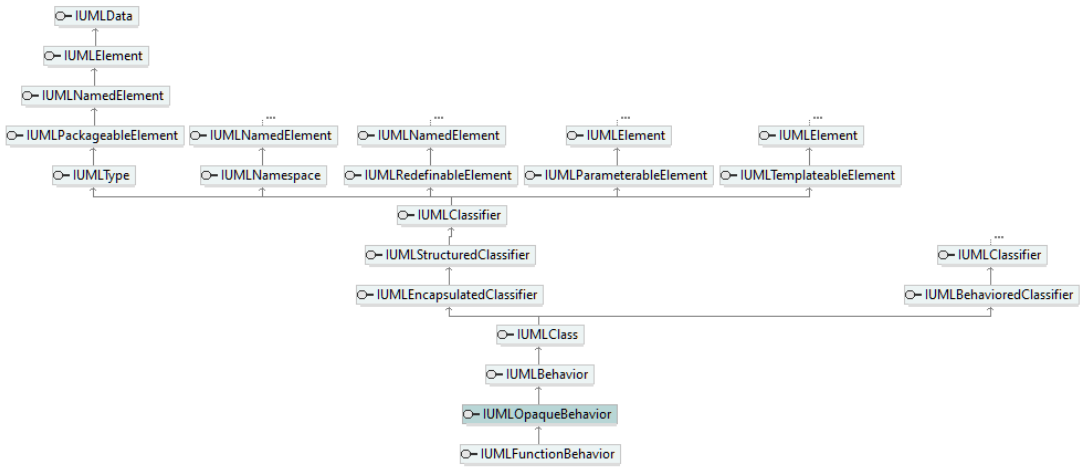
parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLDataList 906			

UML documentation generated by **UModel** UML Editor **http://www.altova.com/umodel** Wed Sep 25 10:33:04 2024

16.4.3.5.126 UModelAPI - IUMLOpaqueBehavior

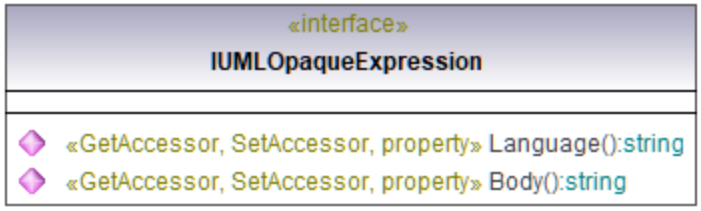
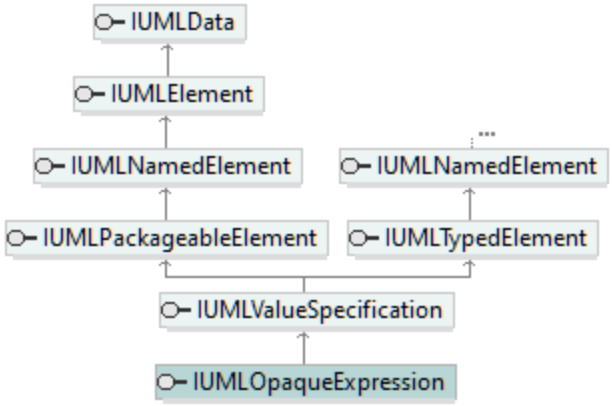
Interface **IUMLOpaqueBehavior**

diagram	

hierarchy	

16.4.3.5.127 UModelAPI - IUMLOpaqueExpression

Interface **IUMLOpaqueExpression**

diagram	
hierarchy	
typedElements	Interface IUMLAbstraction 980 Interface IUMLDataAll 911 Operation Mapping 981 SetNewMapping 981 Operation Mapping 952 SetNewMapping 968

 © 2019-2025 Altova GmbH

Operation **IUMLOpaqueExpression::Body**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**string**			

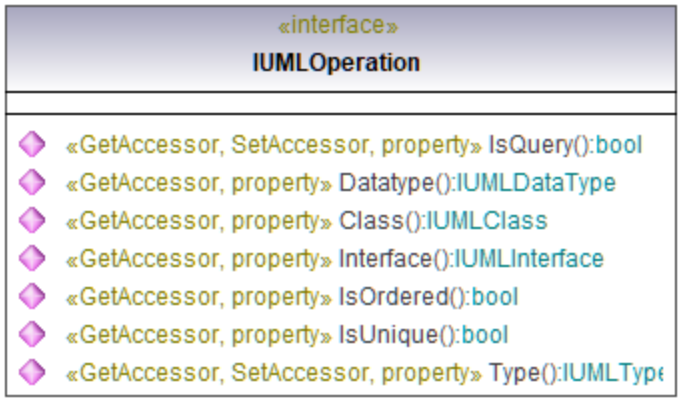
Operation **IUMLOpaqueExpression::Language**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**string**			

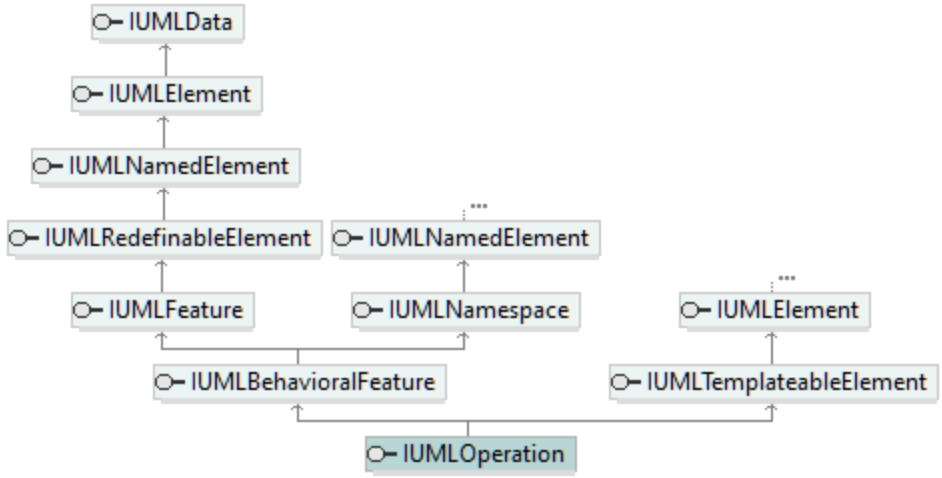
16.4.3.5.128 UModelAPI - IUMLOperation

Interface **IUMLOperation**

diagram



hierarchy



typedElements

Interface **IDocument** 832 Operation **GenerateSequenceDiagram** 834

	Interface IUMLArtifact 998 Interface IUMLCallEvent 1011 Interface IUMLCallOperationAction 1011 Interface IUMLClass 1015 Interface IUMLDataAll 911 Interface IUMLDataType 1039 Interface IUMLGuiSequenceDiagram 1238 Interface IUMLInterface 1094 Interface IUMLMessage 1111 Interface IUMLParameter 1140	Operation InsertOwnedOperationAt 999 Operation Operation 1011 Operation CallOperation 1012 Operation InsertOwnedOperationAt 1016 Operation CallOperation 917 CodeOperation 918 GetOperation 928 InsertOwnedOperationAt 940 Operation 956 SetOperation 970 Operation InsertOwnedOperationAt 1040 Operation CodeOperation 1238 Operation InsertOwnedOperationAt 1095 Operation GetOperation 1111 SetOperation 1112 Operation Operation 1141

Operation **IUMLOperation::Class**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLClass 1015			

Operation **IUMLOperation::Datatype**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLDataType 1039			

Operation **IUMLOperation::Interface**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLInterface 1094			

Operation **IUMLOperation::IsOrdered**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IUMLOperation::IsQuery**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IUMLOperation::IsUnique**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IUMLOperation::Type**

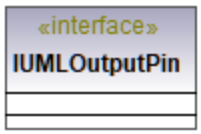
parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLType 1189			

© 2019-2025 Altova GmbH

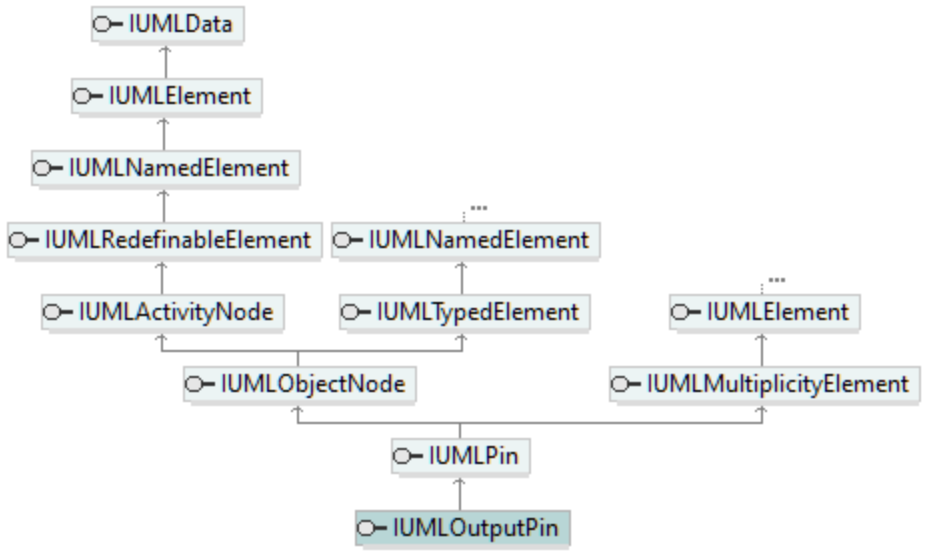
16.4.3.5.129 UModelAPI - IUMLOutputPin

Interface **IUMLOutputPin**

diagram



hierarchy



typedElements

Interface **IUMLAcceptEventAction** 981
Interface **IUMLCallAction** 1008
Interface **IUMLDataAll** 911
Interface **IUMLOpaqueAction** 1128
Interface **IUMLValueSpecificationAction** 1198

Operation **InsertEventActionResultAt** 982
Operation **InsertResultAt** 1009
Operation **InsertEventActionResultAt** 934
InsertOutputValueAt 938
InsertResultAt 943 **Result** 964
Operation **InsertOutputValueAt** 1129
Operation **Result** 1198

© 2019-2025 Altova GmbH

16.4.3.5.130 UModelAPI - IUMLPackage

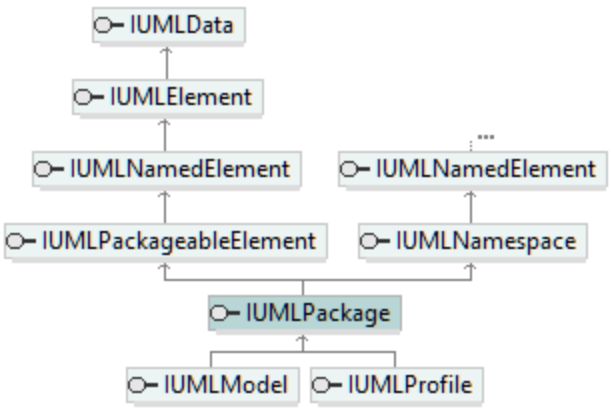
Interface **IUMLPackage**

diagram

«interface»
IUMLPackage

- InsertPackagedElementAt(in nIdx:int, in strKind:string):IUMLPackageableElement
- InsertPackagedElementRelationshipAt(in nIdx:int, in strKind:string, in ipFrom:IUMLElement, in ipTo:IUMLElement):IUMLPackageableElement
- InsertProfileApplicationAt(in nIdx:int, in ipAppliedProfile:IUMLProfile):IUMLProfileApplication
- IsCodeLangNamespace(in nCodeLang:ENUMCodeLang):bool
- IsCodeLangNamespaceRoot(in nCodeLang:ENUMCodeLang):bool
- «GetAccessor, property» PackagedElements():IUMLDataList
- «GetAccessor, property» OwnedTypes():IUMLDataList
- «GetAccessor, property» NestedPackages():IUMLDataList
- «GetAccessor, property» NestingPackage():IUMLPackage
- «GetAccessor, property» ProfileApplications():IUMLDataList
- «GetAccessor, property» IsShared():bool
- «GetAccessor, property» OwnedStereotypes():IUMLDataList
- «GetAccessor, property» URI():string
- «GetAccessor, property» SubprojectFilePath():string

hierarchy



typedElements

Interface **IDocument** 832
Interface **IImportSourceDlg** 858
Interface **IModelTransformationDlg** 881
Interface **IUMLDataAll** 911
Interface **IUMLNamespace** 1120
Interface **IUMLPackage** 1134
Interface **IUMLPackageableElement** 1137
Interface **IUMLPackageImport** 1138

Operation **RootPackage** 837
Operation **ImportTarget** 859
Operation **SourcePackage** 882
TargetPackage 882
Operation **ApplyingPackage** 915
ImportedPackage 930
InsertPackageImportAt 941
InsertPackageMergeAt 941
MergedPackage 953
NestingPackage 955
OwningPackage 960 **Package** 961
ReceivingPackage 963
Operation **InsertPackageImportAt** 1121
InsertPackageMergeAt 1121
Operation **NestingPackage** 1136
Operation **OwningPackage** 1137
Operation **ImportedPackage** 1138

© 2019-2025 Altova GmbH

	Interface **IUMLPackageMerge** 1139	Operation **MergedPackage** 1139
		ReceivingPackage 1139
	Interface **IUMLProfileApplication** 1146	Operation **ApplyingPackage** 1146
	Interface **IUMLType** 1189	Operation **Package** 1190

Operation **IUMLPackage::InsertPackagedElementAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	strKind	**in**	**string**			
	return	**return**	**IUMLPackageable Element** 1137			

Operation **IUMLPackage::InsertPackagedElementRelationshipAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	strKind	**in**	**string**			
	ipFrom	**in**	**IUMLElement** 1050			
	ipTo	**in**	**IUMLElement** 1050			
	return	**return**	**IUMLPackageable Element** 1137			

Operation **IUMLPackage::InsertProfileApplicationAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	ipAppliedProfile	**in**	**IUMLProfile** 1145			
	return	**return**	**IUMLProfileAppli cation** 1146			

Operation **IUMLPackage::IsCodeLangNamespace**

parameter	name	direction	type	type modifier	multiplicity	default
	nCodeLang	**in**	**ENUMCodeLang** 897			
	return	**return**	**bool**			

Operation **IUMLPackage::IsCodeLangNamespaceRoot**

parameter	name	direction	type	type modifier	multiplicity	default
	nCodeLang	**in**	**ENUMCodeLang** 897			
	return	**return**	**bool**			

Operation **IUMLPackage::IsShared**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IUMLPackage::NestedPackages**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			
documentation	A list of elements of type IUMLPackage 1134.					

Operation **IUMLPackage::NestingPackage**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLPackage** 1134			

Operation **IUMLPackage::OwnedStereotypes**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			

Operation **IUMLPackage::OwnedTypes**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			
documentation	A list of elements of type IUMLType 1189.					

Operation **IUMLPackage::PackagedElements**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			
documentation	A list of elements of type IUMLPackageableElement 1137.					

Operation **IUMLPackage::ProfileApplications**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			
documentation	A list of elements of type IUMLProfileApplication 1146.					

Operation **IUMLPackage::SubprojectFilePath**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**string**			

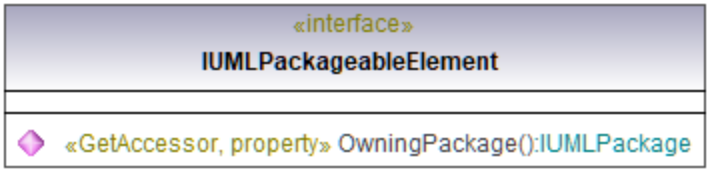
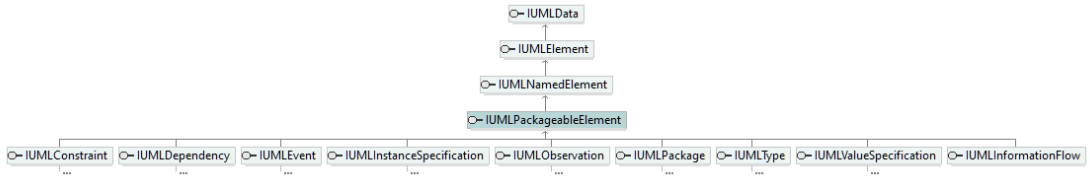
Operation **IUMLPackage::URI**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**string**			

© 2019-2025 Altova GmbH

16.4.3.5.131 UModelAPI - IUMLPackageableElement

Interface **IUMLPackageableElement**

diagram	
hierarchy	
typedElements	Interface **IUMLArtifact** 998 Interface **IUMLDataAll** 911 Interface **IUMLElementImport** 1054 Interface **IUMLManifestation** 1109 Interface **IUMLNamespace** 1120 Interface **IUMLPackage** 1134 Operation **InsertManifestationAt** 999 Operation **ImportedElement** 930 **InsertElementImportAt** 934 **InsertManifestationAt** 937 **InsertPackagedElementAt** 941 **InsertPackagedElementRelationshipAt** 941 **UtilizedElement** 978 Operation **ImportedElement** 1054 Operation **UtilizedElement** 1110 Operation **InsertElementImportAt** 1121 Operation **InsertPackagedElementAt** 1135 **InsertPackagedElementRelationshipAt** 1135

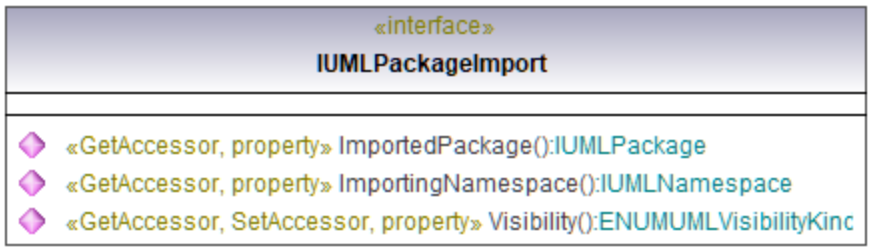
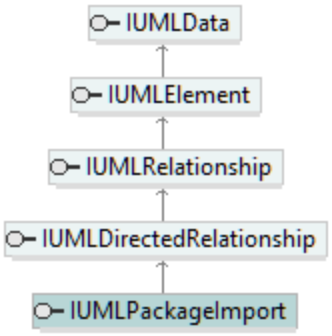
Operation **IUMLPackageableElement::OwningPackage**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLPackage** 1134			

© 2019-2025 Altova GmbH

16.4.3.5.132 UModelAPI - IUMLPackageImport

Interface **IUMLPackageImport**

diagram	
hierarchy	
typedElements	Interface IUMLDataAll 911 Interface IUMLNamespace 1120 Operation InsertPackageImportAt 941 Operation InsertPackageImportAt 1121

Operation **IUMLPackageImport::ImportedPackage**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLPackage 1134			

Operation **IUMLPackageImport::ImportingNamespace**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLNamespace 1120			

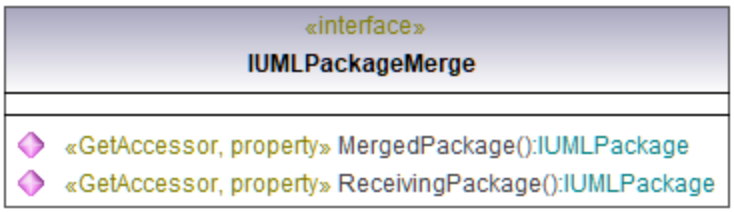
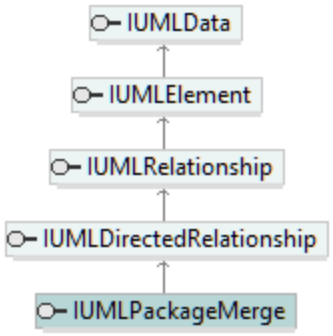
Operation **IUMLPackageImport::Visibility**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	ENUMUMLVisibilityKind 1273			

© 2019-2025 Altova GmbH

16.4.3.5.133 UModelAPI - IUMLPackageMerge

Interface **IUMLPackageMerge**

diagram	
hierarchy	
typedElements	Interface IUMLDataAll 911 Interface IUMLNamespace 1120 Operation InsertPackageMergeAt 941 Operation InsertPackageMergeAt 1121

Operation **IUMLPackageMerge::MergedPackage**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLPackage 1134			

Operation **IUMLPackageMerge::ReceivingPackage**

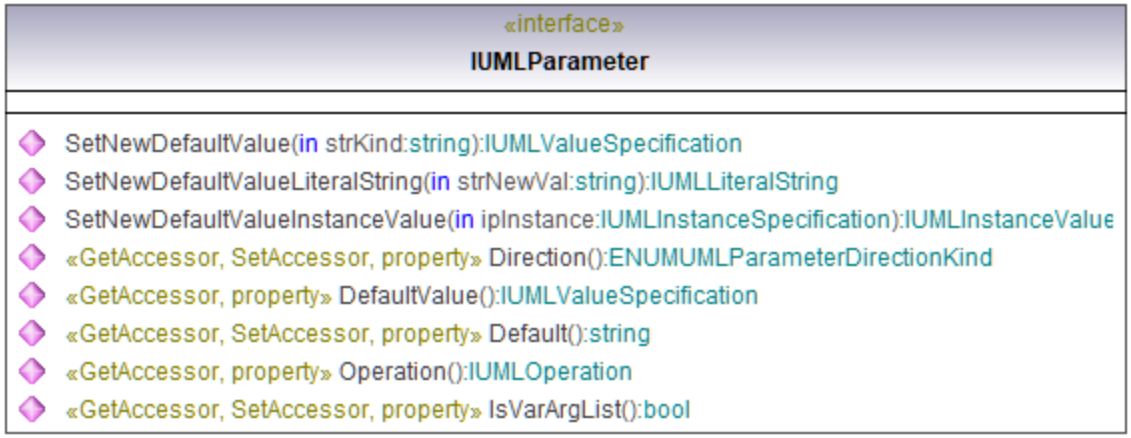
parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLPackage 1134			

© 2019-2025 Altova GmbH

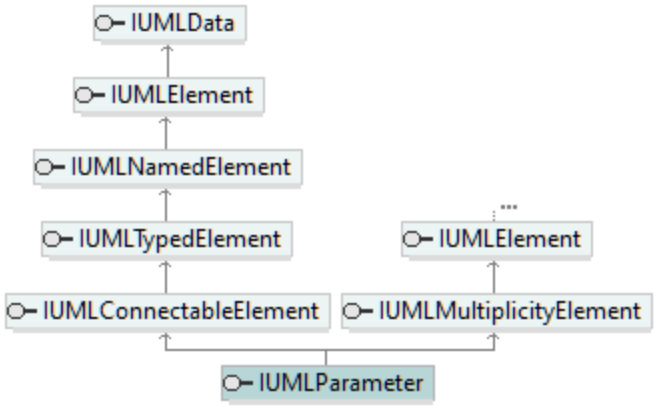
16.4.3.5.134 UModelAPI - IUMLParameter

Interface **IUMLParameter**

diagram



hierarchy



typed Elements

Interface IUMLActivityParameterNode 994
Interface IUMLBehavior 1002
Interface IUMLBehavioralFeature 1004
Interface IUMLDataAll 911
Interface IUMLValueSpecification 1195

Operation Parameter 994
Operation InsertOwnedParameterAt 1003
Operation InsertOwnedParameterAt 1005
Operation InsertOwnedParameterAt 940
OwningParameter 960 Parameter 961
Operation OwningParameter 1197

Operation **IUMLParameter::Default**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**string**			

Operation **IUMLParameter::DefaultValue**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLValueSpecification 1195			

© 2019-2025 Altova GmbH

Operation **IUMLParameter::Direction**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**ENUMUMLParameterDirectionKind** 1271			

Operation **IUMLParameter::IsVarArgList**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IUMLParameter::Operation**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLOperation** 1131			

Operation **IUMLParameter::SetNewDefaultValue**

parameter	name	direction	type	type modifier	multiplicity	default
	strKind	**in**	**string**			
	return	**return**	**IUMLValueSpecification** 1195			

Operation **IUMLParameter::SetNewDefaultValueInstanceValue**

parameter	name	direction	type	type modifier	multiplicity	default
	ipInstance	**in**	**IUMLInstanceSpecification** 1085			
	return	**return**	**IUMLInstanceValue** 1087			

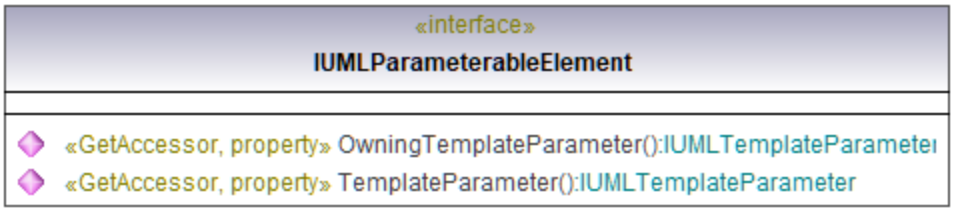
Operation **IUMLParameter::SetNewDefaultValueLiteralString**

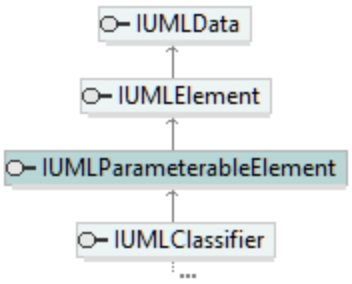
parameter	name	direction	type	type modifier	multiplicity	default
	strNewVal	**in**	**string**			
	return	**return**	**IUMLLiteralString** 1108			

16.4.3.5.135 UModelAPI - IUMLParameterableElement

Interface **IUMLParameterableElement**

diagram



hierarchy	
typedElements	Interface IUMLDataAll 911 Interface IUMLTemplateBinding 1177 Interface IUMLTemplateParameter 1178 Interface IUMLTemplateParameterSubstitution 1180 Operation Actual 912, InsertParameterSubstitutionAt 942, OwnedActual 957, OwnedParameteredElement 958, ParameteredElement 961, SetNewOwnedParameteredElement 968 Operation InsertParameterSubstitutionAt 1177 Operation OwnedParameteredElement 1179, ParameteredElement 1179, SetNewOwnedParameteredElement 1179 Operation Actual 1180, OwnedActual 1180

Operation **IUMLParameterableElement::OwningTemplateParameter**

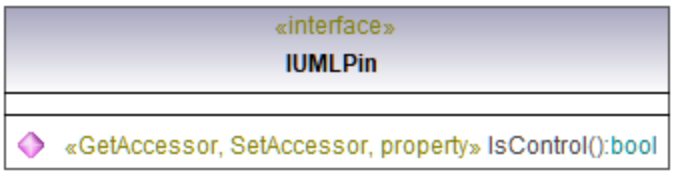
parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLTemplateParameter 1178			

Operation **IUMLParameterableElement::TemplateParameter**

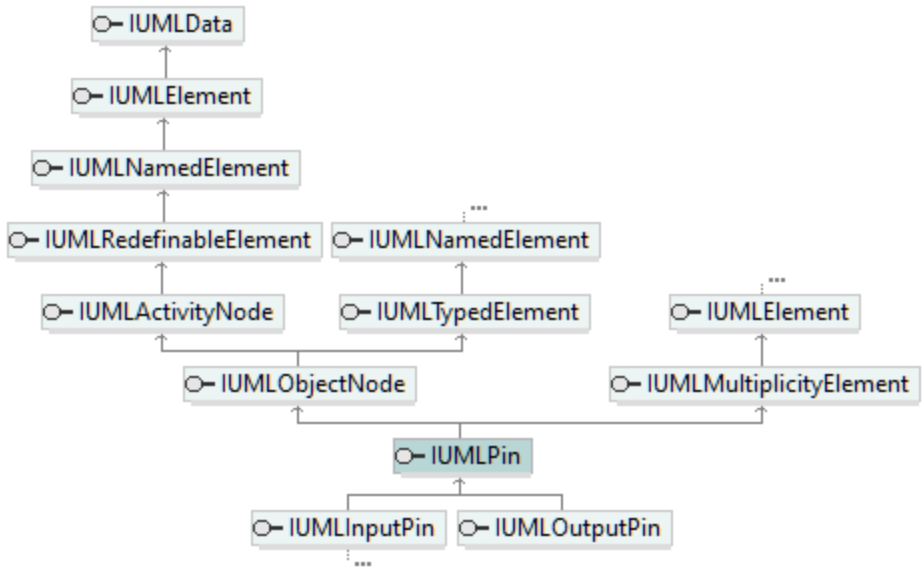
parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLTemplateParameter 1178			

16.4.3.5.136 UModelAPI - IUMLPin

Interface **IUMLPin**

diagram	

 © 2019-2025 Altova GmbH

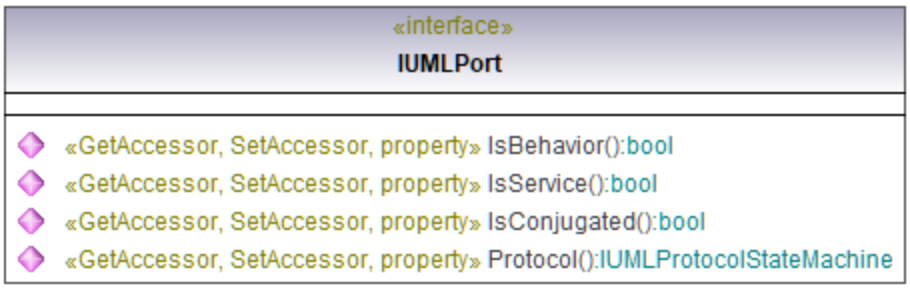
hierarchy	

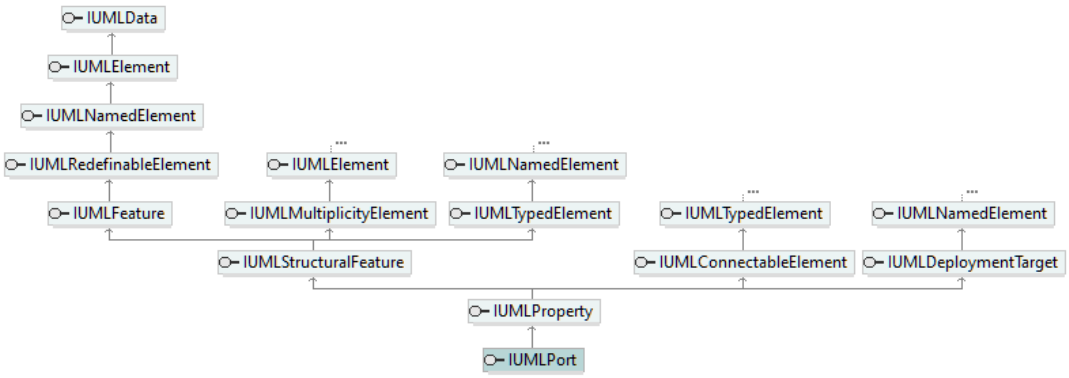
Operation **IUMLPin::IsControl**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

16.4.3.5.137 UModelAPI - IUMLPort

Interface **IUMLPort**

diagram	

hierarchy	
typedElements	Interface IUMLDataAll 911 Interface IUMLStructuredClassifier 1174 Operation InsertOwnedPortAt 940 Operation InsertOwnedPortAt 1175

Operation **IUMLPort::IsBehavior**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IUMLPort::IsConjugated**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IUMLPort::IsService**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

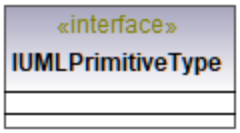
Operation **IUMLPort::Protocol**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLProtocolStateMachine 1150			

UML documentation generated by **UModel** UML Editor **http://www.altova.com/umodel** Wed Sep 25 10:33:04 2024

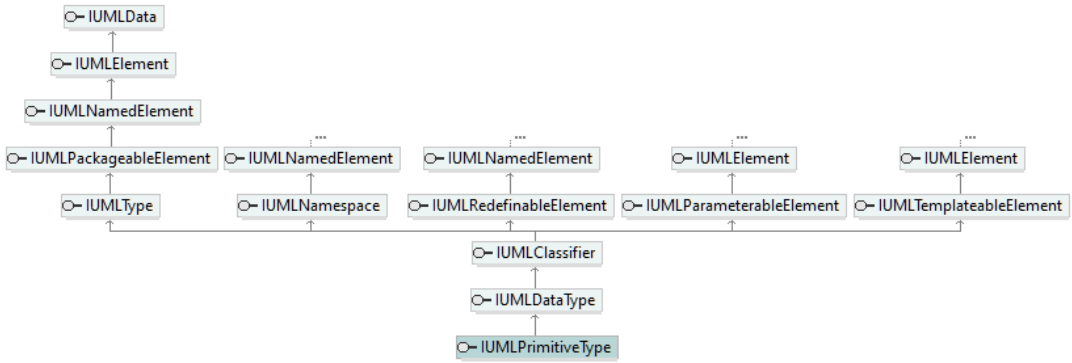
16.4.3.5.138 UModelAPI - IUMLPrimitiveType

Interface **IUMLPrimitiveType**

diagram	

© 2019-2025 Altova GmbH

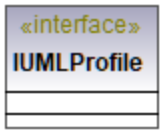
hierarchy



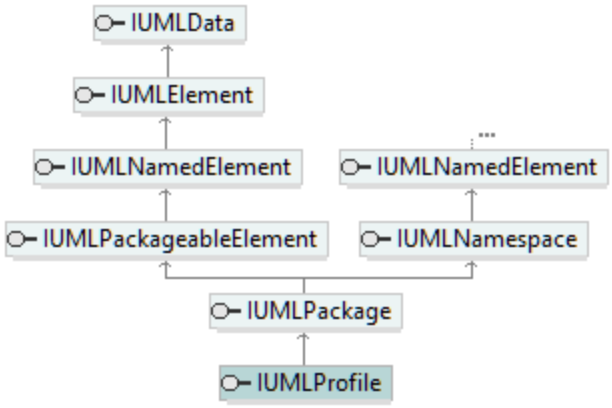
16.4.3.5.139 UModelAPI - IUMLProfile

Interface **IUMLProfile**

diagram



hierarchy



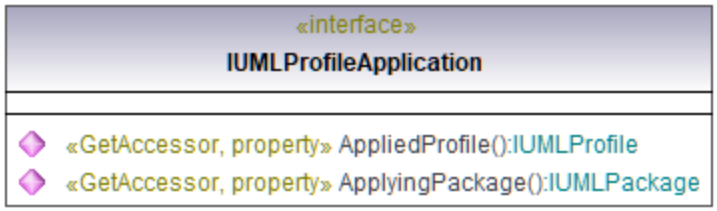
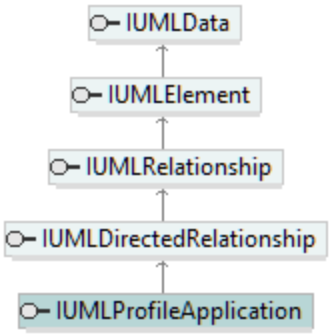
typedElements

Interface **IUMLDataAll** 911
Interface **IUMLPackage** 1134
Interface **IUMLProfileApplication** 1146
Operation **AppliedProfile** 915
InsertProfileApplicationAt 942
Operation **InsertProfileApplicationAt** 1135
Operation **AppliedProfile** 1146

UML documentation generated by UModel UML Editor http://www.altova.com/umodel Wed Sep 25 10:33:04 2024

16.4.3.5.140 UModelAPI - IUMLProfileApplication

Interface **IUMLProfileApplication**

diagram	
hierarchy	
typedElements	Interface IUMLDataAll 911 Interface IUMLPackage 1134 Operation InsertProfileApplicationAt 942 Operation InsertProfileApplicationAt 1135

Operation **IUMLProfileApplication::AppliedProfile**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLProfile 1145			

Operation **IUMLProfileApplication::ApplyingPackage**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLPackage 1134			

UML documentation generated by UModel UML Editor http://www.altova.com/umodel Wed Sep 25 10:33:04 2024

 © 2019-2025 Altova GmbH

16.4.3.5.141 UModelAPI - IUMLProperty

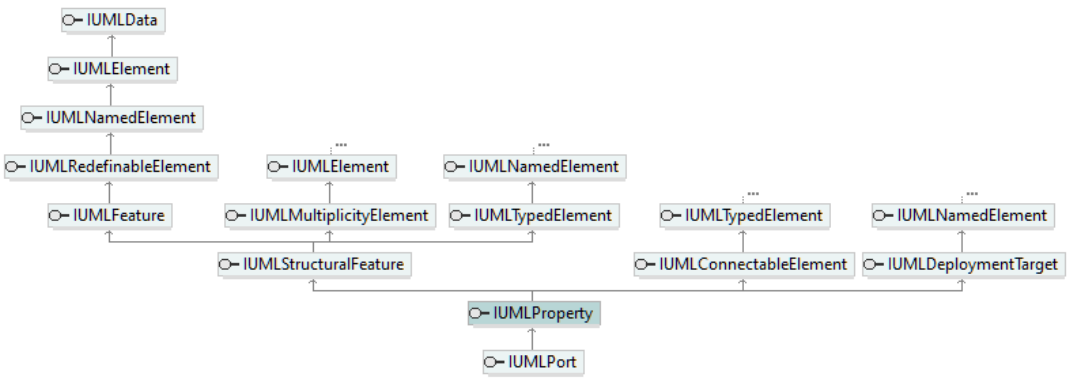
Interface **IUMLProperty**

diagram

«interface»
IUMLProperty

- InsertQualifierAt(in nIdx:int):IUMLProperty
- SetNewDefaultValue(in strKind:string):IUMLValueSpecification
- SetNewDefaultValueLiteralString(in strNewVal:string):IUMLLiteralString
- SetNewDefaultValueInstanceValue(in ipInstance:IUMLInstanceSpecification):IUMLInstanceValue
- «GetAccessor, SetAccessor, property» IsDerived():bool
- «GetAccessor, SetAccessor, property» IsDerivedUnion():bool
- «GetAccessor, property» IsComposite():bool
- «GetAccessor, property» Opposite():IUMLProperty
- «GetAccessor, property» Association():IUMLAssociation
- «GetAccessor, property» OwningAssociation():IUMLAssociation
- «GetAccessor, SetAccessor, property» IsNavigable():bool
- «GetAccessor, SetAccessor, property» IsOwnedEnd():bool
- «GetAccessor, property» AssociationEnd():IUMLProperty
- «GetAccessor, property» Qualifiers():IUMLDataList
- «GetAccessor, SetAccessor, property» Aggregation():ENUMUMLAggregationKind
- «GetAccessor, property» OwningSignal():IUMLSignal
- «GetAccessor, property» Datatype():IUMLDataType
- «GetAccessor, property» Class():IUMLClass
- «GetAccessor, property» Interface():IUMLInterface
- «GetAccessor, property» DefaultValue():IUMLValueSpecification
- «GetAccessor, SetAccessor, property» Default():string
- «GetAccessor, property» Classifier():IUMLClassifier

hierarchy



typedElements

Interface IUMLArtifact 998
Interface IUMLDataAll 911

Operation InsertOwnedAttributeAt 999
Operation AssociationEnd 915
InsertOwnedAttributeAt 938

	Interface IUMLDataType 1039 Interface IUMLInterface 1094 Interface IUMLProperty 1147 Interface IUMLSignal 1160 Interface IUMLStructuredClassifier 1174 Interface IUMLValueSpecification 1195	InsertQualifierAt 942 Opposite 956 OwningProperty 960 Operation InsertOwnedAttributeAt 1040 Operation InsertOwnedAttributeAt 1095 Operation AssociationEnd 1148 InsertQualifierAt 1149 Opposite 1149 Operation InsertOwnedAttributeAt 1160 Operation InsertOwnedAttributeAt 1175 Operation OwningProperty 1197

Operation **IUMLProperty::Aggregation**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLAggregationKind 1264			

Operation **IUMLProperty::Association**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLAssociation 1000			

Operation **IUMLProperty::AssociationEnd**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLProperty 1147			

Operation **IUMLProperty::Class**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLClass 1015			

Operation **IUMLProperty::Classifier**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLClassifier 1018			

Operation **IUMLProperty::Datatype**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLDataType 1039			

Operation **IUMLProperty::Default**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**string**			

Operation **IUMLProperty::DefaultValue**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLValueSpecification 1195			

© 2019-2025 Altova GmbH

Operation **IUMLProperty::InsertQualifierAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	**IUMLProperty** 1147			

Operation **IUMLProperty::Interface**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLInterface** 1094			

Operation **IUMLProperty::IsComposite**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IUMLProperty::IsDerived**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IUMLProperty::IsDerivedUnion**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IUMLProperty::IsNavigable**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IUMLProperty::IsOwnedEnd**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IUMLProperty::Opposite**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLProperty** 1147			

Operation **IUMLProperty::OwningAssociation**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLAssociation** 1000			

Operation **IUMLProperty::OwningSignal**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLSignal** 1160			

Operation **IUMLProperty::Qualifiers**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			
documentation	A list of elements of type IUMLProperty 1147.					

Operation **IUMLProperty::SetNewDefaultValue**

parameter	name	direction	type	type modifier	multiplicity	default
	strKind	**in**	**string**			
	return	**return**	**IUMLValueSpecification** 1195			

Operation **IUMLProperty::SetNewDefaultValueInstanceValue**

parameter	name	direction	type	type modifier	multiplicity	default
	ipInstance	**in**	**IUMLInstanceSpecification** 1085			
	return	**return**	**IUMLInstanceValue** 1087			

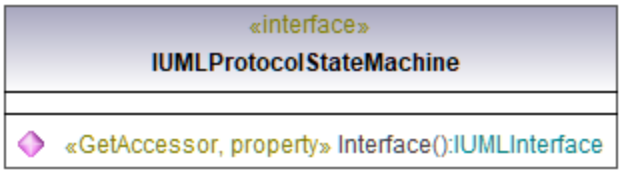
Operation **IUMLProperty::SetNewDefaultValueLiteralString**

parameter	name	direction	type	type modifier	multiplicity	default
	strNewVal	**in**	**string**			
	return	**return**	**IUMLLiteralString** 1108			

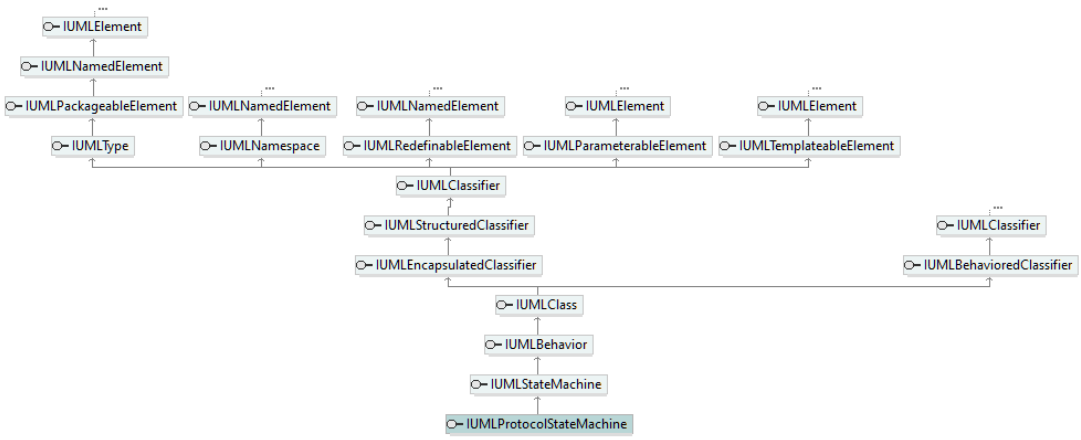
16.4.3.5.142 UModelAPI - IUMLProtocolStateMachine

Interface **IUMLProtocolStateMachine**

diagram	



 © 2019-2025 Altova GmbH

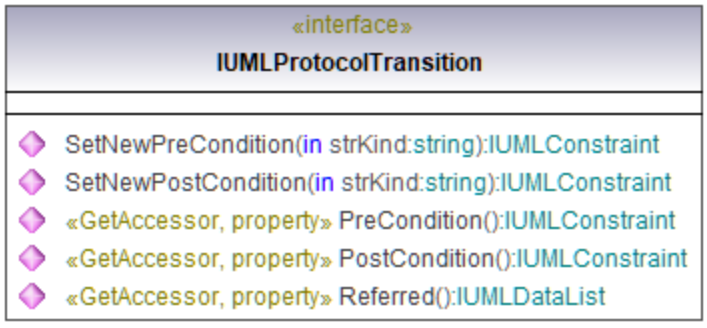
hierarchy	
typedElements	Interface **IUMLDataAll** 911 Interface **IUMLInterface** 1094 Interface **IUMLPort** 1143 Operation **Protocol** 962 **SetNewProtocol** 969 Operation **Protocol** 1096 **SetNewProtocol** 1096 Operation **Protocol** 1144

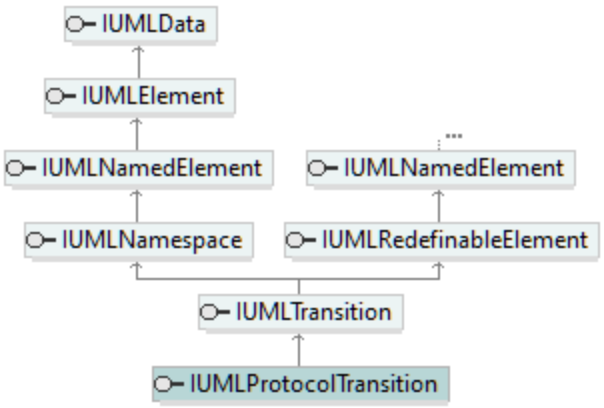
Operation **IUMLProtocolStateMachine::Interface**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLInterface** 1094			

16.4.3.5.143 UModelAPI - IUMLProtocolTransition

Interface **IUMLProtocolTransition**

diagram	

hierarchy	
typedElements	Interface IUMLConstraint 1035 Interface IUMLDataAll 911 Operation OwningTransition 1036 Operation OwningTransition 960

Operation **IUMLProtocolTransition::PostCondition**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLConstraint 1035			

Operation **IUMLProtocolTransition::PreCondition**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLConstraint 1035			

Operation **IUMLProtocolTransition::Referred**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLDataList 906			

Operation **IUMLProtocolTransition::SetNewPostCondition**

parameter	name	direction	type	type modifier	multiplicity	default
	strKind	**in**	**string**			
	return	**return**	IUMLConstraint 1035			

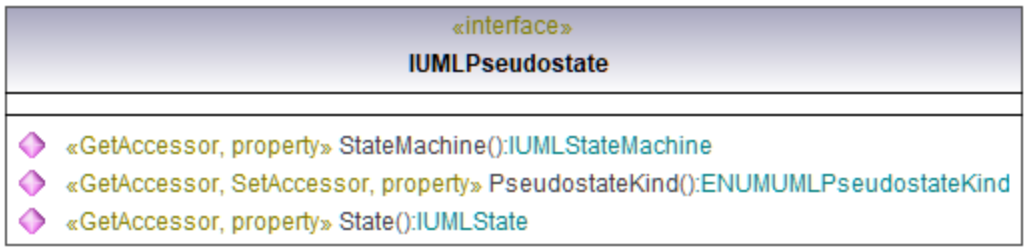
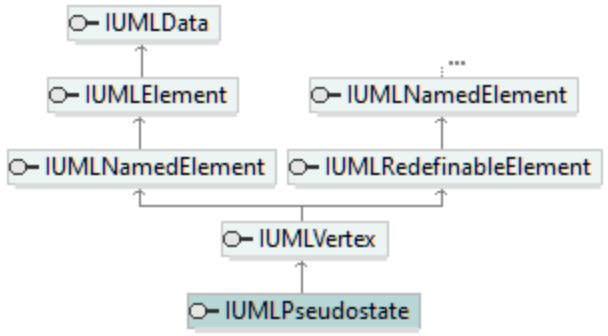
Operation **IUMLProtocolTransition::SetNewPreCondition**

parameter	name	direction	type	type modifier	multiplicity	default
	strKind	**in**	**string**			
	return	**return**	IUMLConstraint 1035			

© 2019-2025 Altova GmbH

16.4.3.5.144 UModelAPI - IUMLPseudostate

Interface **IUMLPseudostate**

diagram	
hierarchy	
typedElements	Interface IUMLConnectionPointReference 1031 Interface IUMLDataAll 911 Interface IUMLState 1163 Interface IUMLStateMachine 1167 Operation InsertEntryAt 1032 InsertExitAt 1032 Operation InsertConnectionPointAt 933 InsertEntryAt 934 InsertExitAt 934 Operation InsertConnectionPointAt 1165 Operation InsertConnectionPointAt 1168

Operation **IUMLPseudostate::PseudostateKind**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	ENUMUMLPseudostateKind 1272			

Operation **IUMLPseudostate::State**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLState 1163			

Operation **IUMLPseudostate::StateMachine**

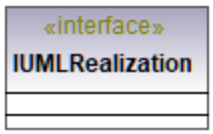
parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLStateMachine 1167			

© 2019-2025 Altova GmbH

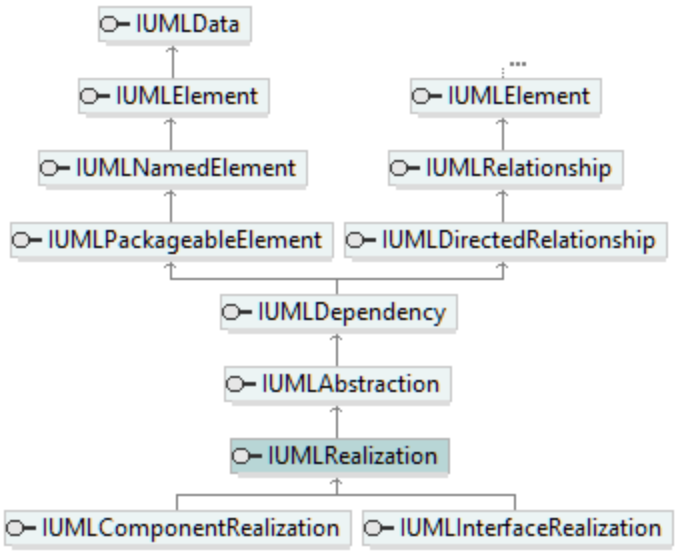
16.4.3.5.145 UModelAPI - IUMLRealization

Interface **IUMLRealization**

diagram



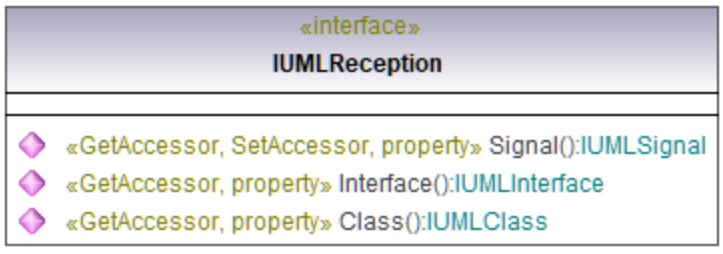
hierarchy



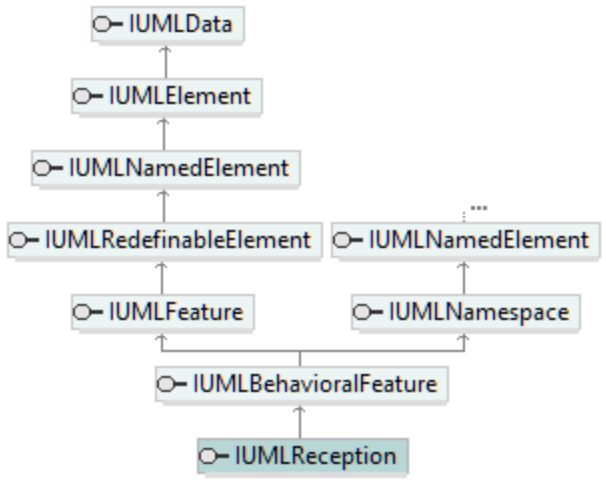
16.4.3.5.146 UModelAPI - IUMLReception

Interface **IUMLReception**

diagram



 © 2019-2025 Altova GmbH

hierarchy	
typedElements	Interface **IUMLClass** 1015 Interface **IUMLDataAll** 911 Interface **IUMLInterface** 1094 Operation **InsertOwnedReceptionAt** 1016 Operation **InsertOwnedReceptionAt** 940 Operation **InsertOwnedReceptionAt** 1095

Operation **IUMLReception::Class**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLClass** 1015			

Operation **IUMLReception::Interface**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLInterface** 1094			

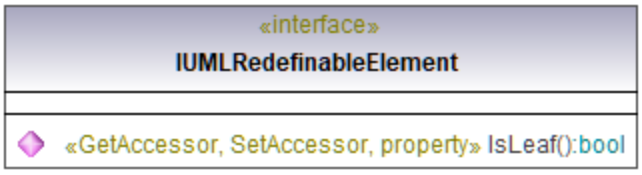
Operation **IUMLReception::Signal**

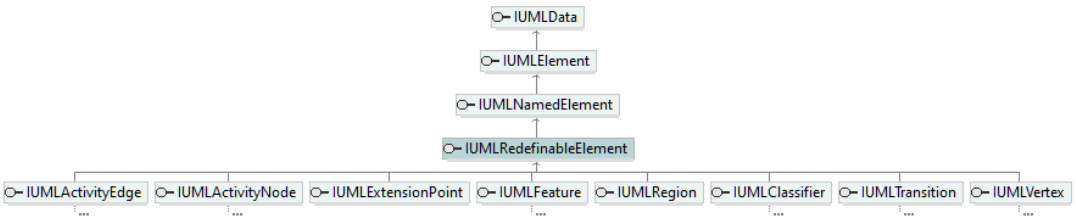
parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLSignal** 1160			

UML documentation generated by **UModel** UML Editor **http://www.altova.com/umodel** Wed Sep 25 10:33:04 2024

16.4.3.5.147 UModelAPI - IUMLRedefinableElement

Interface **IUMLRedefinableElement**

diagram	

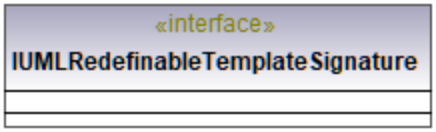
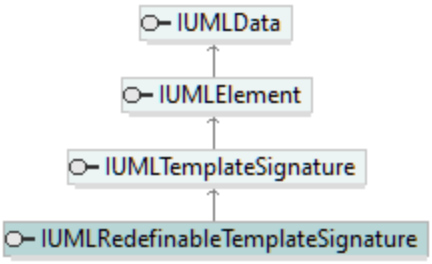
hierarchy	

Operation **IUMLRedefinableElement::IsLeaf**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

16.4.3.5.148 UModelAPI - IUMLRedefinableTemplateSignature

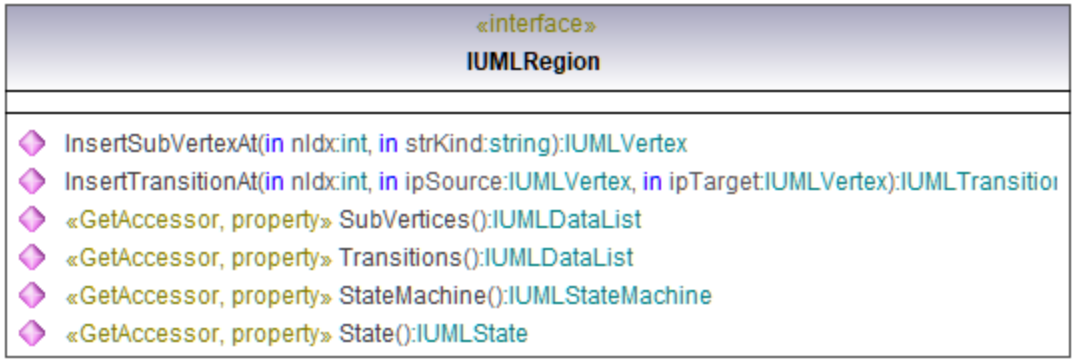
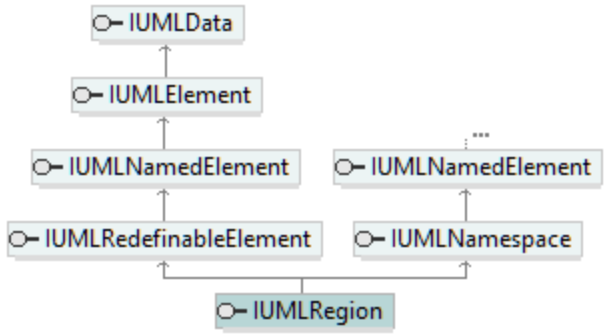
Interface **IUMLRedefinableTemplateSignature**

diagram	
hierarchy	

 © 2019-2025 Altova GmbH

16.4.3.5.149 UModelAPI - IUMLRegion

Interface **IUMLRegion**

diagram	
hierarchy	
typedElements	Interface IUMLDataAll 911 Interface IUMLState 1163 Interface IUMLStateMachine 1167 Interface IUMLVertex 1199 Operation Container 919 InsertRegionAt 943 Operation InsertRegionAt 1165 Operation InsertRegionAt 1168 Operation Container 1199

Operation **IUMLRegion::InsertSubVertexAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	strKind	**in**	**string**			
	return	**return**	**IUMLVertex** 1199			

Operation **IUMLRegion::InsertTransitionAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	ipSource	**in**	**IUMLVertex** 1199			
	ipTarget	**in**	**IUMLVertex** 1199			
	return	**return**	**IUMLTransition** 1187			

Operation **IUMLRegion::State**

parameter	name	direction	type	type modifier	multiplicity	default

	return	**return**	**IUMLState** 1163

Operation **IUMLRegion::StateMachine**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLStateMachine** 1167			

Operation **IUMLRegion::SubVertices**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			
documentation	A list of elements of type IUMLVertex 1199.					

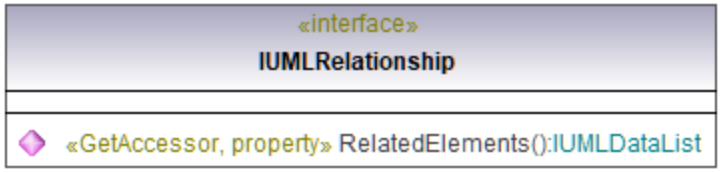
Operation **IUMLRegion::Transitions**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			
documentation	A list of elements of type IUMLTransition 1187.					

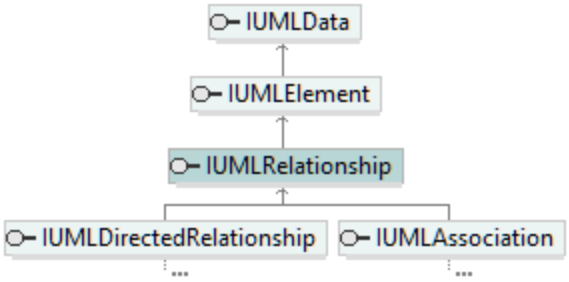
16.4.3.5.150 UModelAPI - IUMLRelationship

Interface **IUMLRelationship**

diagram



hierarchy



typedElements

Interface **IUMLDataAll** 911 Operation **InsertInformationFlowRealizationAt** 935

© 2019-2025 Altova GmbH

	Interface IUMLInformationFlow 1080	Operation InsertInformationFlowRealizationAt 1081

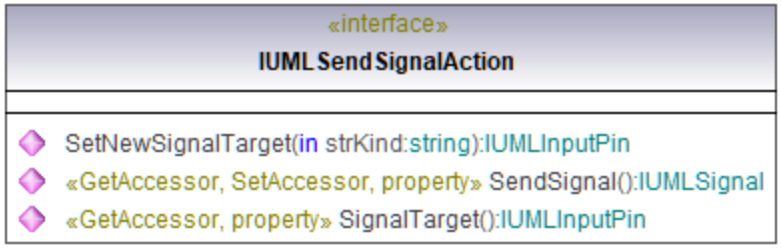
Operation **IUMLRelationship::RelatedElements**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLDataList 906			
documentation	A list of elements of type IUMLElement 1050.					

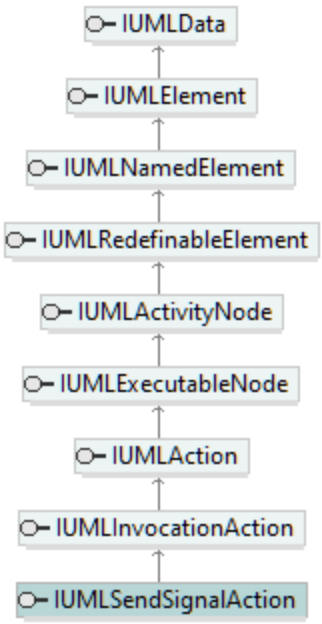
16.4.3.5.151 UModelAPI - IUMLSendSignalAction

Interface **IUMLSendSignalAction**

diagram



hierarchy



Operation **IUMLSendSignalAction::SendSignal**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLSignal** 1160			

Operation **IUMLSendSignalAction::SetNewSignalTarget**

parameter	name	direction	type	type modifier	multiplicity	default
	strKind	**in**	**string**			
	return	**return**	**IUMLInputPin** 1083			

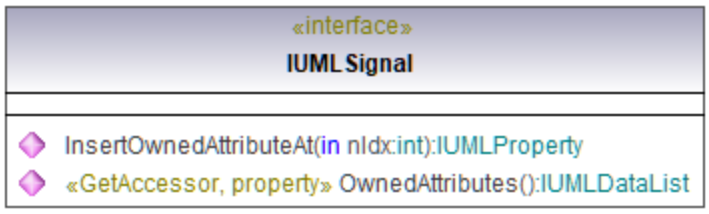
Operation **IUMLSendSignalAction::SignalTarget**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLInputPin** 1083			

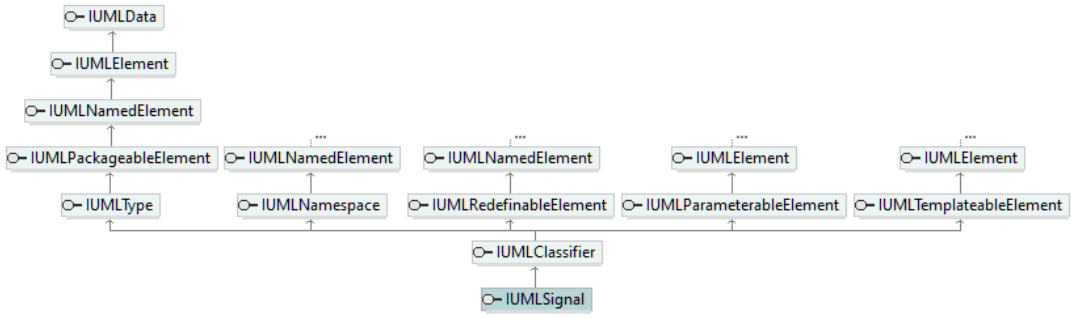
16.4.3.5.152 UModelAPI - IUMLSignal

Interface **IUMLSignal**

diagram



hierarchy



typedElements

Interface **IUMLDataAll** 911

Interface **IUMLProperty** 1147
Interface **IUMLReception** 1154
Interface **IUMLSendSignalAction** 1159
Interface **IUMLSignalEvent** 1161

Operation **OwningSignal** 960 **SendSignal** 965 **Signal** 972
Operation **OwningSignal** 1149
Operation **Signal** 1155
Operation **SendSignal** 1159
Operation **Signal** 1161

Operation **IUMLSignal::InsertOwnedAttributeAt**

 © 2019-2025 Altova GmbH

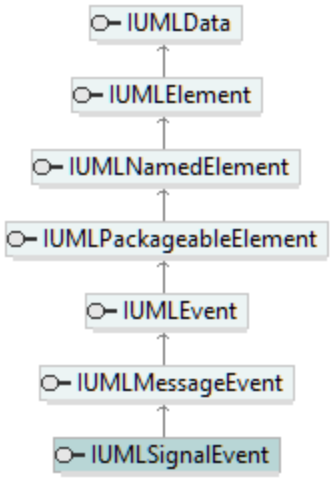
parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	**IUMLProperty** 1147			

Operation **IUMLSignal::OwnedAttributes**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			
documentation	A list of elements of type IUMLProperty 1147.					

16.4.3.5.153 UModelAPI - IUMLSignalEvent

Interface **IUMLSignalEvent**

diagram	
hierarchy	

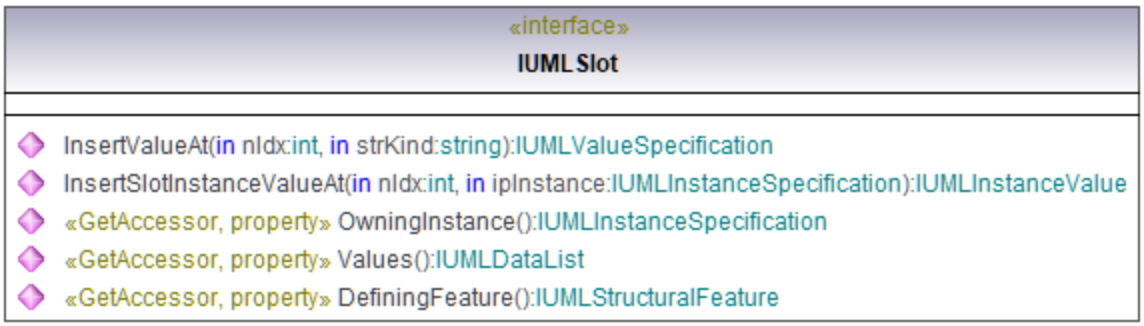
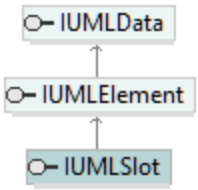
Operation **IUMLSignalEvent::Signal**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLSignal** 1160			

UML documentation generated by UModel UML Editor http://www.altova.com/umodel Wed Sep 25 10:33:04 2024

16.4.3.5.154 UModelAPI - IUMLSlot

Interface **IUMLSlot**

diagram	
hierarchy	
typedElements	Interface IUMLDataAll 911 Interface IUMLInstanceSpecification 1085 Interface IUMLValueSpecification 1195 Operation InsertSlotAt 943 OwningSlot 960 Operation InsertSlotAt 1086 Operation OwningSlot 1197

Operation **IUMLSlot::DefiningFeature**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLStructuralFeature 1171			

Operation **IUMLSlot::InsertSlotInstanceValueAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	ipInstance	**in**	IUMLInstanceSpecification 1085			
	return	**return**	IUMLInstanceValue 1087			

Operation **IUMLSlot::InsertValueAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	strKind	**in**	**string**			
	return	**return**	IUMLValueSpecification 1195			

 © 2019-2025 Altova GmbH

Operation **IUMLSlot::OwningInstance**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLInstanceSpecification** 1085			

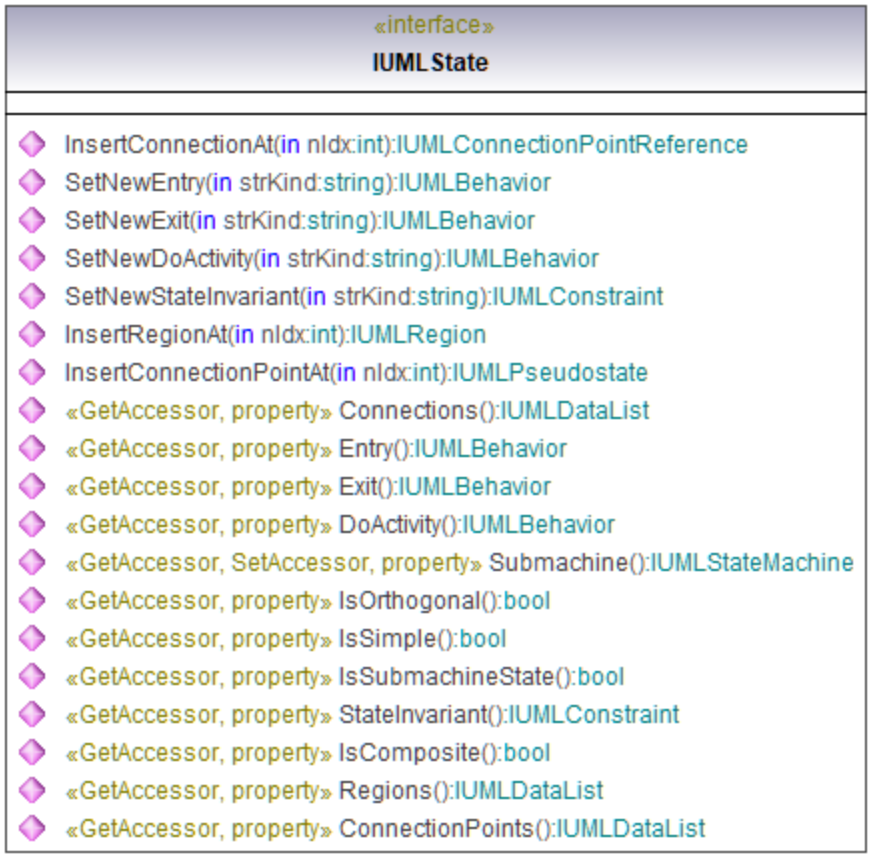
Operation **IUMLSlot::Values**

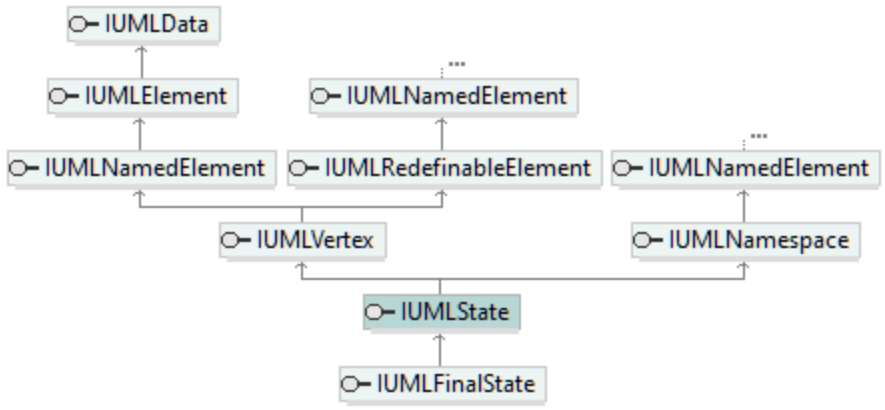
parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			
documentation	A list of elements of type IUMLValueSpecification 1195.					

16.4.3.5.155 UModelAPI - IUMLState

Interface **IUMLState**

diagram



hierarchy	
typedElements	Interface IUMLConnectionPointReference 1031 Interface IUMLConstraint 1035 Interface IUMLDataAll 911 Interface IUMLObjectNode 1124 Interface IUMLPseudostate 1153 Interface IUMLRegion 1157 Operation State 1033 Operation OwningState 1036 Operation InsertInStateAt 936 OwningState 960 State 973 Operation InsertInStateAt 1125 Operation State 1153 Operation State 1157

Operation **IUMLState::ConnectionPoints**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLDataList 906			
documentation	A list of elements of type IUMLPseudostate 1153.					

Operation **IUMLState::Connections**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLDataList 906			
documentation	A list of elements of type IUMLConnectionPointReference 1031.					

Operation **IUMLState::DoActivity**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLBehavior 1002			

Operation **IUMLState::Entry**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLBehavior 1002			

Operation **IUMLState::Exit**

parameter	name	direction	type	type modifier	multiplicity	default

© 2019-2025 Altova GmbH

	return	**return**	**IUMLBehavior** 1002

Operation **IUMLState::InsertConnectionAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	**IUMLConnectionPointReference** 1031			

Operation **IUMLState::InsertConnectionPointAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	**IUMLPseudostate** 1153			

Operation **IUMLState::InsertRegionAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	**IUMLRegion** 1157			

Operation **IUMLState::IsComposite**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IUMLState::IsOrthogonal**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IUMLState::IsSimple**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IUMLState::IsSubmachineState**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IUMLState::Regions**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			
documentation	A list of elements of type IUMLRegion 1157.					

Operation **IUMLState::SetNewDoActivity**

parameter	name	direction	type	type modifier	multiplicity	default
	strKind	**in**	**string**			
	return	**return**	**IUMLBehavior** 1002			

Operation **IUMLState::SetNewEntry**

parameter	name	direction	type	type modifier	multiplicity	default
	strKind	**in**	**string**			
	return	**return**	**IUMLBehavior** 1002			

Operation **IUMLState::SetNewExit**

parameter	name	direction	type	type modifier	multiplicity	default
	strKind	**in**	**string**			
	return	**return**	**IUMLBehavior** 1002			

Operation **IUMLState::SetNewStateInvariant**

parameter	name	direction	type	type modifier	multiplicity	default
	strKind	**in**	**string**			
	return	**return**	**IUMLConstraint** 1035			

Operation **IUMLState::StateInvariant**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLConstraint** 1035			

Operation **IUMLState::Submachine**

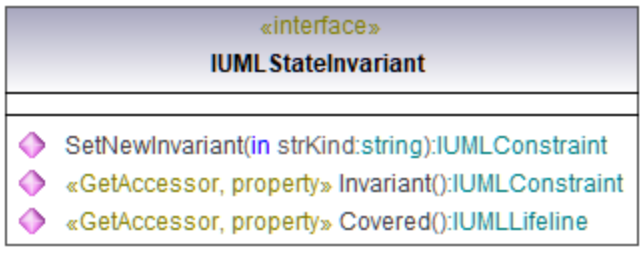
parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLStateMachine** 1167			

UML documentation generated by **UModel** UML Editor **http://www.altova.com/umodel** Wed Sep 25 10:33:04 2024

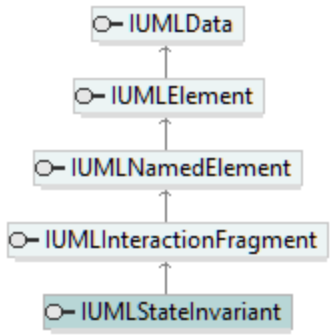
16.4.3.5.156 UModelAPI - IUMLStateInvariant

Interface **IUMLStateInvariant**

diagram



 © 2019-2025 Altova GmbH

hierarchy	

Operation **IUMLStateInvariant::Covered**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLLifeline** 1104			

Operation **IUMLStateInvariant::Invariant**

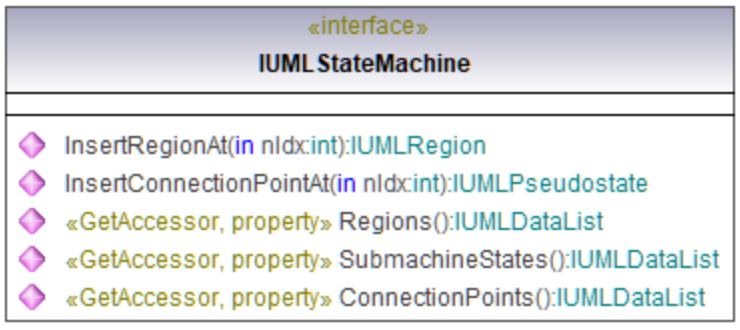
parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLConstraint** 1035			

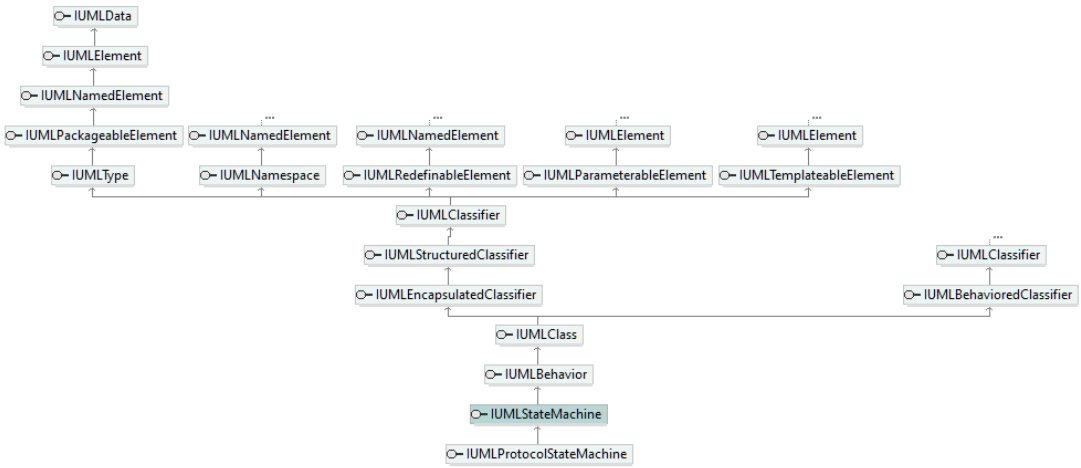
Operation **IUMLStateInvariant::SetNewInvariant**

parameter	name	direction	type	type modifier	multiplicity	default
	strKind	**in**	**string**			
	return	**return**	**IUMLConstraint** 1035			

16.4.3.5.157 UModelAPI - IUMLStateMachine

Interface **IUMLStateMachine**

diagram	

hierarchy	
typedElements	Interface **IDocument** 832 Interface **IUMLDataAll** 911 Interface **IUMLPseudostate** 1153 Interface **IUMLRegion** 1157 Interface **IUMLState** 1163 Operation **GenerateStateMachineCode** 835 Operation **StateMachine** 973 **Submachine** 974 Operation **StateMachine** 1153 Operation **StateMachine** 1158 Operation **Submachine** 1166

Operation **IUMLStateMachine::ConnectionPoints**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			
documentation	A list of elements of type IUMLPseudostate 1153.					

Operation **IUMLStateMachine::InsertConnectionPointAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	**IUMLPseudostate** 1153			

Operation **IUMLStateMachine::InsertRegionAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	**IUMLRegion** 1157			

Operation **IUMLStateMachine::Regions**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			
documentation	A list of elements of type IUMLRegion 1157.					

Operation **IUMLStateMachine::SubmachineStates**

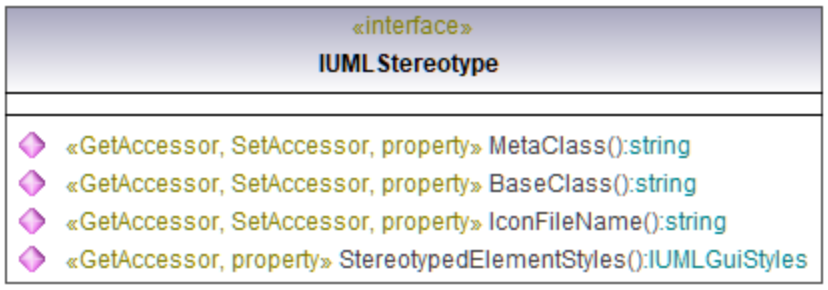
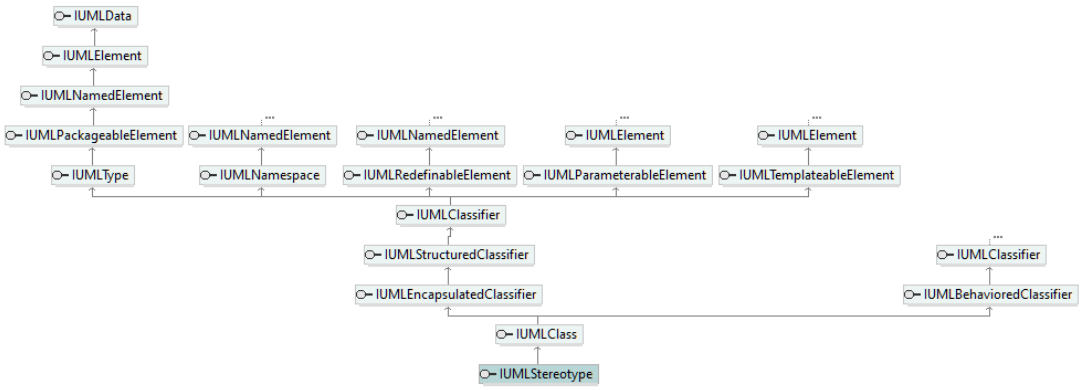
parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			

© 2019-2025 Altova GmbH

documentation	A list of elements of type IUMLState[1163].

16.4.3.5.158 UModelAPI - IUMLStereotype

Interface **IUMLStereotype**

diagram	
hierarchy	
typedElements	Interface **IUMLDataAll**[911] Interface **IUMLElement**[1050] Interface **IUMLStereotypeApplication**[1170] Operation **ApplyStereotype**[915] **GetStereotypeApplicationForStereotype**[928] **IsStereotypeApplied**[949] **Stereotype**[974] **UnapplyStereotype**[977] Operation **ApplyStereotype**[1051] **GetStereotypeApplicationForStereotype**[1052] **IsStereotypeApplied**[1052] **UnapplyStereotype**[1053] Operation **Stereotype**[1171]

Operation **IUMLStereotype::BaseClass**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**string**			

Operation **IUMLStereotype::IconFileName**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**string**			

Operation **IUMLStereotype::MetaClass**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**string**			

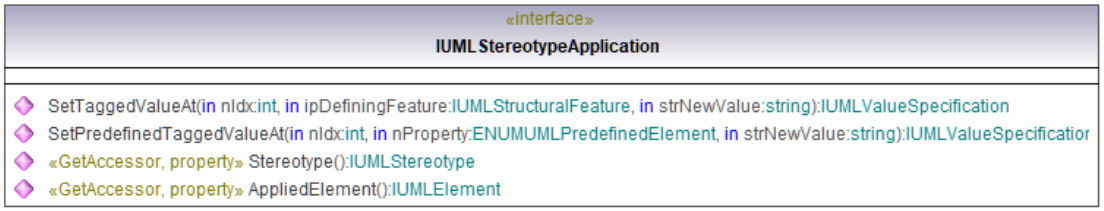
Operation **IUMLStereotype::StereotypedElementStyles**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLGuiStyles 1241			

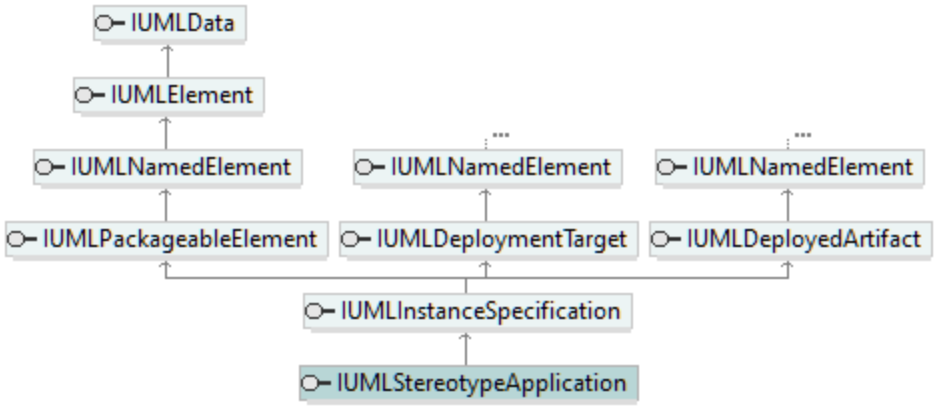
16.4.3.5.159 UModelAPI - IUMLStereotypeApplication

Interface **IUMLStereotypeApplication**

diagram



hierarchy



typedElements

Interface IUMLDataAll 911

Operation ApplyPredefinedStereotype 915
ApplyStereotype 915
GetStereotypeApplicationForPredefinedStereotype 928
GetStereotypeApplicationForStereotype 928

© 2019-2025 Altova GmbH

	Interface **IUMLElement**[1050]	Operation **ApplyPredefinedStereotype**[1051] **ApplyStereotype**[1051] **GetStereotypeApplicationForPredefinedStereotype**[1052] **GetStereotypeApplicationForStereotype**[1052]

Operation **IUMLStereotypeApplication::AppliedElement**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLElement**[1050]			

Operation **IUMLStereotypeApplication::SetPredefinedTaggedValueAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	nProperty	**in**	**ENUMUMLPredefinedElement**[1271]			
	strNewValue	**in**	**string**			
	return	**return**	**IUMLValueSpecification**[1195]			

Operation **IUMLStereotypeApplication::SetTaggedValueAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	ipDefiningFeature	**in**	**IUMLStructuralFeature**[1171]			
	strNewValue	**in**	**string**			
	return	**return**	**IUMLValueSpecification**[1195]			

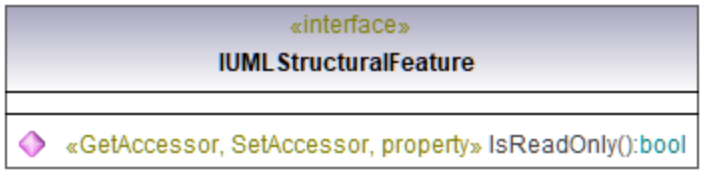
Operation **IUMLStereotypeApplication::Stereotype**

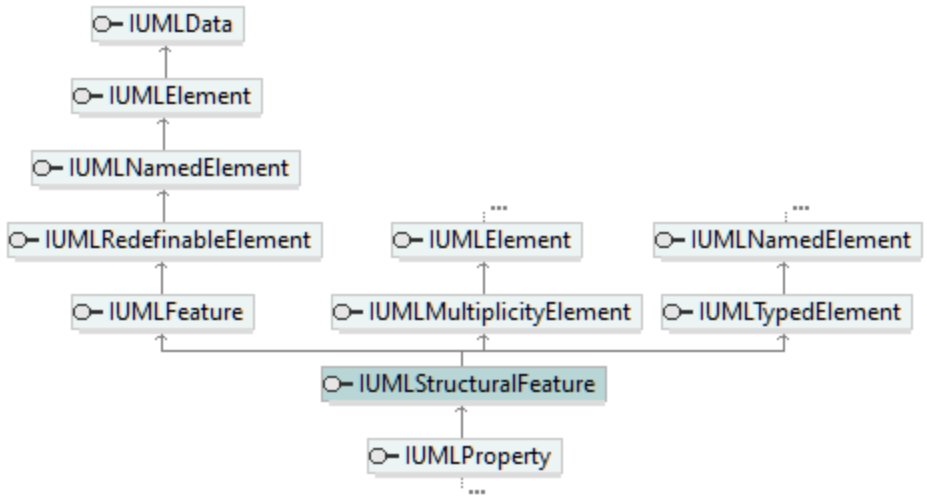
parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLStereotype**[1169]			

16.4.3.5.160 UModelAPI - IUMLStructuralFeature

Interface **IUMLStructuralFeature**

diagram



hierarchy	
typedElements	Interface **IUMLDataAll** 911 Interface **IUMLInstanceSpecification** 1085 Interface **IUMLSlot** 1162 Interface **IUMLStereotypeApplication** 1170 Operation **DefiningFeature** 920 **InsertSlotAt** 943 **SetSlotInstanceValueAt** 971 **SetSlotValueAt** 971 **SetTaggedValueAt** 971 Operation **InsertSlotAt** 1086 **SetSlotInstanceValueAt** 1086 **SetSlotValueAt** 1086 Operation **DefiningFeature** 1162 Operation **SetTaggedValueAt** 1171

Operation **IUMLStructuralFeature::IsReadOnly**

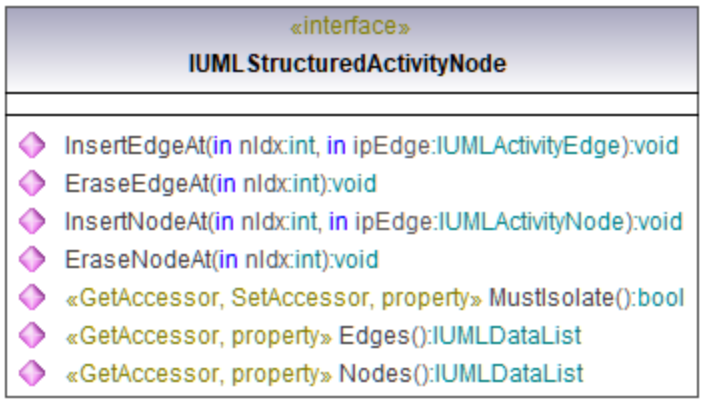
parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

© 2019-2025 Altova GmbH

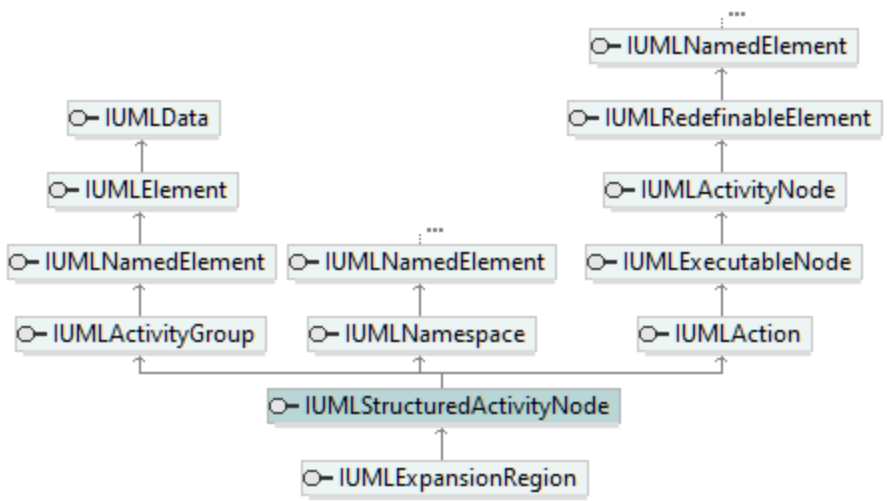
16.4.3.5.161 UModelAPI - IUMLStructuredActivityNode

Interface **IUMLStructuredActivityNode**

diagram



hierarchy



Operation **IUMLStructuredActivityNode::Edges**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			

Operation **IUMLStructuredActivityNode::EraseEdgeAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	**void**			

Operation **IUMLStructuredActivityNode::EraseNodeAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	**void**			

Operation **IUMLStructuredActivityNode::InsertEdgeAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	ipEdge	**in**	**IUMLActivityEdge** 988			
	return	**return**	**void**			

Operation **IUMLStructuredActivityNode::InsertNodeAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	ipEdge	**in**	**IUMLActivityNode** 992			
	return	**return**	**void**			

Operation **IUMLStructuredActivityNode::MustIsolate**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IUMLStructuredActivityNode::Nodes**

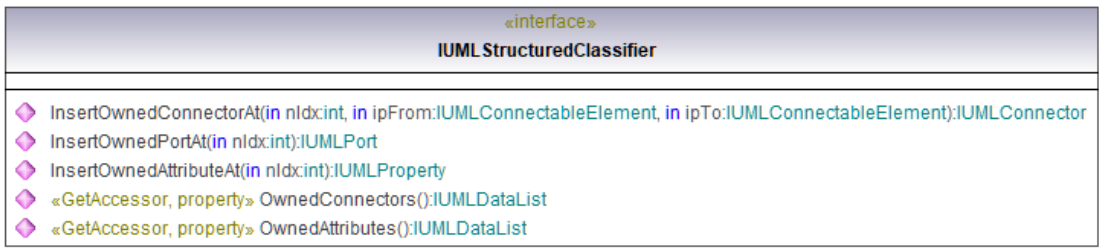
parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			

UML documentation generated by **UModel** UML Editor **http://www.altova.com/umodel** Wed Sep 25 10:33:04 2024

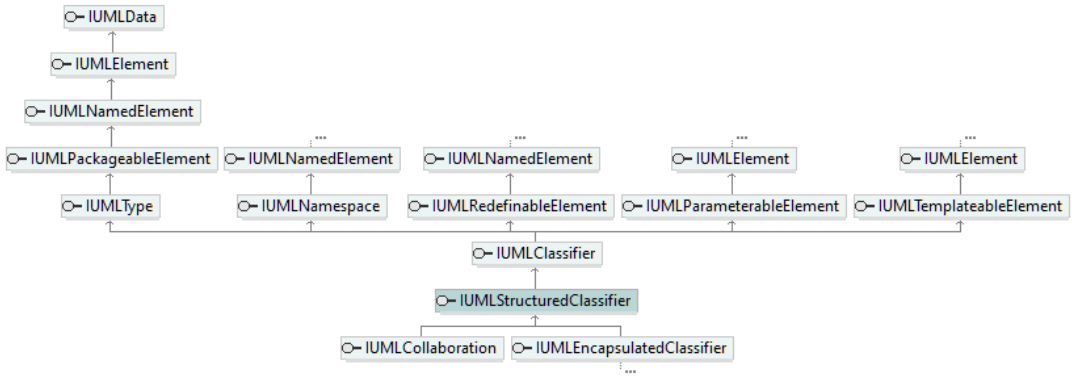
16.4.3.5.162 UModelAPI - IUMLStructuredClassifier

Interface **IUMLStructuredClassifier**

diagram



 © 2019-2025 Altova GmbH

hierarchy	

Operation **IUMLStructuredClassifier::InsertOwnedAttributeAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	**IUMLProperty** 1147			

Operation **IUMLStructuredClassifier::InsertOwnedConnectorAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	ipFrom	**in**	**IUMLConnectableElement** 1030			
	ipTo	**in**	**IUMLConnectableElement** 1030			
	return	**return**	**IUMLConnector** 1033			

Operation **IUMLStructuredClassifier::InsertOwnedPortAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	**IUMLPort** 1143			

Operation **IUMLStructuredClassifier::OwnedAttributes**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			
documentation	A list of elements of type IUMLProperty 1147.					

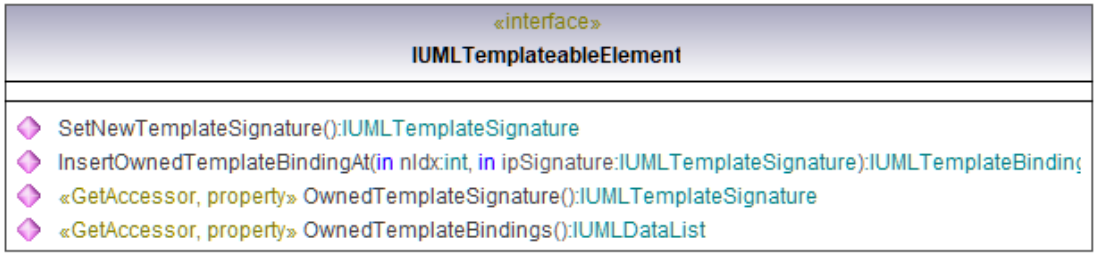
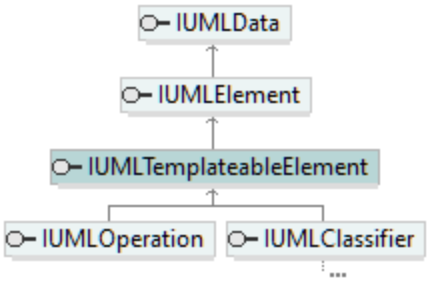
Operation **IUMLStructuredClassifier::OwnedConnectors**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			
documentation	A list of elements of type IUMLConnector 1033.					

UML documentation generated by UModel UML Editor http://www.altova.com/umodel Wed Sep 25 10:33:04 2024

16.4.3.5.163 UModelAPI - IUMLTemplateableElement

Interface **IUMLTemplateableElement**

diagram	
hierarchy	
typedElements	Interface IUMLDataAll 911 Interface IUMLTemplateBinding 1177 Interface IUMLTemplateSignature 1181 Operation BoundElement 916 Template 975 Operation BoundElement 1177 Operation Template 1181

Operation **IUMLTemplateableElement::InsertOwnedTemplateBindingAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	ipSignature	**in**	**IUMLTemplateSignature** 1181			
	return	**return**	**IUMLTemplateBinding** 1177			

Operation **IUMLTemplateableElement::OwnedTemplateBindings**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			
documentation	A list of elements of type IUMLTemplateBinding 1177.					

Operation **IUMLTemplateableElement::OwnedTemplateSignature**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLTemplateSignature** 1181			

© 2019-2025 Altova GmbH

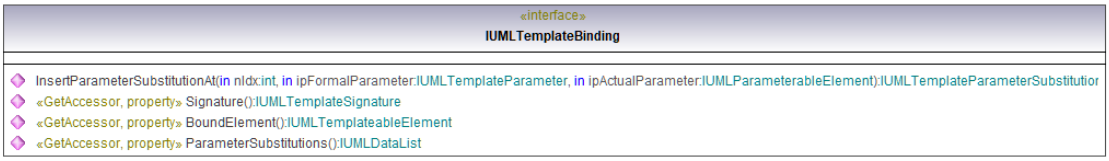
Operation **IUMLTemplateableElement::SetNewTemplateSignature**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLTemplateSignature** 1181			

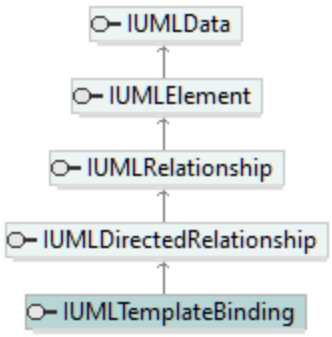
16.4.3.5.164 UModelAPI - IUMLTemplateBinding

Interface **IUMLTemplateBinding**

diagram



hierarchy



typedElements

Interface **IUMLDataAll** 911
Interface **IUMLTemplateableElement** 1176
Interface **IUMLTemplateParameterSubstitution** 1180
Operation **InsertOwnedTemplateBindingAt TemplateBinding** 976
Operation **InsertOwnedTemplateBindingAt** 1176
Operation **TemplateBinding** 1180

Operation **IUMLTemplateBinding::BoundElement**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLTemplateableElement** 1176			

Operation **IUMLTemplateBinding::InsertParameterSubstitutionAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	ipFormalParameter	**in**	**IUMLTemplateParameter** 1178			

	ipActualParameter **return**	in **return**	IUMLParameterableElement 1141 IUMLTemplateParameterSubstitution 1180			

Operation **IUMLTemplateBinding::ParameterSubstitutions**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLDataList 906			
documentation	A list of elements of type IUMLTemplateParameterSubstitution 1180.					

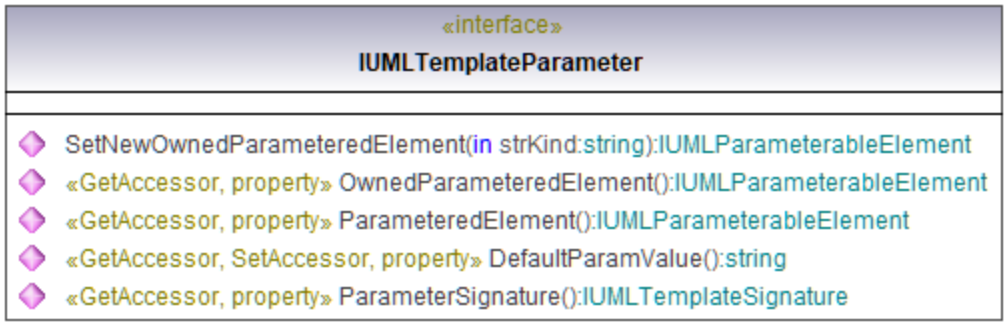
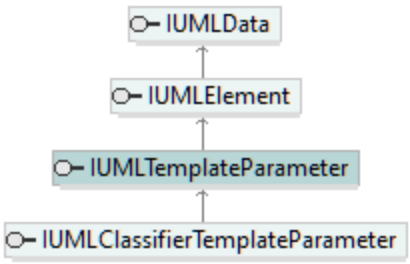
Operation **IUMLTemplateBinding::Signature**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLTemplateSignature 1181			

UML documentation generated by **UModel** UML Editor **http://www.altova.com/umodel** Wed Sep 25 10:33:04 2024

16.4.3.5.165 UModelAPI - IUMLTemplateParameter

Interface **IUMLTemplateParameter**

diagram	
hierarchy	
typedElements	Interface IUMLDataAll 911	Operation Formal 927 InsertParameterSubstitutionAt 942

 © 2019-2025 Altova GmbH

	Interface **IUMLParameterableElement** 1141 Interface **IUMLTemplateBinding** 1177 Interface **IUMLTemplateParameterSubstitution** 1180	Operation **OwningTemplateParameter** 960 **TemplateParameter** 976 Operation **OwningTemplateParameter** 1142 **TemplateParameter** 1142 Operation **InsertParameterSubstitutionAt** 1177 Operation **Formal** 1180

Operation **IUMLTemplateParameter::DefaultParamValue**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**string**			

Operation **IUMLTemplateParameter::OwnedParameteredElement**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLParameterableElement** 1141			

Operation **IUMLTemplateParameter::ParameteredElement**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLParameterableElement** 1141			

Operation **IUMLTemplateParameter::ParameterSignature**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLTemplateSignature** 1181			

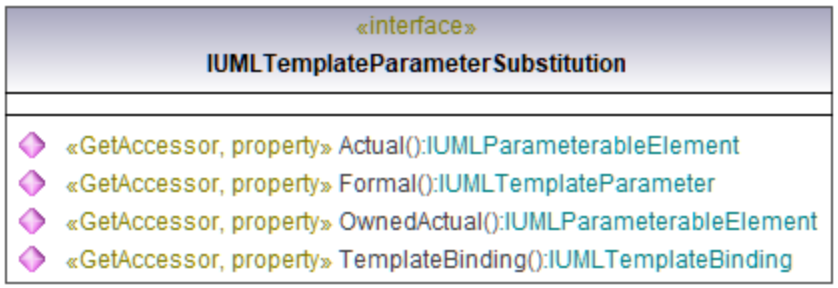
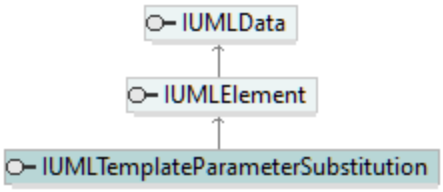
Operation **IUMLTemplateParameter::SetNewOwnedParameteredElement**

parameter	name	direction	type	type modifier	multiplicity	default
	strKind	**in**	**string**			
	return	**return**	**IUMLParameterableElement** 1141			

© 2019-2025 Altova GmbH

16.4.3.5.166 UModelAPI - IUMLTemplateParameterSubstitution

Interface **IUMLTemplateParameterSubstitution**

diagram	
hierarchy	
typedElements	Interface **IUMLDataAll** 911 Interface **IUMLTemplateBinding** 1177 Operation **InsertParameterSubstitutionAt** 942 Operation **InsertParameterSubstitutionAt** 1177

Operation **IUMLTemplateParameterSubstitution::Actual**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLParameterableElement** 1141			

Operation **IUMLTemplateParameterSubstitution::Formal**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLTemplateParameter** 1178			

Operation **IUMLTemplateParameterSubstitution::OwnedActual**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLParameterableElement** 1141			

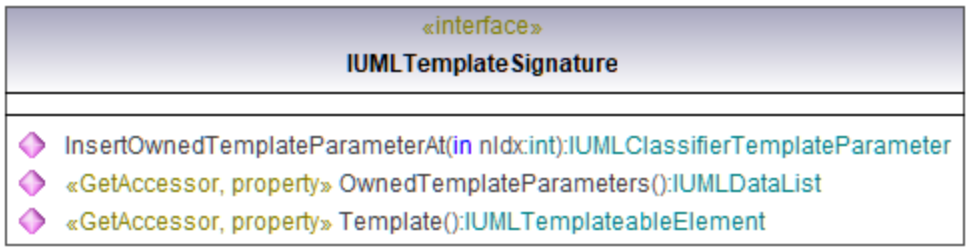
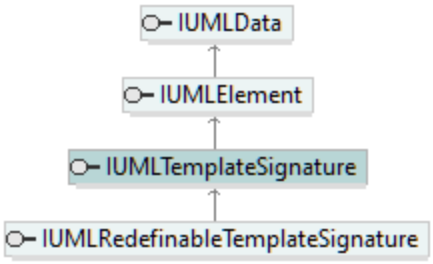
Operation **IUMLTemplateParameterSubstitution::TemplateBinding**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLTemplateBinding** 1177			

© 2019-2025 Altova GmbH

16.4.3.5.167 UModelAPI - IUMLTemplateSignature

Interface **IUMLTemplateSignature**

diagram	
hierarchy	
typedElements	Interface **IUMLDataAll** 911 Interface **IUMLTemplateableElement** 1176 Interface **IUMLTemplateBinding** 1177 Interface **IUMLTemplateParameter** 1178 Operation **InsertOwnedTemplateBindingAt** 941 **OwnedTemplateSignature** 959 **ParameterSignature** 961 **SetNewTemplateSignature** 970 **Signature** 972 Operation **InsertOwnedTemplateBindingAt** 1176 **OwnedTemplateSignature** 1176 **SetNewTemplateSignature** 1177 Operation **Signature** 1178 Operation **ParameterSignature** 1179

Operation **IUMLTemplateSignature::InsertOwnedTemplateParameterAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	**IUMLClassifierTemplateParameter** 1021			

Operation **IUMLTemplateSignature::OwnedTemplateParameters**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			
documentation	A list of elements of type IUMLTemplateParameter 1178.					

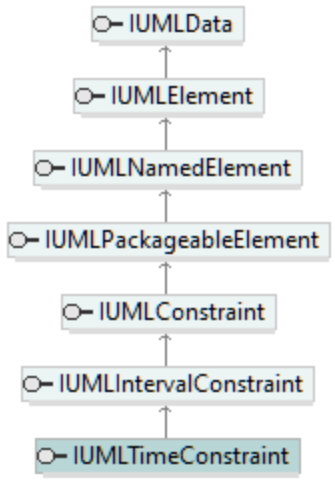
Operation **IUMLTemplateSignature::Template**

parameter	name	direction	type	type modifier	multiplicity	default

	return	**return**	**IUMLTemplateableElement** 1176

16.4.3.5.168 UModelAPI - IUMLTimeConstraint

Interface **IUMLTimeConstraint**

diagram	
hierarchy	

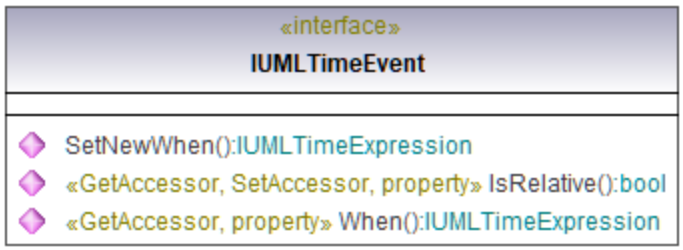
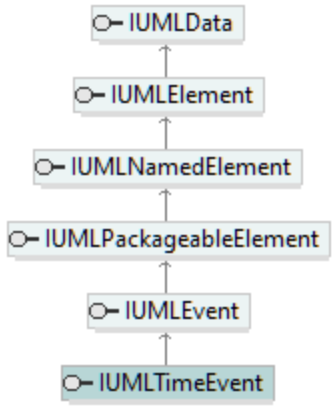
Operation **IUMLTimeConstraint::IsFirstEvent**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

 © 2019-2025 Altova GmbH

16.4.3.5.169 UModelAPI - IUMLTimeEvent

Interface **IUMLTimeEvent**

diagram	
hierarchy	

Operation **IUMLTimeEvent::IsRelative**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IUMLTimeEvent::SetNewWhen**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLTimeExpression** 1184			

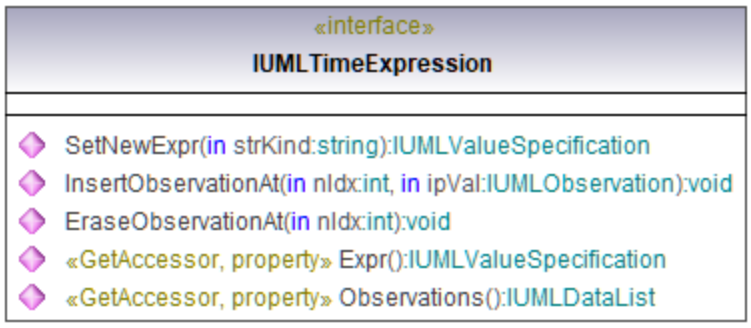
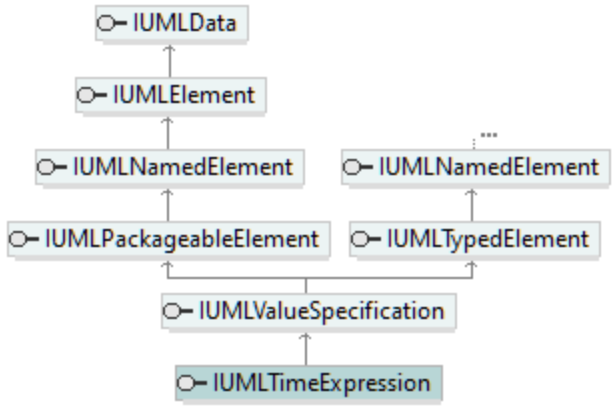
Operation **IUMLTimeEvent::When**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLTimeExpression** 1184			

© 2019-2025 Altova GmbH

16.4.3.5.170 UModelAPI - IUMLTimeExpression

Interface **IUMLTimeExpression**

diagram	
hierarchy	
typedElements	Interface IUMLDataAll 911 Interface IUMLTimeEvent 1183 Operation SetNewWhen 970 When 979 Operation SetNewWhen 1183 When 1183

Operation **IUMLTimeExpression::EraseObservationAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	**void**			

Operation **IUMLTimeExpression::Expr**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLValueSpecification 1195			

Operation **IUMLTimeExpression::InsertObservationAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	ipVal	**in**	IUMLObservation 1126			
	return	**return**	**void**			

© 2019-2025 Altova GmbH

Operation **IUMLTimeExpression::Observations**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			
documentation	A list of elements of type IUMLObservation 1126.					

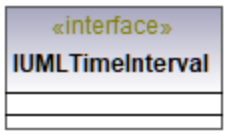
Operation **IUMLTimeExpression::SetNewExpr**

parameter	name	direction	type	type modifier	multiplicity	default
	strKind	**in**	**string**			
	return	**return**	**IUMLValueSpecification** 1195			

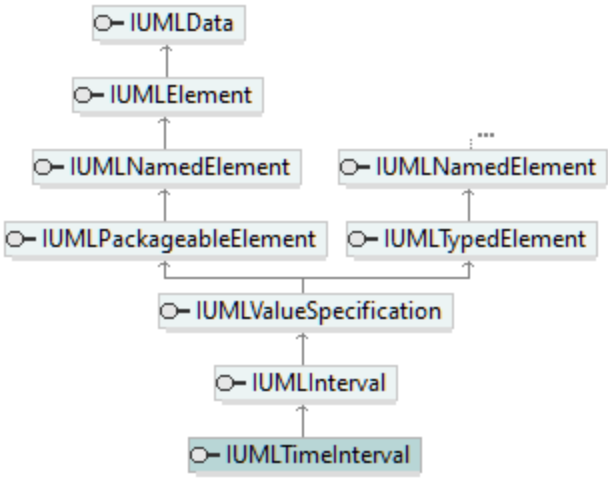
16.4.3.5.171 UModelAPI - IUMLTimeInterval

Interface **IUMLTimeInterval**

diagram



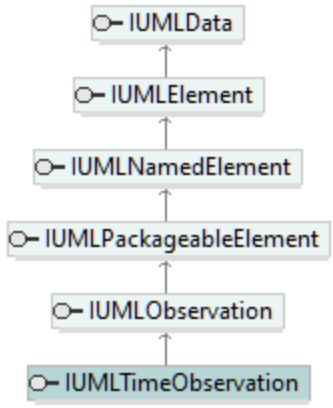
hierarchy



© 2019-2025 Altova GmbH

16.4.3.5.172 UModelAPI - IUMLTimeObservation

Interface **IUMLTimeObservation**

diagram	
hierarchy	

Operation **IUMLTimeObservation::IsFirstEvent**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

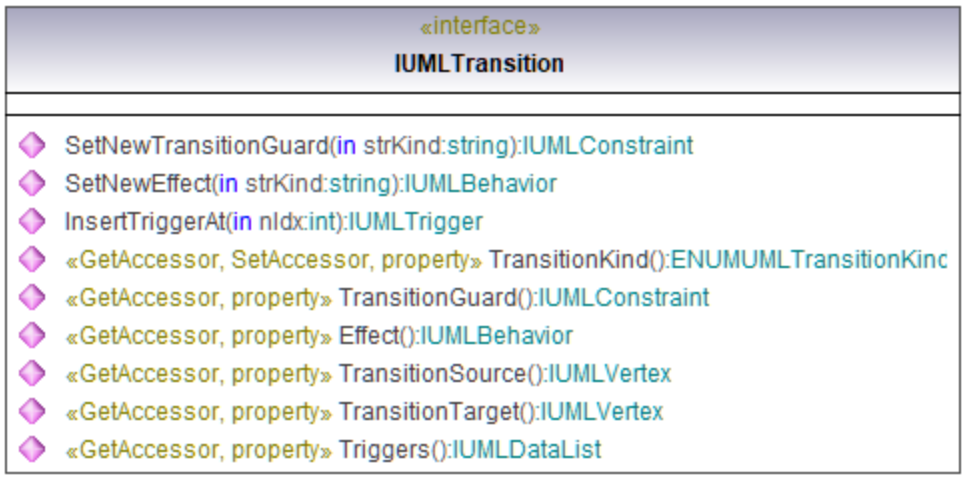
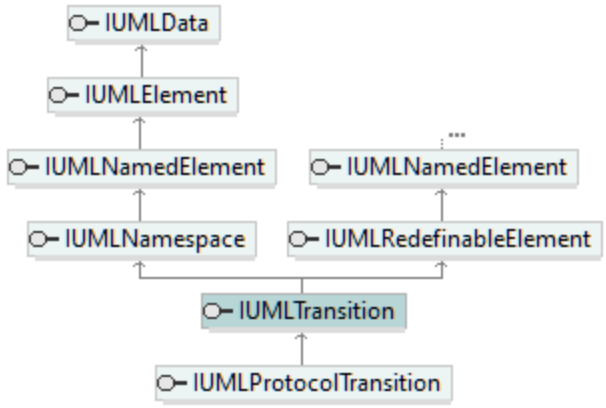
Operation **IUMLTimeObservation::TimeObservationEvent**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLNamedElement** 1117			

© 2019-2025 Altova GmbH

16.4.3.5.173 UModelAPI - IUMLTransition

Interface **IUMLTransition**

diagram	
hierarchy	
typedElements	Interface **IUMLDataAll** 911 Interface **IUMLRegion** 1157 Operation **InsertTransitionAt** 944 Operation **InsertTransitionAt** 1157

Operation **IUMLTransition::Effect**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLBehavior** 1002			

Operation **IUMLTransition::InsertTriggerAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	**IUMLTrigger** 1188			

Operation **IUMLTransition::SetNewEffect**

parameter	name	direction	type	type modifier	multiplicity	default

	strKind	**in**	**string**
	return	**return**	**IUMLBehavior** 1002

Operation **IUMLTransition::SetNewTransitionGuard**

parameter	name	direction	type	type modifier	multiplicity	default
	strKind	**in**	**string**			
	return	**return**	**IUMLConstraint** 1035			

Operation **IUMLTransition::TransitionGuard**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLConstraint** 1035			

Operation **IUMLTransition::TransitionKind**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**ENUMUMLTransitionKind** 1273			

Operation **IUMLTransition::TransitionSource**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLVertex** 1199			

Operation **IUMLTransition::TransitionTarget**

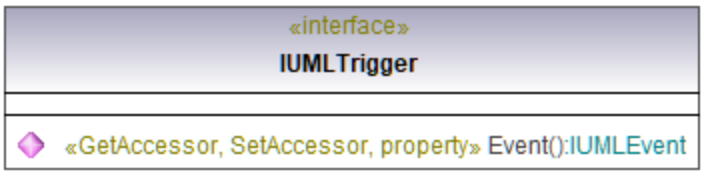
parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLVertex** 1199			

Operation **IUMLTransition::Triggers**

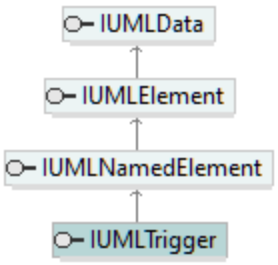
parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			
documentation	A list of elements of type IUMLTrigger 1188.					

16.4.3.5.174 UModelAPI - IUMLTrigger

Interface **IUMLTrigger**

diagram	

 © 2019-2025 Altova GmbH

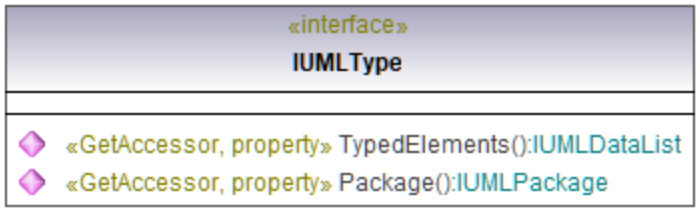
hierarchy	
typedElements	Interface **IUMLAcceptEventAction** 981 Interface **IUMLDataAll** 911 Interface **IUMLTransition** 1187 Operation **InsertActionTriggerAt** 982 Operation **InsertActionTriggerAt** 931 **InsertTriggerAt** 944 Operation **InsertTriggerAt** 1187

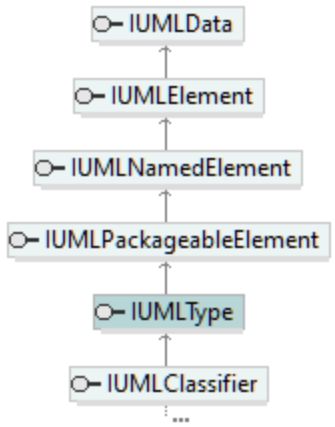
Operation **IUMLTrigger::Event**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLEvent** 1058			

16.4.3.5.175 UModelAPI - IUMLType

Interface **IUMLType**

diagram	

hierarchy	
typedElements	Interface **IUMLBehavioralFeature** 1004 Interface **IUMLDataAll** 911 Interface **IUMLOperation** 1131 Interface **IUMLTypedElement** 1190 Operation **InsertRaisedExceptionAt** 1005 Operation **InsertRaisedExceptionAt** 942 Operation **Type** 977 Operation **Type** 1132 Operation **Type** 1191

Operation **IUMLType::Package**

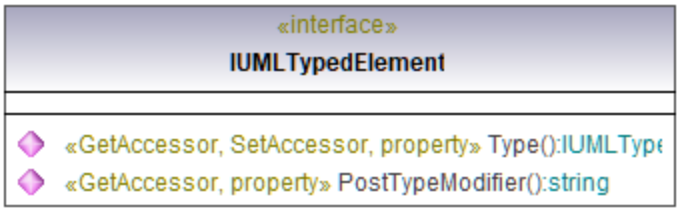
parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLPackage** 1134			

Operation **IUMLType::TypedElements**

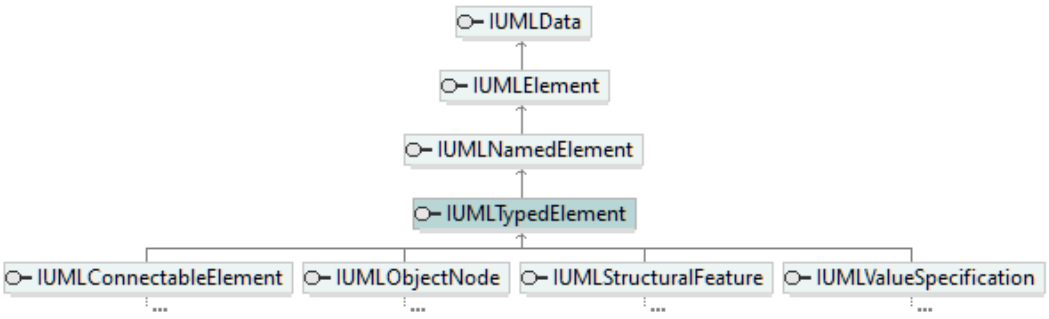
parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			
documentation	A list of elements of type IUMLTypedElement 1190.					

16.4.3.5.176 UModelAPI - IUMLTypedElement

Interface **IUMLTypedElement**

diagram	

 © 2019-2025 Altova GmbH

hierarchy	

Operation **IUMLTypedElement::PostTypeModifier**

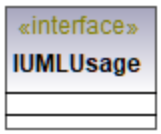
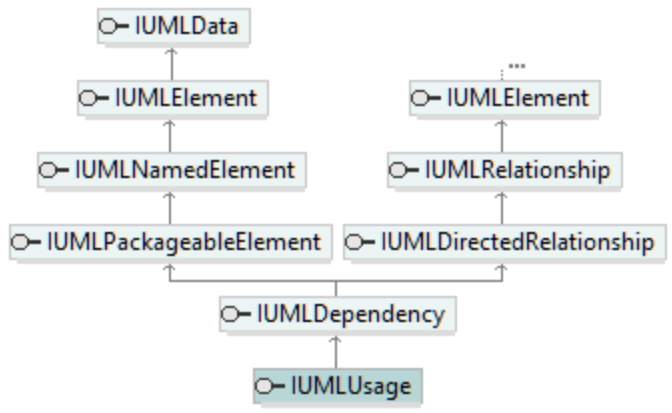
parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**string**			

Operation **IUMLTypedElement::Type**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLType** 1189			

16.4.3.5.177 UModelAPI - IUMLUsage

Interface **IUMLUsage**

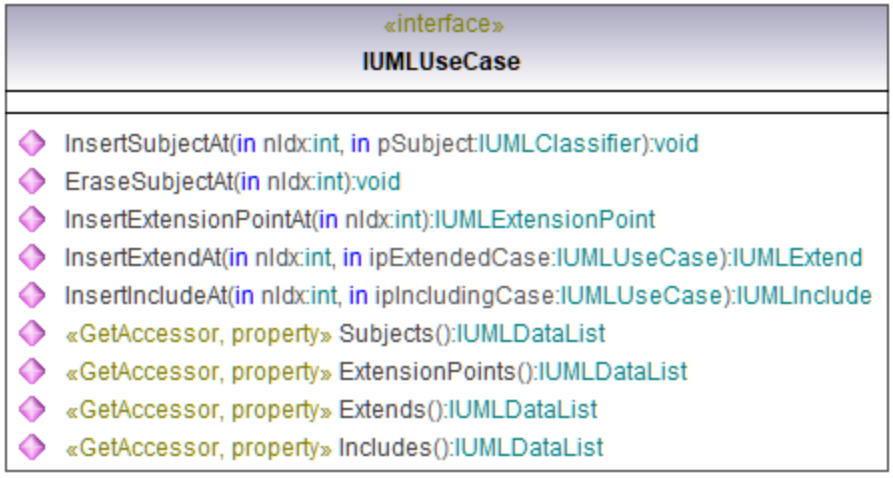
diagram	
hierarchy	

UML documentation generated by UModel UML Editor http://www.altova.com/umodel Wed Sep 25 10:33:04 2024

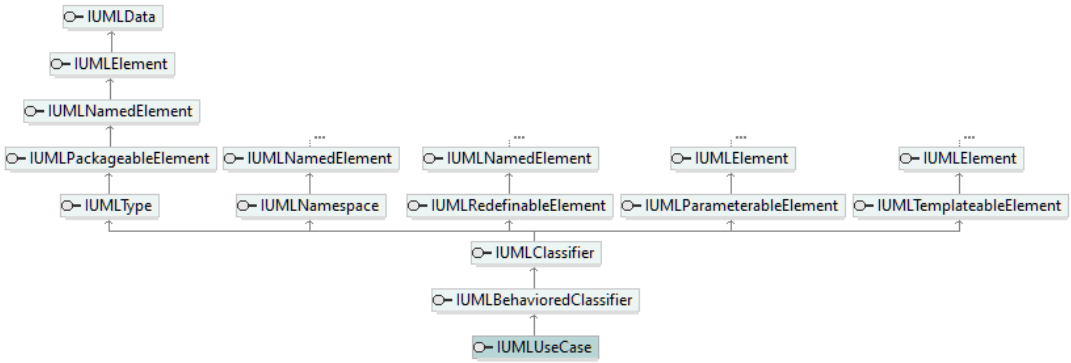
16.4.3.5.178 UModelAPI - IUMLUseCase

Interface **IUMLUseCase**

diagram



hierarchy



typedElements

Interface **IUMLClassifier** 1018
Interface **IUMLDataAll** 911
Interface **IUMLExtend** 1067
Interface **IUMLExtensionPoint** 1069
Interface **IUMLInclude** 1079
Interface **IUMLUseCase** 1192

Operation **InsertOwnedUseCaseAt** 1020
Operation **Addition** 913 **ExtendedCase** 926 **Extension** 926 **IncludingCase** 930 **InsertExtendAt** 934 **InsertIncludeAt** 935 **InsertOwnedUseCaseAt** 941 **UseCase** 978
Operation **ExtendedCase** 1068 **Extension** 1068
Operation **UseCase** 1039
Operation **Addition** 1079 **IncludingCase** 1079
Operation **InsertExtendAt** 1193 **InsertIncludeAt** 1193

Operation **IUMLUseCase::EraseSubjectAt**

 © 2019-2025 Altova GmbH

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	**void**			

Operation **IUMLUseCase::Extends**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			
documentation	A list of elements of type IUMLExtend 1067.					

Operation **IUMLUseCase::ExtensionPoints**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			
documentation	A list of elements of type IUMLExtensionPoint 1069.					

Operation **IUMLUseCase::Includes**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			
documentation	A list of elements of type IUMLInclude 1079.					

Operation **IUMLUseCase::InsertExtendAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	ipExtendedCase	**in**	**IUMLUseCase** 1192			
	return	**return**	**IUMLExtend** 1067			

Operation **IUMLUseCase::InsertExtensionPointAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	**IUMLExtensionPoint** 1069			

Operation **IUMLUseCase::InsertIncludeAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	ipIncludingCase	**in**	**IUMLUseCase** 1192			
	return	**return**	**IUMLInclude** 1079			

Operation **IUMLUseCase::InsertSubjectAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	pSubject	**in**	**IUMLClassifier** 1018			
	return	**return**	**void**			

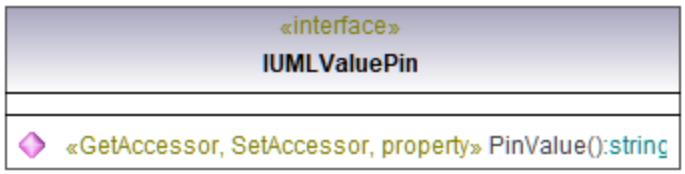
Operation **IUMLUseCase::Subjects**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			
documentation	A list of elements of type IUMLClassifier 1018.					

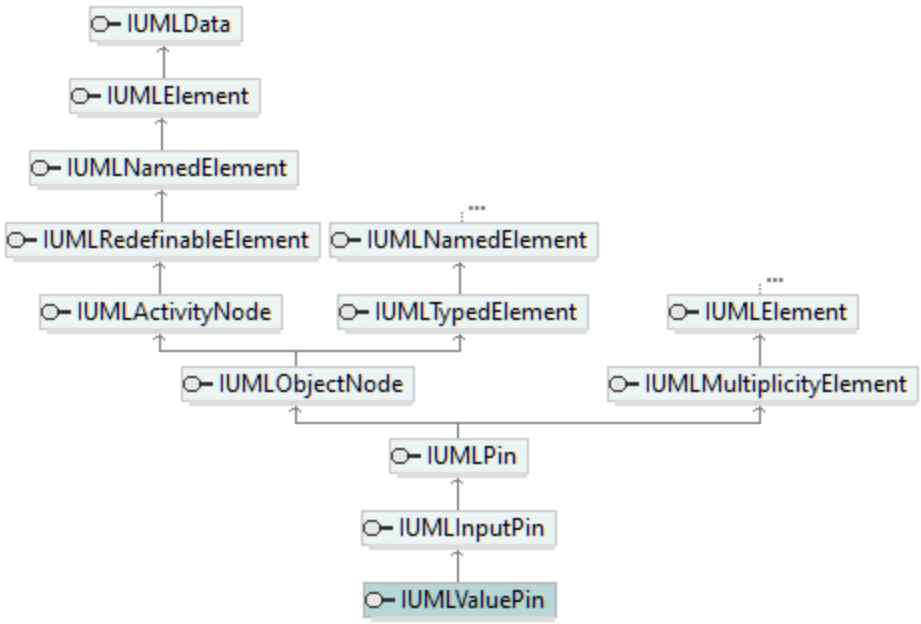
16.4.3.5.179 UModelAPI - IUMLValuePin

Interface **IUMLValuePin**

diagram



hierarchy



Operation **IUMLValuePin::PinValue**

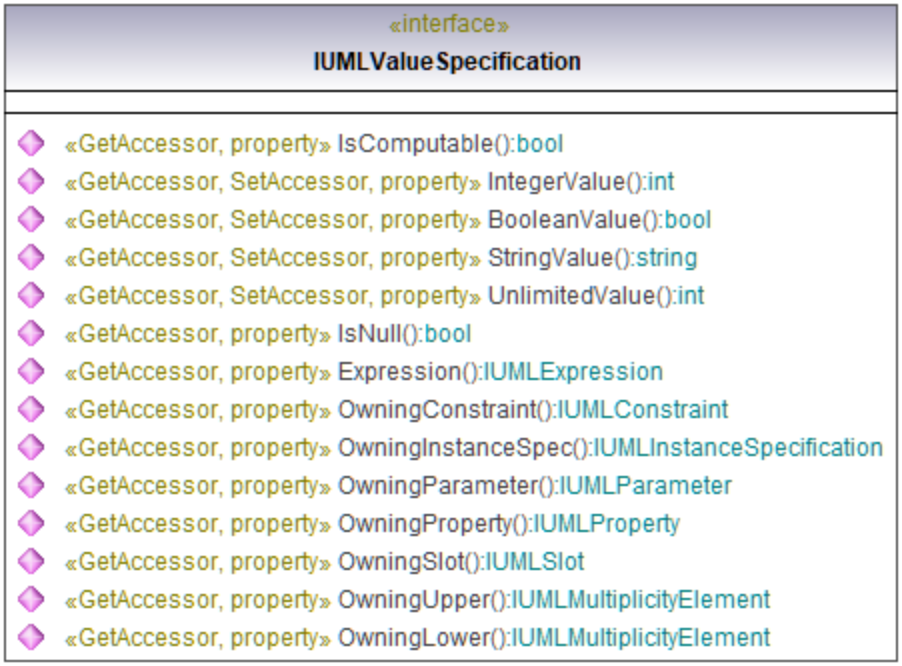
parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**string**			

 © 2019-2025 Altova GmbH

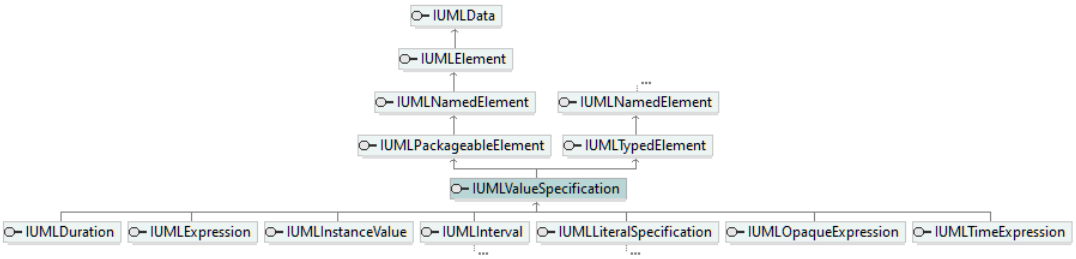
16.4.3.5.180 UModelAPI - IUMLValueSpecification

Interface **IUMLValueSpecification**

diagram



hierarchy



typedElements

Interface **IUMLChangeEvent** 1013

Interface **IUMLConstraint** 1035

Interface **IUMLDataAll** 911

Operation **ChangeExpression** 1014
SetNewChangeExpression 1014

Operation **SetNewSpecification** 1036
Specification 1037

Operation **ActionValue** 912
ChangeExpression 917
DefaultValue 920 **Expr** 925
InsertOwnedArgumentAt 938
InsertValueAt 944 **Max** 952
MaxInt 952 **Min** 954 **MinInt** 954
Selector 965 **SetNewActionValue** 966
SetNewChangeExpression 967
SetNewDefaultValue 967
SetNewExpr 968 **SetNewMax** 968
SetNewMaxInt 968 **SetNewMin** 968
SetNewMinInt 968
SetNewSelector 969
SetNewSpecification 969

	Interface IUMLDuration 1047	Operation Expr 1047 SetNewExpr 1048
	Interface IUMLInstanceSpecification 1085	Operation SetNewSpecification 1086
	Interface IUMLInteractionConstraint 1090	Operation MaxInt 1090 MinInt 1090
	Interface IUMLInterval 1099	Operation Max 1100 Min 1100 SetNewMax 1100 SetNewMin 1100
	Interface IUMLLifeline 1104	Operation Selector 1105 SetNewSelector 1105
	Interface IUMLMessage 1111	Operation InsertOwnedArgumentAt 1111
	Interface IUMLParameter 1140	Operation DefaultValue 1140 SetNewDefaultValue 1141
	Interface IUMLProperty 1147	Operation DefaultValue 1148 SetNewDefaultValue 1150
	Interface IUMLSlot 1162	Operation InsertValueAt 1162
	Interface IUMLStereotypeApplication 1170	Operation SetPredefinedTaggedValueAt 1171 SetTaggedValueAt 1171
	Interface IUMLTimeExpression 1184	Operation Expr 1184 SetNewExpr 1185
	Interface IUMLValueSpecificationAction 1198	Operation ActionValue 1198 SetNewActionValue 1198
		SetPredefinedTaggedValueAt 970
		SetSlotValueAt 971
		SetTaggedValueAt 971
		Specification 973
		SetSlotValueAt 1086 Specification 1087

Operation **IUMLValueSpecification::BooleanValue**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IUMLValueSpecification::Expression**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLExpression 1066			

Operation **IUMLValueSpecification::IntegerValue**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**int**			

Operation **IUMLValueSpecification::IsComputable**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IUMLValueSpecification::IsNull**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IUMLValueSpecification::OwningConstraint**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	IUMLConstraint 1035			

Operation **IUMLValueSpecification::OwningInstanceSpec**

© 2019-2025 Altova GmbH

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLInstanceSpecification** 1085			

Operation **IUMLValueSpecification::OwningLower**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLMultiplicityElement** 1116			

Operation **IUMLValueSpecification::OwningParameter**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLParameter** 1140			

Operation **IUMLValueSpecification::OwningProperty**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLProperty** 1147			

Operation **IUMLValueSpecification::OwningSlot**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLSlot** 1162			

Operation **IUMLValueSpecification::OwningUpper**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLMultiplicityElement** 1116			

Operation **IUMLValueSpecification::StringValue**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**string**			

Operation **IUMLValueSpecification::UnlimitedValue**

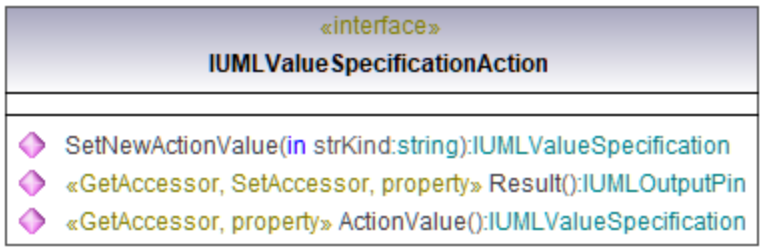
parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**int**			

© 2019-2025 Altova GmbH

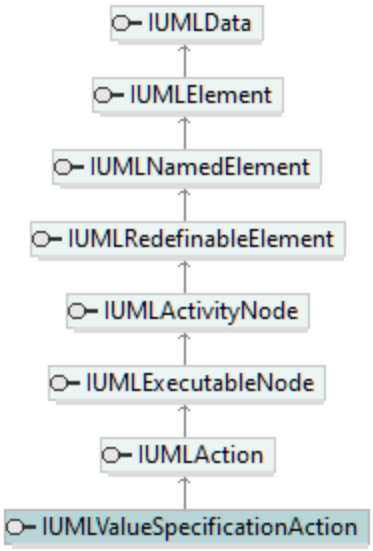
16.4.3.5.181 UModelAPI - IUMLValueSpecificationAction

Interface **IUMLValueSpecificationAction**

diagram



hierarchy



Operation **IUMLValueSpecificationAction::ActionValue**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLValueSpecification** 1195			

Operation **IUMLValueSpecificationAction::Result**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLOutputPin** 1133			

Operation **IUMLValueSpecificationAction::SetNewActionValue**

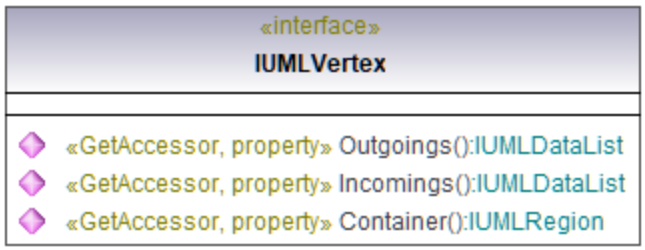
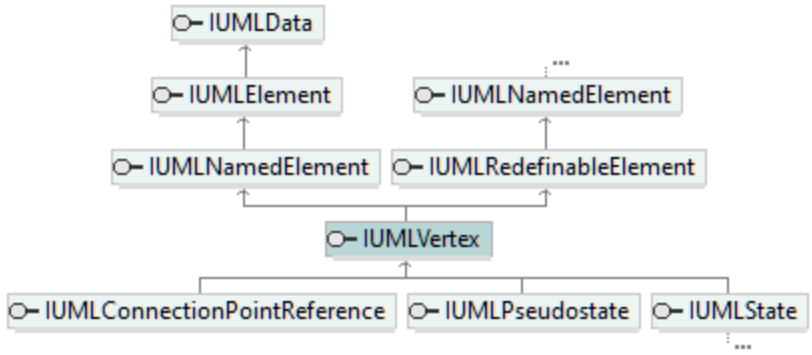
parameter	name	direction	type	type modifier	multiplicity	default
	strKind	**in**	**string**			
	return	**return**	**IUMLValueSpecification** 1195			

© 2019-2025 Altova GmbH

UML documentation generated by **UModel** UML Editor **http://www.altova.com/umodel** Wed Sep 25 10:33:04 2024

16.4.3.5.182 UModelAPI - IUMLVertex

Interface **IUMLVertex**

diagram	
hierarchy	
typedElements	Interface **IUMLDataAll** 911 Interface **IUMLRegion** 1157 Interface **IUMLTransition** 1187 Operation **InsertSubVertexAt** 944 **InsertTransitionAt** 944 **TransitionSource** 977 **TransitionTarget** 977 Operation **InsertSubVertexAt** 1157 **InsertTransitionAt** 1157 Operation **TransitionSource** 1188 **TransitionTarget** 1188

Operation **IUMLVertex::Container**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLRegion** 1157			

Operation **IUMLVertex::Incomings**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			
documentation	A list of elements of type IUMLTransition 1187.					

Operation **IUMLVertex::Outgoings**

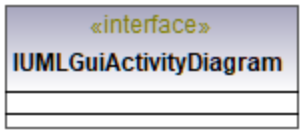
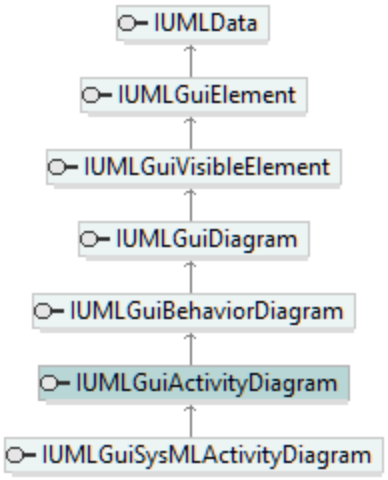
parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			
documentation	A list of elements of type IUMLTransition 1187.					

16.4.3.6 IUMLGuiElement

This is a list of Altova-specific elements for diagrams, and members used to show IUMLElements 980 on diagrams.

16.4.3.6.1 UModelAPI - IUMLGuiActivityDiagram

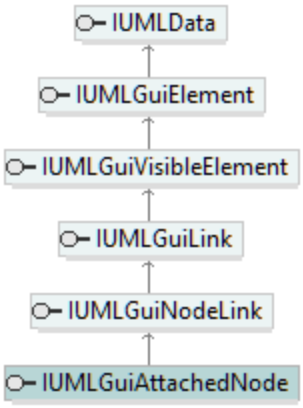
Interface **IUMLGuiActivityDiagram**

diagram	
hierarchy	

 © 2019-2025 Altova GmbH

16.4.3.6.2 UModelAPI - IUMLGuiAttachedNode

Interface **IUMLGuiAttachedNode**

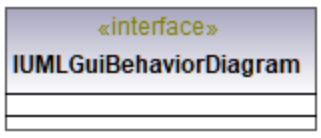
diagram	
hierarchy	
documentation	This GUI element is a node (possibly without a linked IUMLElement[1050]) which is directly attached to a IUMLGuiNodeLink[1226]. It disappears and pops up based on data set in the element of the IUMLGuiNodeLink[1226] it is attached to. The user usually only has control of this element via styles or the node it is attached to. This node is used as graphical object on diagrams to represent Tagged Values for example.

Operation **IUMLGuiAttachedNode::AttachedTo**

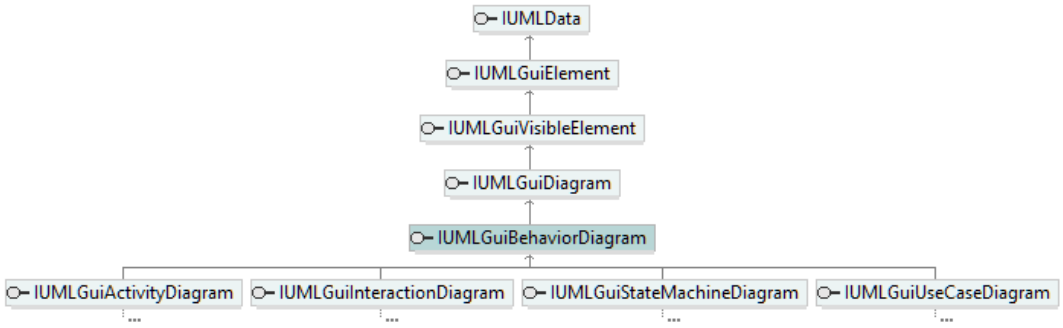
parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLGuiLink**[1224]			

16.4.3.6.3 UModelAPI - IUMLGuiBehaviorDiagram

Interface **IUMLGuiBehaviorDiagram**

diagram	

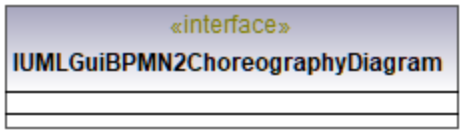
hierarchy



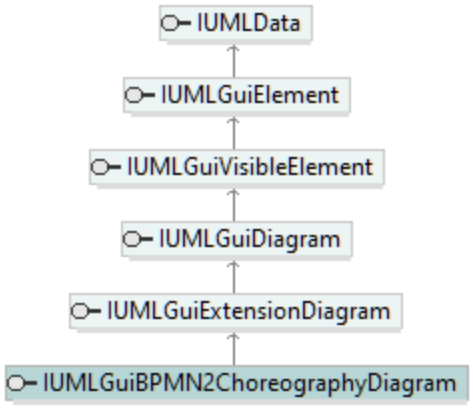
16.4.3.6.4 UModelAPI - IUMLGuiBPMN2ChoreographyDiagram

Interface **IUMLGuiBPMN2ChoreographyDiagram**

diagram



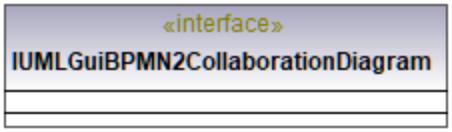
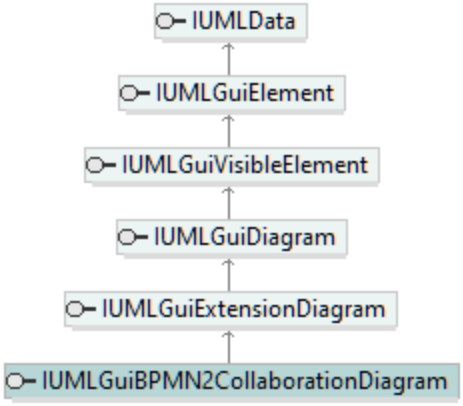
hierarchy



 © 2019-2025 Altova GmbH

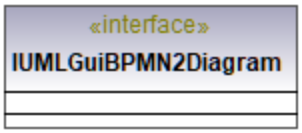
16.4.3.6.5 UModelAPI - IUMLGuiBPMN2CollaborationDiagram

Interface **IUMLGuiBPMN2CollaborationDiagram**

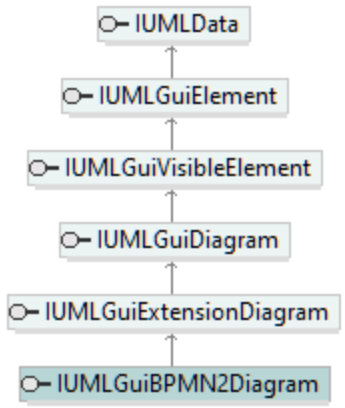
diagram	
hierarchy	

16.4.3.6.6 UModelAPI - IUMLGuiBPMN2Diagram

Interface **IUMLGuiBPMN2Diagram**

diagram	

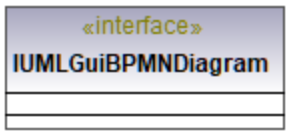
hierarchy



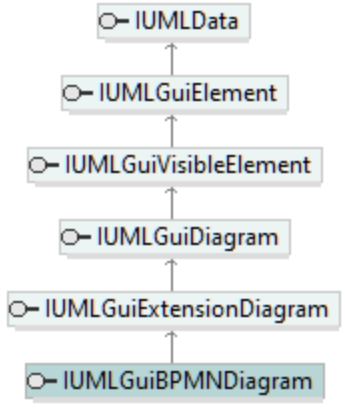
16.4.3.6.7 UModelAPI - IUMLGuiBPMNDiagram

Interface **IUMLGuiBPMNDiagram**

diagram



hierarchy

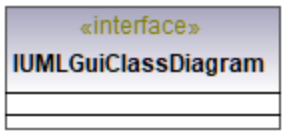


 © 2019-2025 Altova GmbH

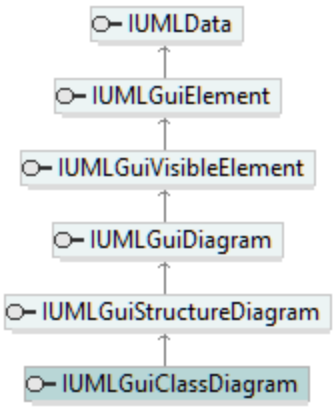
16.4.3.6.8 UModelAPI - IUMLGuiClassDiagram

Interface **IUMLGuiClassDiagram**

diagram



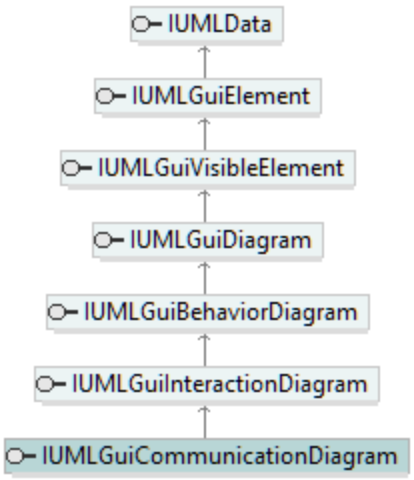
hierarchy



16.4.3.6.9 UModelAPI - IUMLGuiCommunicationDiagram

Interface **IUMLGuiCommunicationDiagram**

diagram

hierarchy	

Operation **IUMLGuiCommunicationDiagram::AddUMLGuiCommunicationLink**

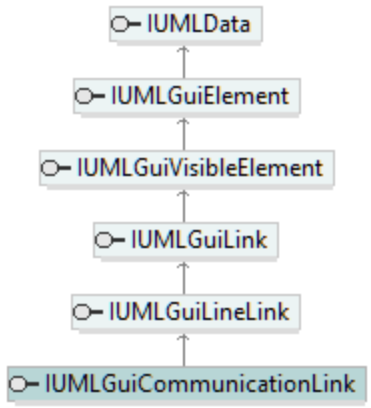
parameter	name	direction	type	type modifier	multiplicity	default
	ipFromLink	**in**	IUMLGuiLink 1224			
	ipToLink	**in**	IUMLGuiLink 1224			
	return	**return**	IUMLGuiCommunicationLink 1206			

16.4.3.6.10 UModelAPI - IUMLGuiCommunicationLink

Interface **IUMLGuiCommunicationLink**

diagram	

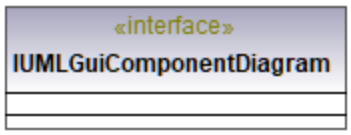
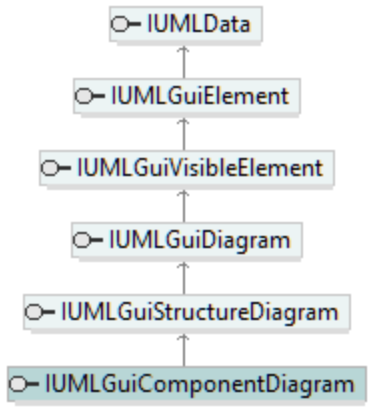
 © 2019-2025 Altova GmbH

hierarchy	
typedElements	Interface **IUMLGuiCommunicationDiagram** 1205 Operation **AddUMLGuiCommunicationLink** 1206
documentation	This line link is used on communication diagrams to provide a connection link for messages between lifelines.

UML documentation generated by **UModel** UML Editor **http://www.altova.com/umodel** Wed Sep 25 10:33:04 2024

16.4.3.6.11 UModelAPI - IUMLGuiComponentDiagram

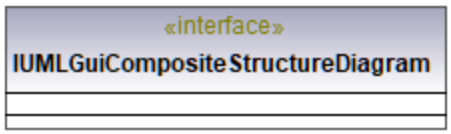
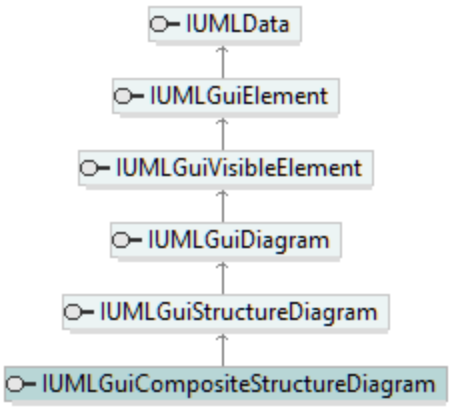
Interface **IUMLGuiComponentDiagram**

diagram	
hierarchy	

UML documentation generated by UModel UML Editor http://www.altova.com/umodel Wed Sep 25 10:33:04 2024

16.4.3.6.12 UModelAPI - IUMLGuiCompositeStructureDiagram

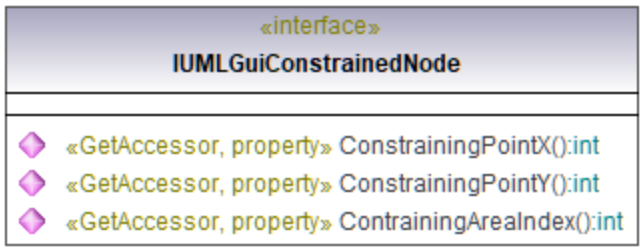
Interface **IUMLGuiCompositeStructureDiagram**

diagram	
hierarchy	

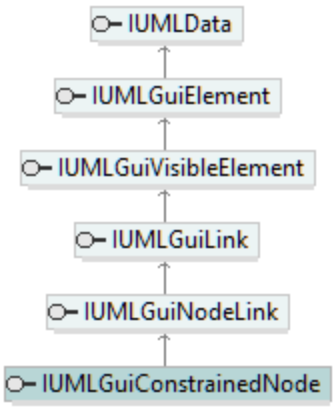
UML documentation generated by UModel UML Editor http://www.altova.com/umodel Wed Sep 25 10:33:04 2024

16.4.3.6.13 UModelAPI - IUMLGuiConstrainedNode

Interface **IUMLGuiConstrainedNode**

diagram	

 © 2019-2025 Altova GmbH

hierarchy	
documentation	This node link is used to represent objects on diagrams which can be directly attached to a parent node link and placed relatively to and in special constraining areas of the parent. Used for example for Pins on Activtiy Diagrams.

Operation **IUMLGuiConstrainedNode::ConstrainingPointX**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**int**			
documentation	X coordinate relative to the uppor left position of the contraining area.					

Operation **IUMLGuiConstrainedNode::ConstrainingPointY**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**int**			
documentation	Y coordinate relative to the uppor left position of the contraining area.					

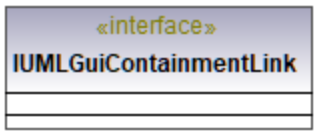
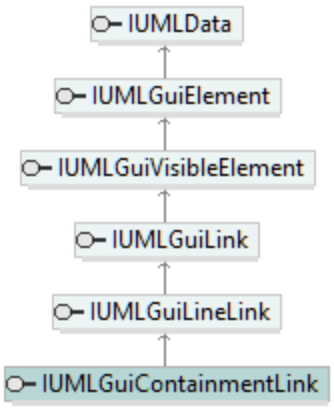
Operation **IUMLGuiConstrainedNode::ContrainingAreaIndex**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**int**			
documentation	Defines the index of the area where this node is currently in and to which its relative position has its origin.					

© 2019-2025 Altova GmbH

16.4.3.6.14 UModelAPI - IUMLGuiContainmentLink

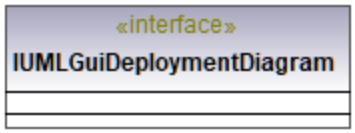
Interface **IUMLGuiContainmentLink**

diagram	
hierarchy	
typedElements	Interface **IUMLDataAll** 911 Interface **IUMLGuiDiagram** 1211 Operation **AddUMLGuiContainmentLink** 913 Operation **AddUMLGuiContainmentLink** 1212

UML documentation generated by **UModel** UML Editor **http://www.altova.com/umodel** Wed Sep 25 10:33:04 2024

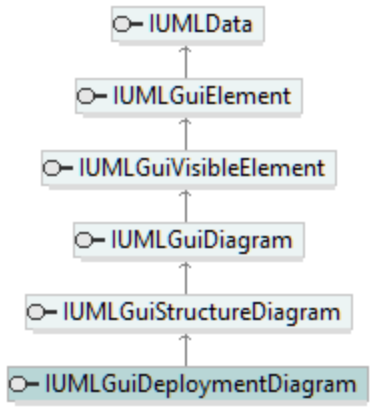
16.4.3.6.15 UModelAPI - IUMLGuiDeploymentDiagram

Interface **IUMLGuiDeploymentDiagram**

diagram	

© 2019-2025 Altova GmbH

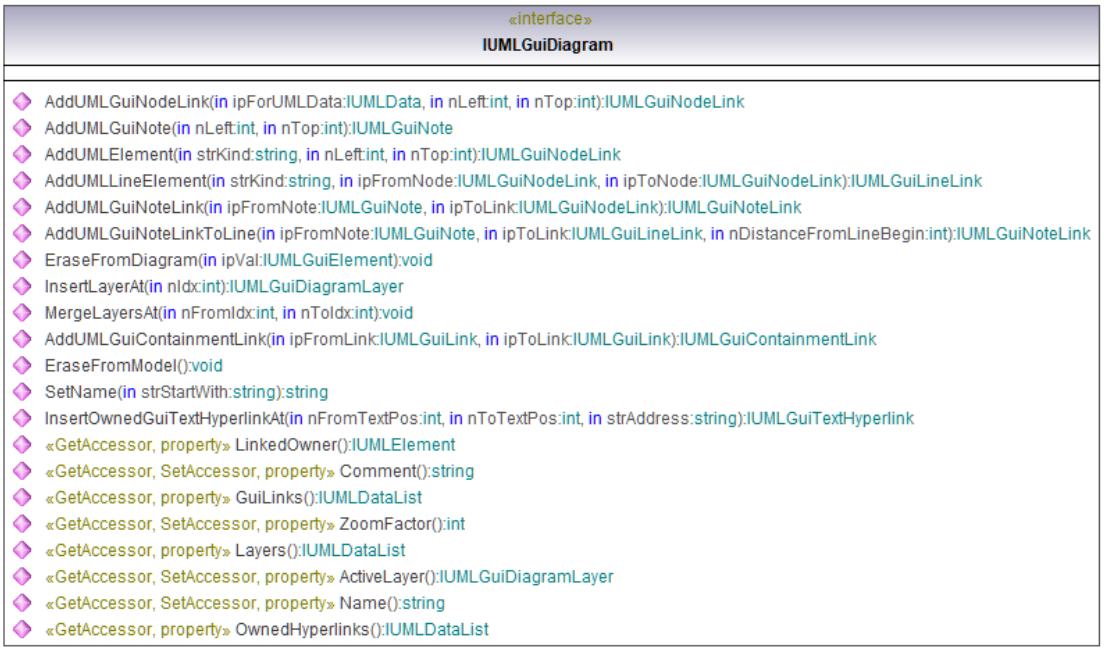
hierarchy

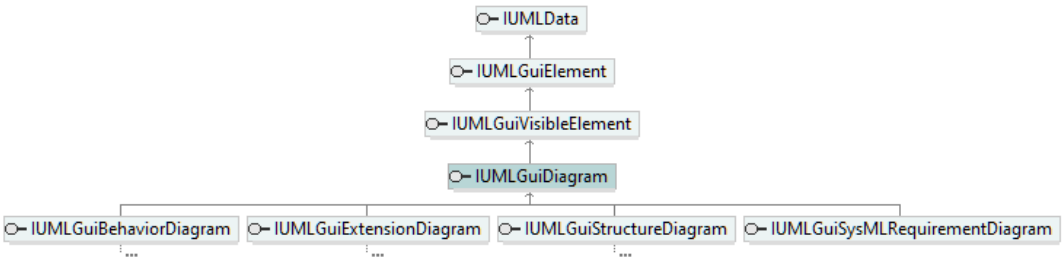


16.4.3.6.16 UModelAPI - IUMLGuiDiagram

Interface **IUMLGuiDiagram**

diagram



hierarchy	
typedElements	Interface **IDiagramWindow** 825 Interface **IDocument** 832 Interface **IUMLDataAll** 911 Interface **IUMLGuiRootElement** 1234 Interface **IUMLGuiSubDiagramNode** 1243 Operation **Diagram** 826 Operation **OpenDiagram** 837 Operation **InsertOwnedDiagramAt** 939 **ReferencedDiagram** 963 Operation **InsertOwnedDiagramAt** 1234 Operation **ReferencedDiagram** 1243
documentation	Represents an UML diagram and contains all layers, nodes (represented as IUMLGuiNodeLink 1226) and lines (represented as IUMLGuiLineLink 1222).Use the property GuiLinks 1214 to access these.

Operation **IUMLGuiDiagram::ActiveLayer**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLGuiDiagramLayer** 1215			

Operation **IUMLGuiDiagram::AddUMLElement**

parameter	name	direction	type	type modifier	multiplicity	default
	strKind	**in**	**string**			
	nLeft	**in**	**int**			
	nTop	**in**	**int**			
	return	**return**	**IUMLGuiNodeLink** 1226			
documentation	Adds a new UML element (e.g. IUMLClass 1015, IUMLPackage 1134,...) to the model and shows it with a new IUMLGuiNodeLink 1226 on the diagram.					

Operation **IUMLGuiDiagram::AddUMLGuiContainmentLink**

parameter	name	direction	type	type modifier	multiplicity	default
	ipFromLink	**in**	**IUMLGuiLink** 1224			
	ipToLink	**in**	**IUMLGuiLink** 1224			
	return	**return**	**IUMLGuiContainmentLink** 1210			

Operation **IUMLGuiDiagram::AddUMLGuiNodeLink**

parameter	name	direction	type	type modifier	multiplicity	default
	ipForUMLData	**in**	**IUMLData** 904			
	nLeft	**in**	**int**			
	nTop	**in**	**int**			
	return	**return**	**IUMLGuiNodeLink** 1226			
documentation	Adds a new IUMLGuiNodeLink 1226 for an existing UML element (e.g. IUMLClass 1015, IUMLPackage 1134,...) on the diagram.					

© 2019-2025 Altova GmbH

Operation **IUMLGuiDiagram::AddUMLGuiNote**

parameter	name	direction	type	type modifier	multiplicity	default
	nLeft	**in**	**int**			
	nTop	**in**	**int**			
	return	**return**	**IUMLGuiNote** 1228			

Operation **IUMLGuiDiagram::AddUMLGuiNoteLink**

parameter	name	direction	type	type modifier	multiplicity	default
	ipFromNote	**in**	**IUMLGuiNote** 1228			
	ipToLink	**in**	**IUMLGuiNodeLink** 1226			
	return	**return**	**IUMLGuiNoteLink** 1229			

Operation **IUMLGuiDiagram::AddUMLGuiNoteLinkToLine**

parameter	name	direction	type	type modifier	multiplicity	default
	ipFromNote	**in**	**IUMLGuiNote** 1228			
	ipToLink	**in**	**IUMLGuiLineLink** 1222			
	nDistanceFromLineBegin	**in**	**int**			
	return	**return**	**IUMLGuiNoteLink** 1229			

Operation **IUMLGuiDiagram::AddUMLLineElement**

parameter	name	direction	type	type modifier	multiplicity	default
	strKind	**in**	**string**			
	ipFromNode	**in**	**IUMLGuiNodeLink** 1226			
	ipToNode	**in**	**IUMLGuiNodeLink** 1226			
	return	**return**	**IUMLGuiLineLink** 1222			
documentation	Adds a new UML line element (e.g. IUMLGeneralization 1074, IUMLAssociation 1000,...) to the model and shows it with a new IUMLGuiLineLink 1222 on the diagram.					

Operation **IUMLGuiDiagram::Comment**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**string**			

Operation **IUMLGuiDiagram::EraseFromDiagram**

parameter	name	direction	type	type modifier	multiplicity	default
	ipVal	**in**	**IUMLGuiElement** 1216			
	return	**return**	**void**			
documentation	Use this function to erase the element from the diagram only. Use IUMLElement 1050::EraseFromModel 1051 to erase from the model and all diagrams.					

Operation **IUMLGuiDiagram::EraseFromModel**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**void**			

Operation **IUMLGuiDiagram::GuiLinks**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			
documentation	A list of elements of type IUMLGuiLink 1224 which are displayed directly on this diagram. Usually, these are IUMLGuiNodeLink 1226's and IUMLGuiLineLink 1222s.					

Operation **IUMLGuiDiagram::InsertLayerAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	**IUMLGuiDiagramLayer** 1215			

Operation **IUMLGuiDiagram::InsertOwnedGuiTextHyperlinkAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nFromTextPos	**in**	**int**			
	nToTextPos	**in**	**int**			
	strAddress	**in**	**string**			
	return	**return**	**IUMLGuiTextHyperlink** 1250			

Operation **IUMLGuiDiagram::Layers**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			
documentation	A list of all layers in the diagam. The list contains elements of type IUMLGuiDiagramLayer 1215.					

Operation **IUMLGuiDiagram::LinkedOwner**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLElement** 1050			

Operation **IUMLGuiDiagram::MergeLayersAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nFromIdx	**in**	**int**			
	nToIdx	**in**	**int**			
	return	**return**	**void**			

Operation **IUMLGuiDiagram::Name**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**string**			

Operation **IUMLGuiDiagram::OwnedHyperlinks**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			

© 2019-2025 Altova GmbH

Operation **IUMLGuiDiagram::SetName**

parameter	name	direction	type	type modifier	multiplicity	default
	strStartWith	**in**	**string**			
	return	**return**	**string**			

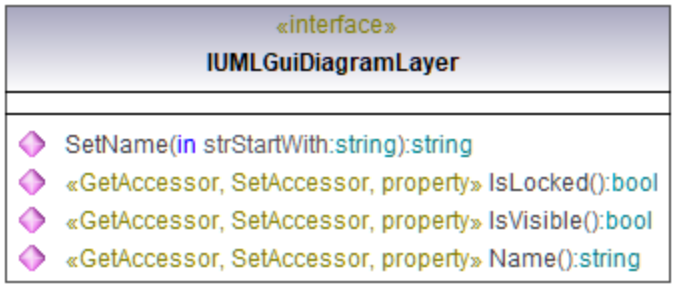
Operation **IUMLGuiDiagram::ZoomFactor**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**int**			

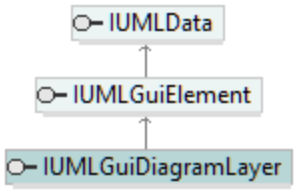
16.4.3.6.17 UModelAPI - IUMLGuiDiagramLayer

Interface **IUMLGuiDiagramLayer**

diagram



hierarchy



typedElements

Interface **IUMLDataAll** 911

Interface **IUMLGuiDiagram** 1211

Interface **IUMLGuiLink** 1224

Operation **ActiveLayer** 912 **InsertLayerAt** 936 **Layer** 950

Operation **ActiveLayer** 1212 **InsertLayerAt** 1214

Operation **Layer** 1225

documentation

Represents a layer on an IUMLGuiDiagram 1211. Makes it possible to group elements on a diagram into categories, to lock/unlock them and to make them visible or invisible.

Operation **IUMLGuiDiagramLayer::IsLocked**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IUMLGuiDiagramLayer::IsVisible**

parameter	name	direction	type	type modifier	multiplicity	default

	return	**return**	**bool**

Operation **IUMLGuiDiagramLayer::Name**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**string**			

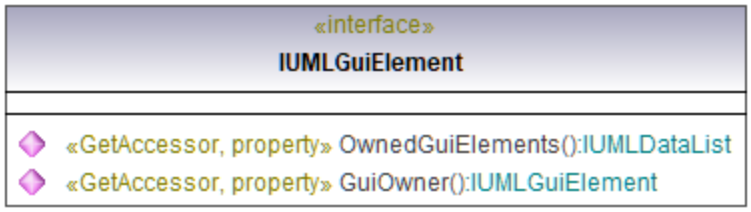
Operation **IUMLGuiDiagramLayer::SetName**

parameter	name	direction	type	type modifier	multiplicity	default
	strStartWith	**in**	**string**			
	return	**return**	**string**			

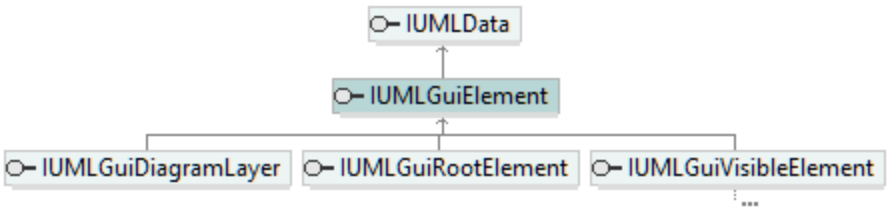
16.4.3.6.18 UModelAPI - IUMLGuiElement

Interface **IUMLGuiElement**

diagram



hierarchy



typedElements

Interface IUMLDataAll 911
Interface IUMLGuiDiagram 1211
Interface IUMLGuiElement 1216
Interface IUMLGuiLineLink 1222

Operation EraseFromDiagram 923 GuiOwner 929 LineBegin 951 LineEnd 951
Operation EraseFromDiagram 1213
Operation GuiOwner 1216
Operation LineBegin 1223 LineEnd 1223

documentation

The base class for all graphical objects.

Operation **IUMLGuiElement::GuiOwner**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLGuiElement** 1216			

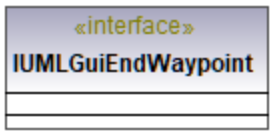
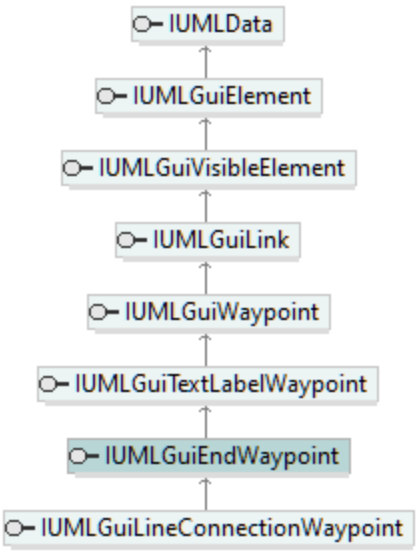
 © 2019-2025 Altova GmbH

Operation **IUMLGuiElement::OwnedGuiElements**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			
documentation	Returns a derived list of all owned Gui elements. All elements in this list are a subtype if IUMLGuiElement 1216.					

16.4.3.6.19 UModelAPI - IUMLGuiEndWaypoint

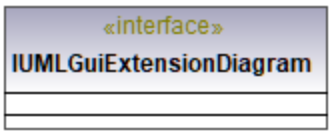
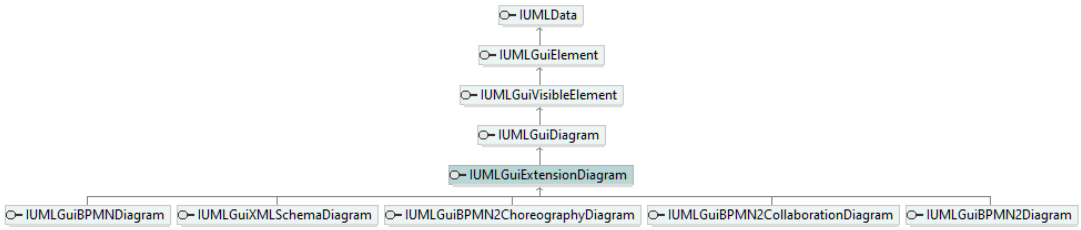
Interface **IUMLGuiEndWaypoint**

diagram	
hierarchy	
documentation	A special waypoint which only occurrs at the end or the begin of a line represented by a IUMLGuiLineLink 1222 .	

© 2019-2025 Altova GmbH

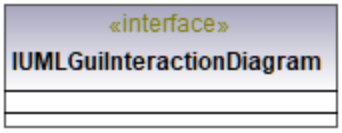
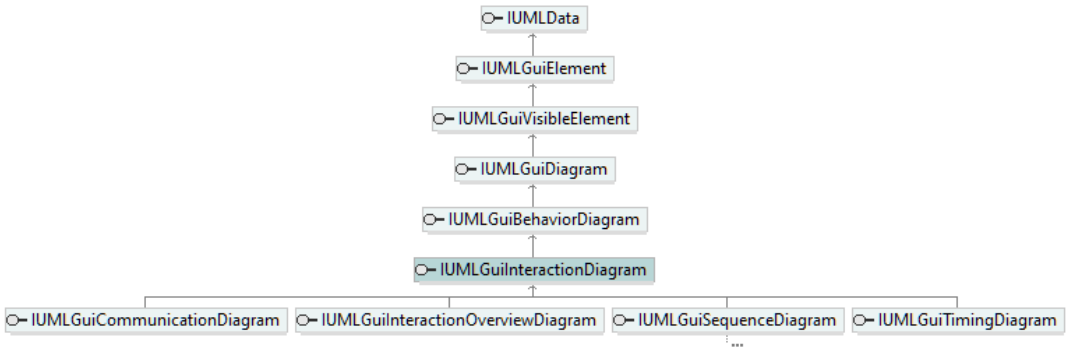
16.4.3.6.20 UModelAPI - IUMLGuiExtensionDiagram

Interface **IUMLGuiExtensionDiagram**

diagram	
hierarchy	
documentation	This diagram type is the base for all UModel specific extension diagrams (for example BPMN diagrams, XML Schema Diagrams) and not a diagram type for itself.

16.4.3.6.21 UModelAPI - IUMLGuiInteractionDiagram

Interface **IUMLGuiInteractionDiagram**

diagram	
hierarchy	

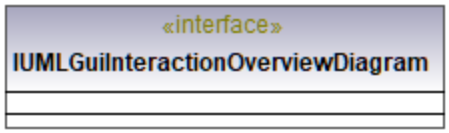
 © 2019-2025 Altova GmbH

UML documentation generated by UModel UML Editor http://www.altova.com/umodel Wed Sep 25 10:33:04 2024

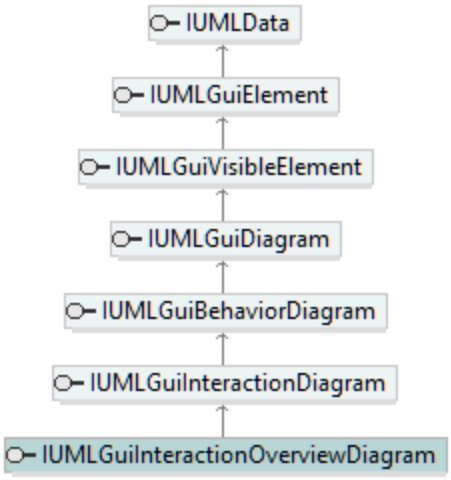
16.4.3.6.22 UModelAPI - IUMLGuiInteractionOverviewDiagram

Interface **IUMLGuiInteractionOverviewDiagram**

diagram



hierarchy

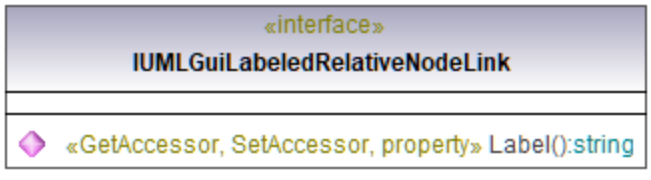


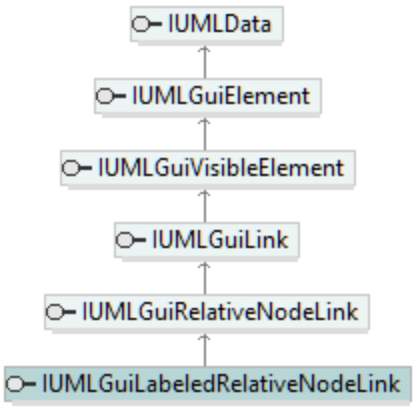
UML documentation generated by UModel UML Editor http://www.altova.com/umodel Wed Sep 25 10:33:04 2024

16.4.3.6.23 UModelAPI - IUMLGuiLabeledRelativeNodeLink

Interface **IUMLGuiLabeledRelativeNodeLink**

diagram



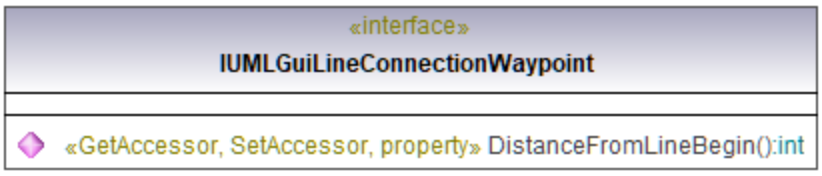
hierarchy	
documentation	This special gui link is used for elements which are relative to another node and have a label, for example for the names of Messages on Communication diagrams.

Operation **IUMLGuiLabeledRelativeNodeLink::Label**

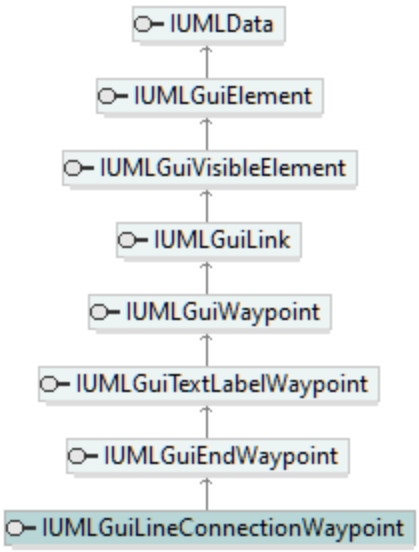
parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**string**			

16.4.3.6.24 UModelAPI - IUMLGuiLineConnectionWaypoint

Interface **IUMLGuiLineConnectionWaypoint**

diagram	

 © 2019-2025 Altova GmbH

hierarchy	
documentation	This special waypoint marks the part of a line where it is connected to another line. For example, when drawing a noteLink from a note to a line on a diagram in UModel, a waypoint of this type is created where the noteLink connects to the target line. Using the DistanceFromLineBegin 1221 property, the waypoint sets a floating fixed position for itself on the line.

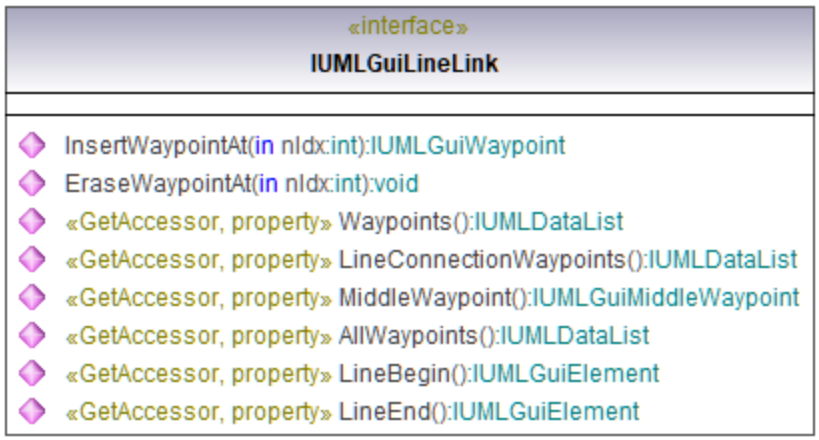
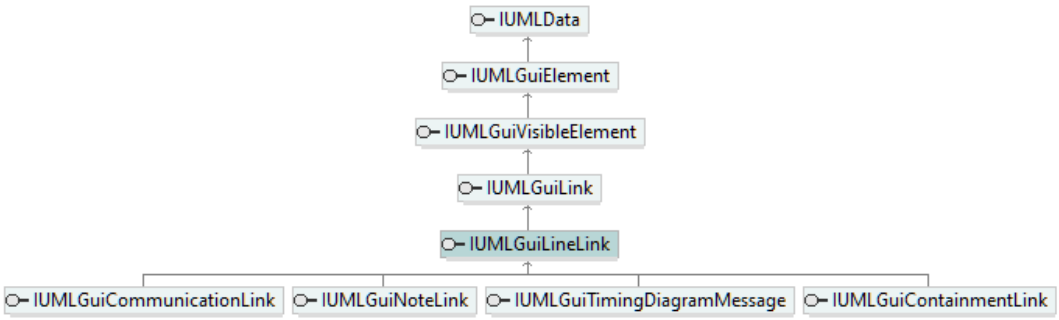
Operation **IUMLGuiLineConnectionWaypoint::DistanceFromLineBegin**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**int**			

© 2019-2025 Altova GmbH

16.4.3.6.25 UModelAPI - IUMLGuiLineLink

Interface **IUMLGuiLineLink**

diagram	
hierarchy	
typedElements	Interface **IUMLDataAll** 911 Interface **IUMLGuiDiagram** 1211 Operation **AddUMLGuiNoteLinkToLine** 914 **AddUMLLineElement** 914 Operation **AddUMLGuiNoteLinkToLine** 1213 **AddUMLLineElement** 1213
documentation	This interface represents a line on a diagram. There are some special lines deriving from this interface available as well. A line is composed of multiple but at least 2 waypoints which are connected to each other, which can be accessed using the AllWaypoints 1222 property. Two of these waypoints are usually of type IUMLGuiEndWaypoint 1217. There may be also a Middlewaypoint accessible with the property MiddleWaypoint 1223 which can be used for example to access textlabels and a LineConnectionWaypoint 1223 when the line is connected to another line, like to a IUMLGuiNoteLink 1229. The LineBegin 1223 property refers the first object and LineEnd 1223 property refers the second graphical object which the line connects.

Operation **IUMLGuiLineLink::AllWaypoints**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			
documentation	A derived list of all waypoints which are part of this line. All elements in this list are of type (or subtype of) IUMLGuiWaypoint 1261.					

© 2019-2025 Altova GmbH

Operation **IUMLGuiLineLink::EraseWaypointAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	**void**			

Operation **IUMLGuiLineLink::InsertWaypointAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	**IUMLGuiWaypoint** 1261			

Operation **IUMLGuiLineLink::LineBegin**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLGuiElement** 1216			
documentation	A reference to the first object, where the line starts.					

Operation **IUMLGuiLineLink::LineConnectionWaypoints**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			
documentation	A list of all waypoints which connect the line with other lines. All elements in this list are of type (or subtype of) IUMLGuiWaypoint 1261.					

Operation **IUMLGuiLineLink::LineEnd**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLGuiElement** 1216			
documentation	A reference to the second object, where the line ends.					

Operation **IUMLGuiLineLink::MiddleWaypoint**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLGuiMiddleWaypoint** 1225			

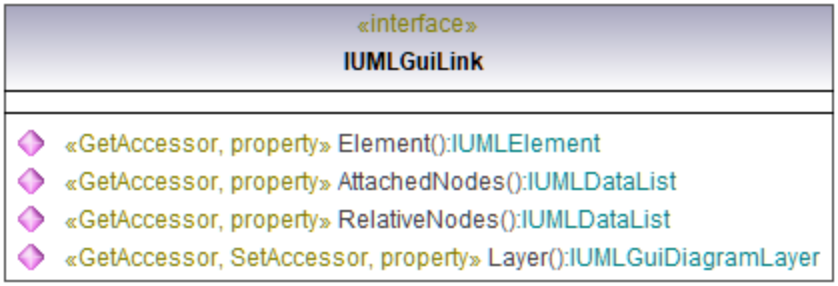
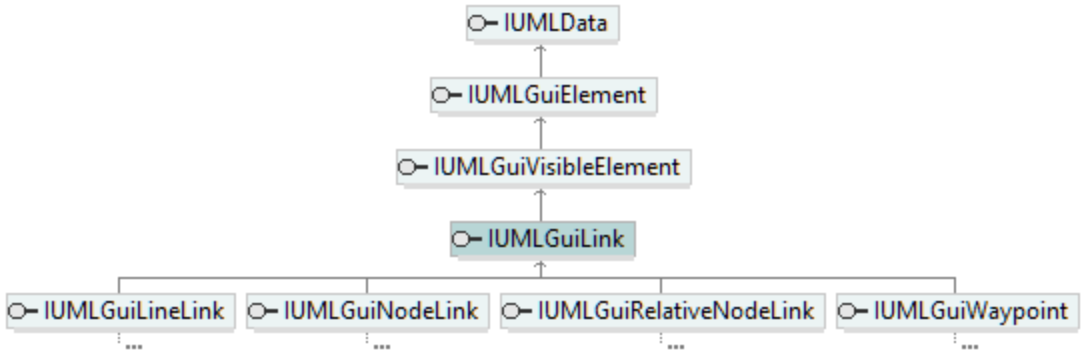
Operation **IUMLGuiLineLink::Waypoints**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			
documentation	A list of all waypoints which form the vertices of this line. All elements in this list are of type (or subtype of) IUMLGuiWaypoint 1261.					

© 2019-2025 Altova GmbH

16.4.3.6.26 UModelAPI - IUMLGuiLink

Interface **IUMLGuiLink**

diagram	
hierarchy	
typed Elements	Interface **IDiagramWindow** 825 Interface **IUMLDataAll** 911 Interface **IUMLGuiAttachedNode** 1201 Interface **IUMLGuiCommunicationDiagram** 1205 Interface **IUMLGuiDiagram** 1211 Operation **FocusedGuiElement** 826 **ScrollToGuiElement** 827 **SelectGuiElement** 827 Operation **AddUMLGuiContainmentLink** 913 **AttachedTo** 916 Operation **AttachedTo** 1201 Operation **AddUMLGuiCommunicationLink** 1206 Operation **AddUMLGuiContainmentLink** 1212
documentation	A GuiLink represents a graphical object on a diagram which is connected to an element from the UML (like a Class, an Interface or a Lifeline). This connected object can be accessed using the Element 1224 property. IUMLGuiLinks further can have attached nodes (IUMLGuiAttachedNode 1201) which appear when necessary, like Tagged Values and are associated with a Layer on the diagram.

Operation **IUMLGuiLink::AttachedNodes**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			
documentation	Returns a derived list of all attached nodes of this element. All elements in this list are of type (or subtype) of IUMLGuiAttachedNode 1201.					

Operation **IUMLGuiLink::Element**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLElement** 1050			

© 2019-2025 Altova GmbH

Operation **IUMLGuiLink::Layer**

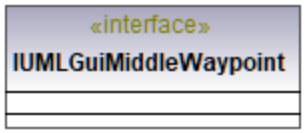
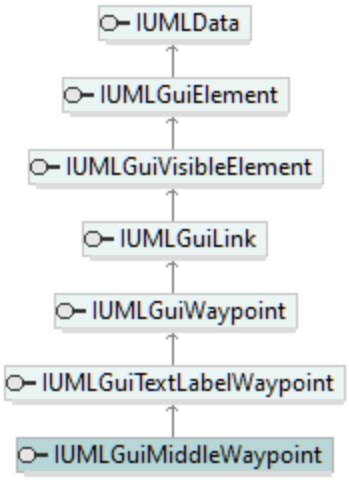
parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLGuiDiagramLayer** 1215			

Operation **IUMLGuiLink::RelativeNodes**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			
documentation	Returns a list of relative nodes to this gui link. The list contains only elements of type (or subtype of) IUMLGuiRelativeNodeLink 1233.					

16.4.3.6.27 UModelAPI - IUMLGuiMiddleWaypoint

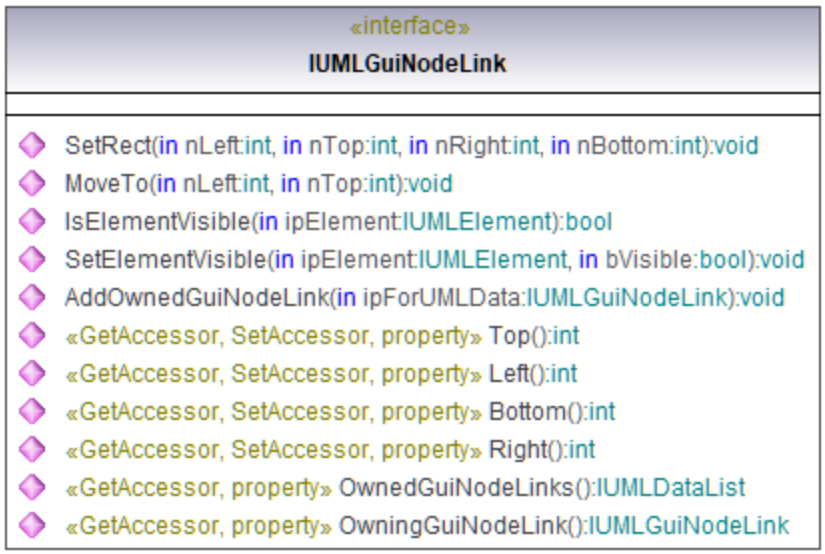
Interface **IUMLGuiMiddleWaypoint**

diagram	
hierarchy	
typedElements	Interface **IUMLDataAll** 911 Interface **IUMLGuiLineLink** 1222 Operation **MiddleWaypoint** 953 Operation **MiddleWaypoint** 1223
documentation	A middle waypoint is a special waypoint on a line (IUMLGuiLineLink 1222) which appears in the center of the line and can have text labels attached to it.

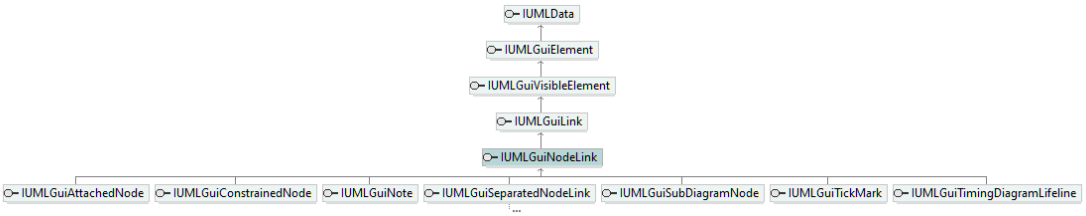
16.4.3.6.28 UModelAPI - IUMLGuiNodeLink

Interface **IUMLGuiNodeLink**

diagram



hierarchy



typedElements

Interface **IUMLDataAll** 911

Interface **IUMLGuiDiagram** 1211

Interface **IUMLGuiNodeLink** 1226

Operation **AddOwnedGuiNodeLink** 913
AddUMLElement 913
AddUMLGuiNodeLink 913
AddUMLGuiNoteLink 913
AddUMLLineElement 914
OwningGuiNodeLink 959

Operation **AddUMLElement** 1212
AddUMLGuiNodeLink 1212
AddUMLGuiNoteLink 1213
AddUMLLineElement 1213

Operation **AddOwnedGuiNodeLink** 1227
OwningGuiNodeLink 1227

documentation

A GuiNodeLink represents a graphical object on a diagram which usually represents an element from the UML (for example a Class, an Interface or a Lifeline). It has a position defined by a rectangle which can be positioned freely on the diagram.
A GuiNodeLink can itself contain other GuiNodeLinks, for example when displaying a big state in a state machine diagam which contains other, smaller substates.

 © 2019-2025 Altova GmbH

	Additionally, if the GuiNodeLink displays cells on it, like for example operations and properties on a class, it can store for each element shown if the element should be visible or not. Use the SetElementVisible[1227] and IsElementVisible[1227] functions for this.

Operation **IUMLGuiNodeLink::AddOwnedGuiNodeLink**

parameter	name	direction	type	type modifier	multiplicity	default
	ipForUMLData	**in**	**IUMLGuiNodeLink**[1226]			
	return	**return**	**void**			

Operation **IUMLGuiNodeLink::Bottom**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**int**			

Operation **IUMLGuiNodeLink::IsElementVisible**

parameter	name	direction	type	type modifier	multiplicity	default
	ipElement	**in**	**IUMLElement**[1050]			
	return	**return**	**bool**			

Operation **IUMLGuiNodeLink::Left**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**int**			

Operation **IUMLGuiNodeLink::MoveTo**

parameter	name	direction	type	type modifier	multiplicity	default
	nLeft	**in**	**int**			
	nTop	**in**	**int**			
	return	**return**	**void**			

Operation **IUMLGuiNodeLink::OwnedGuiNodeLinks**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList**[906]			
documentation	Returns a list of all owned gui node links, all nodes which are directly contained in this node. All elements in this list are of type (or subtype of) IUMLGuiLink[1224].					

Operation **IUMLGuiNodeLink::OwningGuiNodeLink**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLGuiNodeLink**[1226]			

Operation **IUMLGuiNodeLink::Right**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**int**			

Operation **IUMLGuiNodeLink::SetElementVisible**

parameter	name	direction	type	type modifier	multiplicity	default
	ipElement	**in**	**IUMLElement**[1050]			
	bVisible	**in**	**bool**			

	return	**return**	**void**

Operation **IUMLGuiNodeLink::SetRect**

parameter	name	direction	type	type modifier	multiplicity	default
	nLeft	**in**	**int**			
	nTop	**in**	**int**			
	nRight	**in**	**int**			
	nBottom	**in**	**int**			
	return	**return**	**void**			

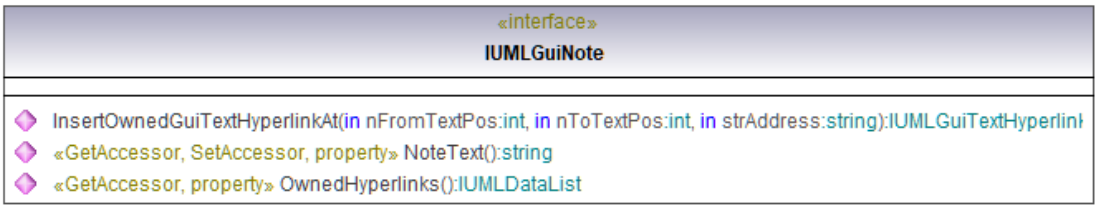
Operation **IUMLGuiNodeLink::Top**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**int**			

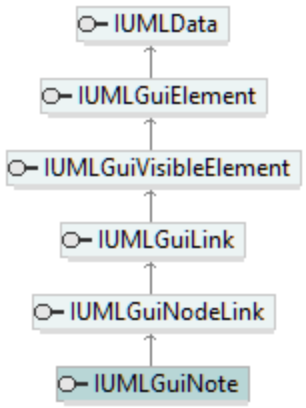
16.4.3.6.29 UModelAPI - IUMLGuiNote

Interface **IUMLGuiNote**

diagram



hierarchy



typedElements

Interface **IUMLDataAll** 911

Interface **IUMLGuiDiagram** 1211

Operation **AddUMLGuiNote** 913
AddUMLGuiNoteLink 913
AddUMLGuiNoteLinkToLine 914

Operation **AddUMLGuiNote** 1213
AddUMLGuiNoteLink 1213

 © 2019-2025 Altova GmbH

	AddUMLGuiNoteLinkToLine 1213
documentation	A IUMLGuiNote 1228 is the graphical object resembling a note on UModel diagrams displaying a text comment. It provides access to the note text and a list of hyperlinks in this text. These hyperlinks are nothing more than a list of URLs together with an begin and end number referencing positions in the text. Any text between a such a begin and end position is displayed as hyperlink and triggers UModel to open the URL when clicked.

Operation **IUMLGuiNote::InsertOwnedGuiTextHyperlinkAt**

parameter	name	direction	type	type modifier	multiplicity	default
	nFromTextPos	**in**	**int**			
	nToTextPos	**in**	**int**			
	strAddress	**in**	**string**			
	return	**return**	**IUMLGuiTextHyperlink** 1250			

Operation **IUMLGuiNote::NoteText**

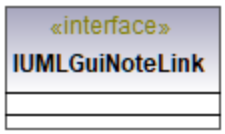
parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**string**			

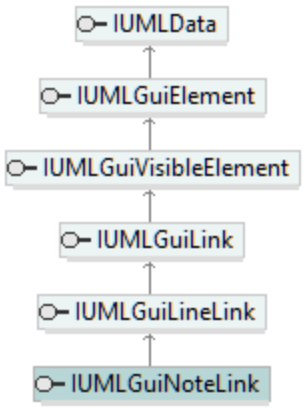
Operation **IUMLGuiNote::OwnedHyperlinks**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			

16.4.3.6.30 UModelAPI - IUMLGuiNoteLink

Interface **IUMLGuiNoteLink**

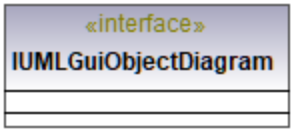
diagram	

hierarchy	
typedElements	Interface **IUMLDataAll** 911 Interface **IUMLGuiDiagram** 1211 Operation **AddUMLGuiNoteLink** 913 **AddUMLGuiNoteLinkToLine** 914 Operation **AddUMLGuiNoteLink** 1213 **AddUMLGuiNoteLinkToLine** 1213
documentation	A notelink is a special IUMLGuiLineLink 1222 which connects a IUMLGuiNote 1228 with another IUMLGuiLink 1224 element. It is displayed as a dotted line.

UML documentation generated by **UModel** UML Editor **http://www.altova.com/umodel** Wed Sep 25 10:33:04 2024

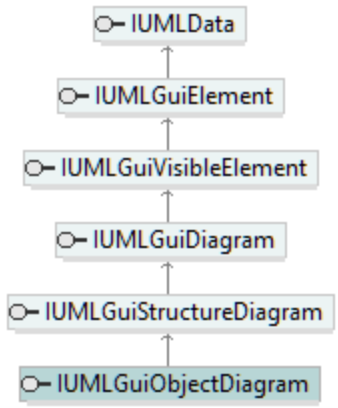
16.4.3.6.31 UModelAPI - IUMLGuiObjectDiagram

Interface **IUMLGuiObjectDiagram**

diagram	

 © 2019-2025 Altova GmbH

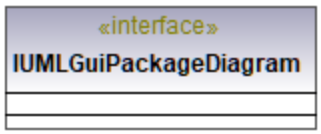
hierarchy



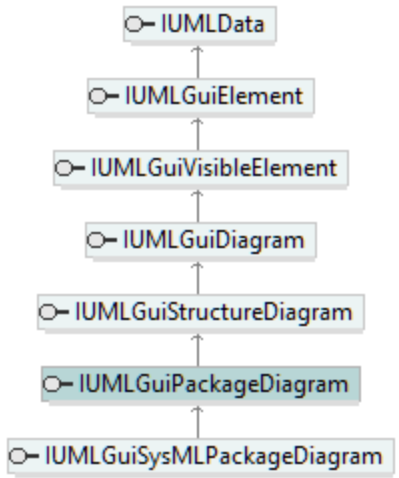
16.4.3.6.32 UModelAPI - IUMLGuiPackageDiagram

Interface **IUMLGuiPackageDiagram**

diagram



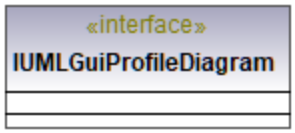
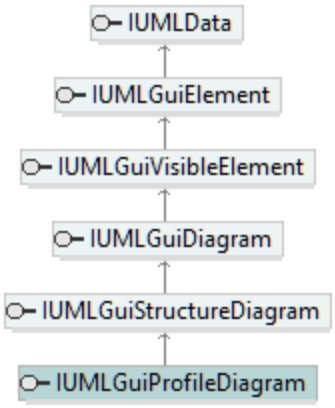
hierarchy



UML documentation generated by UModel UML Editor http://www.altova.com/umodel Wed Sep 25 10:33:04 2024

16.4.3.6.33 UModelAPI - IUMLGuiProfileDiagram

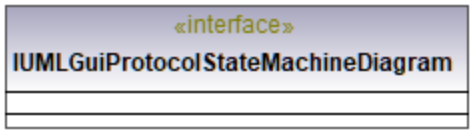
Interface **IUMLGuiProfileDiagram**

diagram	
hierarchy	

UML documentation generated by UModel UML Editor http://www.altova.com/umodel Wed Sep 25 10:33:04 2024

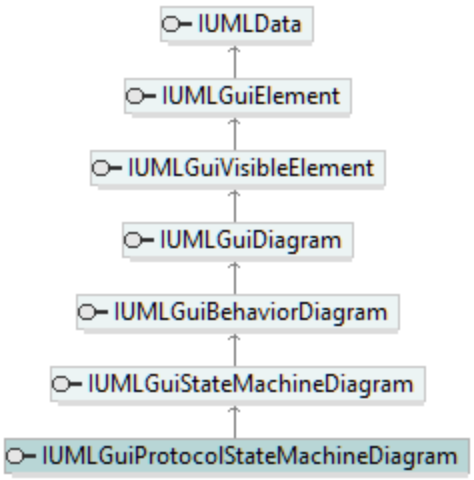
16.4.3.6.34 UModelAPI - IUMLGuiProtocolStateMachineDiagram

Interface **IUMLGuiProtocolStateMachineDiagram**

diagram	

 © 2019-2025 Altova GmbH

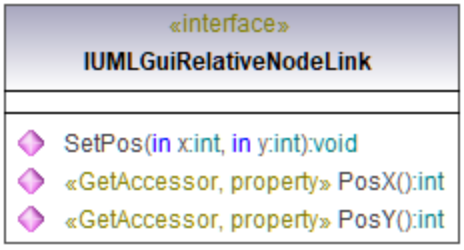
hierarchy



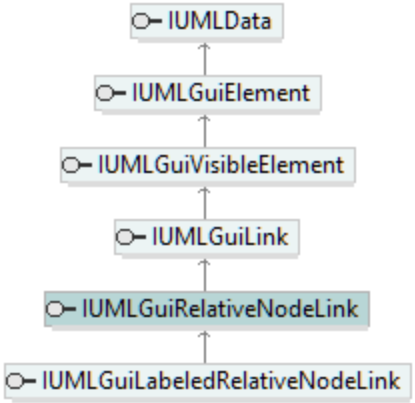
16.4.3.6.35 UModelAPI - IUMLGuiRelativeNodeLink

Interface **IUMLGuiRelativeNodeLink**

diagram



hierarchy



documentation	This gui link is used for elements which are positioned relative to another node. For example the names of Messages on Communication diagrams use a specialization of this interface.

Operation **IUMLGuiRelativeNodeLink::PosX**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**int**			

Operation **IUMLGuiRelativeNodeLink::PosY**

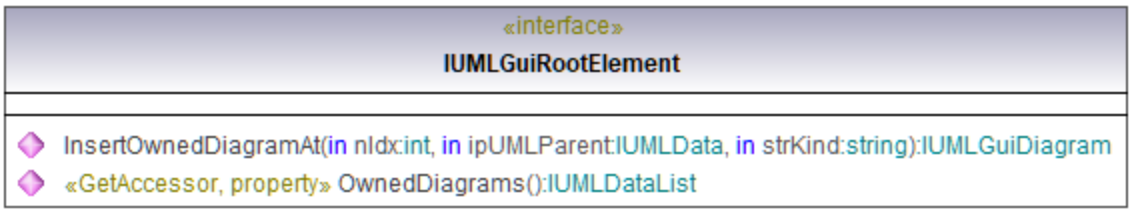
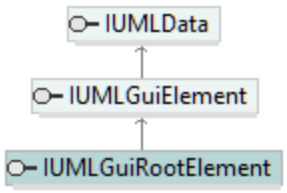
parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**int**			

Operation **IUMLGuiRelativeNodeLink::SetPos**

parameter	name	direction	type	type modifier	multiplicity	default
	x	**in**	**int**			
	y	**in**	**int**			
	return	**return**	**void**			

16.4.3.6.36 UModelAPI - IUMLGuiRootElement

Interface **IUMLGuiRootElement**

diagram	
hierarchy	
typed Elements	Interface **IDocument** 832 Operation **GuiRoot** 835
documentation	This is the root interface for all graphical objects and contains all diagrams which exists in the UModel project.

Operation **IUMLGuiRootElement::InsertOwnedDiagramAt**

 © 2019-2025 Altova GmbH

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	ipUMLParent	**in**	**IUMLData** 904			
	strKind	**in**	**string**			
	return	**return**	**IUMLGuiDiagram** 1211			

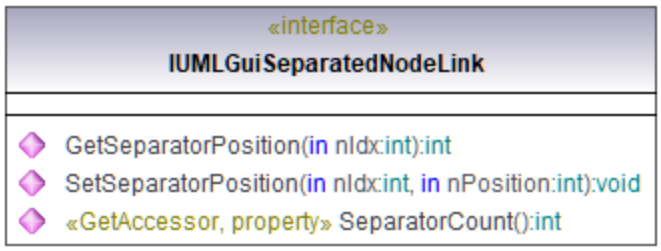
Operation **IUMLGuiRootElement::OwnedDiagrams**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			
documentation	Returns a list of all diagrams in this UModel project. All elements in this list are of type (or subtype of) IUMLGuiDiagram 1211.					

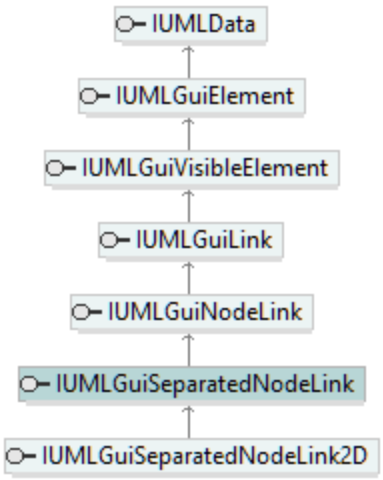
16.4.3.6.37 UModelAPI - IUMLGuiSeparatedNodeLink

Interface **IUMLGuiSeparatedNodeLink**

diagram



hierarchy



documentation	This node link represents a graphical object on a UModel diagram which can be separated into two or more parts by one or more either horizontal or vertical lines. For each line, the position of the separator is stored in this node.

	This node type is used for example by CombinedFragments, ActivityPartitions and States with regions.

Operation **IUMLGuiSeparatedNodeLink::GetSeparatorPosition**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	**int**			

Operation **IUMLGuiSeparatedNodeLink::SeparatorCount**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**int**			

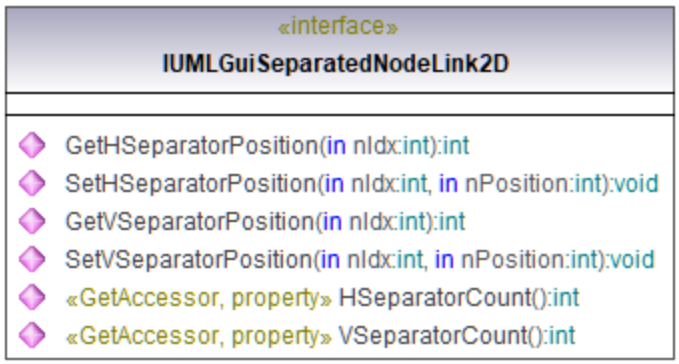
Operation **IUMLGuiSeparatedNodeLink::SetSeparatorPosition**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	nPosition	**in**	**int**			
	return	**return**	**void**			

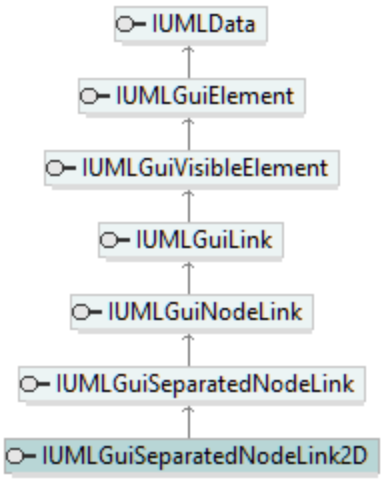
16.4.3.6.38 UModelAPI - IUMLGuiSeparatedNodeLink2D

Interface **IUMLGuiSeparatedNodeLink2D**

diagram



 © 2019-2025 Altova GmbH

hierarchy	
documentation	This node link represents a graphical object on a UModel diagram which can be separated into parts by one or more horizontal or vertical lines, but in contrast to IUMLGuiSeparatedNodeLink[1235], the node can be subdivided vertically and horizontally at the same time. For each vertical or horizontal separation line, the position of the separator is stored in this node. This node type is used for example by ActivityPartitions.

Operation **IUMLGuiSeparatedNodeLink2D::GetHSeparatorPosition**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	**int**			

Operation **IUMLGuiSeparatedNodeLink2D::GetVSeparatorPosition**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	**int**			

Operation **IUMLGuiSeparatedNodeLink2D::HSeparatorCount**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**int**			

Operation **IUMLGuiSeparatedNodeLink2D::SetHSeparatorPosition**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	nPosition	**in**	**int**			
	return	**return**	**void**			

Operation **IUMLGuiSeparatedNodeLink2D::SetVSeparatorPosition**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	nPosition	**in**	**int**			
	return	**return**	**void**			

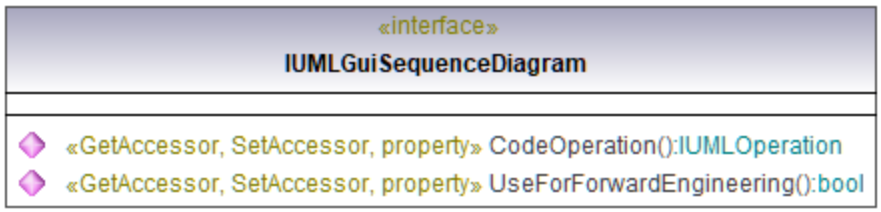
Operation **IUMLGuiSeparatedNodeLink2D::VSeparatorCount**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**int**			

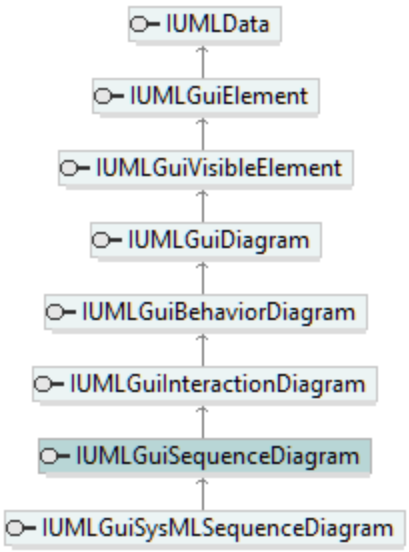
16.4.3.6.39 UModelAPI - IUMLGuiSequenceDiagram

Interface **IUMLGuiSequenceDiagram**

diagram



hierarchy



Operation **IUMLGuiSequenceDiagram::CodeOperation**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLOperation** 1131			

Operation **IUMLGuiSequenceDiagram::UseForForwardEngineering**

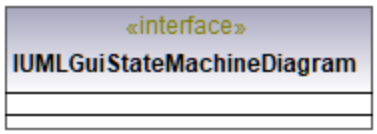
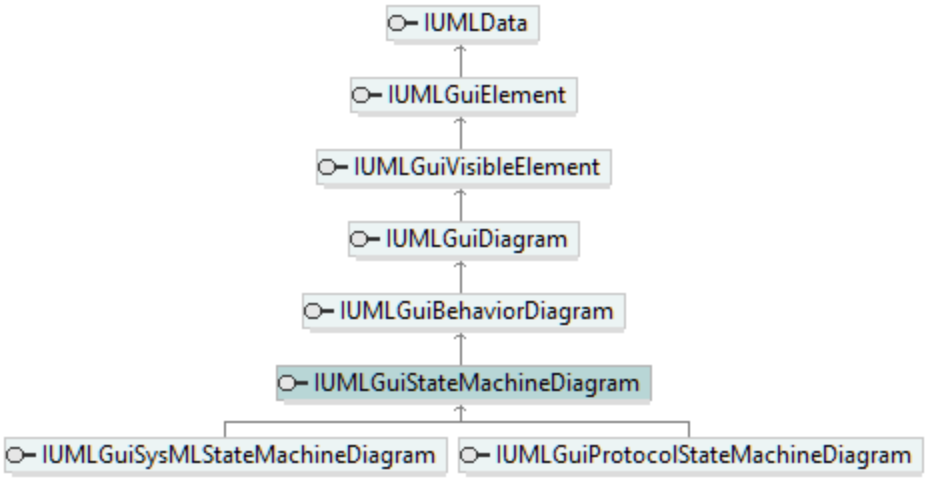
parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

 © 2019-2025 Altova GmbH

UML documentation generated by UModel UML Editor http://www.altova.com/umodel Wed Sep 25 10:33:04 2024

16.4.3.6.40 UModelAPI - IUMLGuiStateMachineDiagram

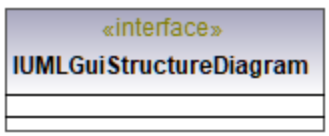
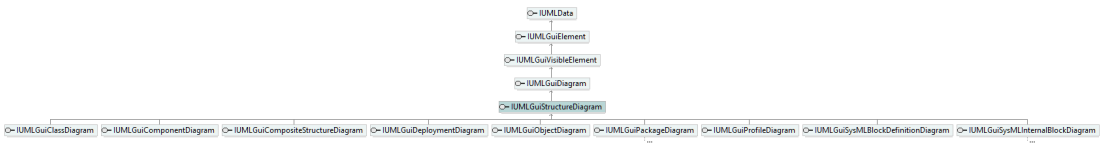
Interface **IUMLGuiStateMachineDiagram**

diagram	
hierarchy	

UML documentation generated by UModel UML Editor http://www.altova.com/umodel Wed Sep 25 10:33:04 2024

16.4.3.6.41 UModelAPI - IUMLGuiStructureDiagram

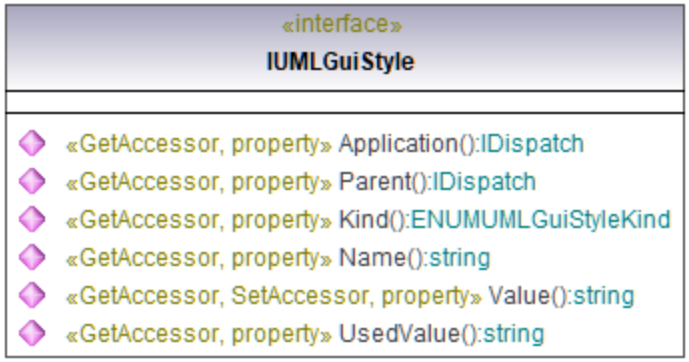
Interface **IUMLGuiStructureDiagram**

diagram	
hierarchy	

UML documentation generated by UModel UML Editor http://www.altova.com/umodel Wed Sep 25 10:33:04 2024

16.4.3.6.42 UModelAPI - IUMLGuiStyle

Interface **IUMLGuiStyle**

diagram	
typedElements	Interface IUMLGuiStyles 1241 Operation GetStyle 1241 Item 1242

Operation **IUMLGuiStyle::Application**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IDispatch**			

Operation **IUMLGuiStyle::Kind**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	ENUMUMLGuiStyleKind 1266			

Operation **IUMLGuiStyle::Name**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**string**			

Operation **IUMLGuiStyle::Parent**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IDispatch**			

Operation **IUMLGuiStyle::UsedValue**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**string**			

Operation **IUMLGuiStyle::Value**

parameter	name	direction	type	type modifier	multiplicity	default

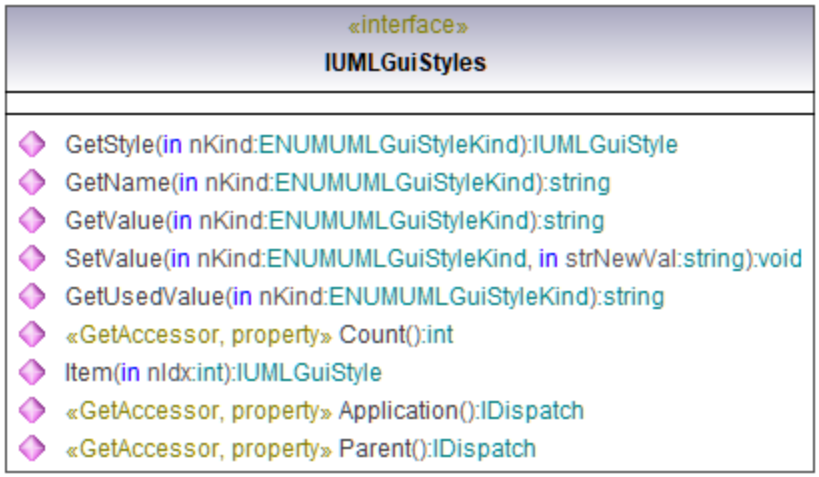
 © 2019-2025 Altova GmbH

	return	**return**	**string**

16.4.3.6.43 UModelAPI - IUMLGuiStyles

Interface **IUMLGuiStyles**

diagram



typedElements

Interface **IDocument** 832	Operation **ElementFamilyStyles** 833, **LineStyles** 836, **NodeStyles** 837, **ProjectStyles** 837
Interface **IUMLDataAll** 911	Operation **StereotypedElementStyles** 974, **Styles** 974
Interface **IUMLGuiVisibleElement** 1260	Operation **Styles** 1260
Interface **IUMLStereotype** 1169	Operation **StereotypedElementStyles** 1170

Operation **IUMLGuiStyles::Application**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IDispatch**			

Operation **IUMLGuiStyles::Count**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**int**			

Operation **IUMLGuiStyles::GetName**

parameter	name	direction	type	type modifier	multiplicity	default
	nKind	**in**	**ENUMUMLGuiStyleKind** 1266			
	return	**return**	**string**			

Operation **IUMLGuiStyles::GetStyle**

parameter	name	direction	type	type modifier	multiplicity	default
	nKind	**in**	**ENUMUMLGuiStyleKind** 1266			
	return	**return**	**IUMLGuiStyle** 1240			

Operation **IUMLGuiStyles::GetUsedValue**

parameter	name	direction	type	type modifier	multiplicity	default
	nKind	**in**	**ENUMUMLGuiStyleKind** 1266			
	return	**return**	**string**			

Operation **IUMLGuiStyles::GetValue**

parameter	name	direction	type	type modifier	multiplicity	default
	nKind	**in**	**ENUMUMLGuiStyleKind** 1266			
	return	**return**	**string**			

Operation **IUMLGuiStyles::Item**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	**IUMLGuiStyle** 1240			

Operation **IUMLGuiStyles::Parent**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IDispatch**			

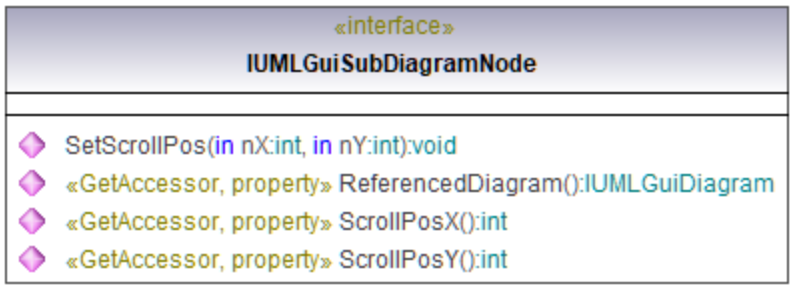
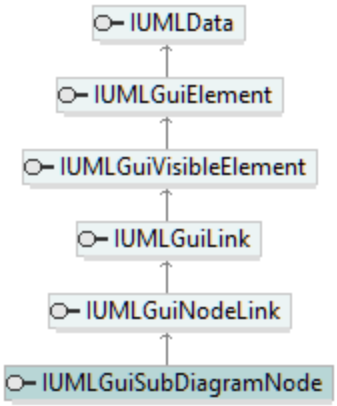
Operation **IUMLGuiStyles::SetValue**

parameter	name	direction	type	type modifier	multiplicity	default
	nKind	**in**	**ENUMUMLGuiStyleKind** 1266			
	strNewVal	**in**	**string**			
	return	**return**	**void**			

© 2019-2025 Altova GmbH

16.4.3.6.44 UModelAPI - IUMLGuiSubDiagramNode

Interface **IUMLGuiSubDiagramNode**

diagram	
hierarchy	
documentation	The sub diagram node represents a node link on a diagram which again includes another diagram. This is used for example on interaction overview diagrams to display sequence, communication and timing diagrams inside nodes. The property ReferencedDiagram 1243 controls the diagrams which is shown inside the node.

Operation **IUMLGuiSubDiagramNode::ReferencedDiagram**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLGuiDiagram** 1211			

Operation **IUMLGuiSubDiagramNode::ScrollPosX**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**int**			

Operation **IUMLGuiSubDiagramNode::ScrollPosY**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**int**			

Operation **IUMLGuiSubDiagramNode::SetScrollPos**

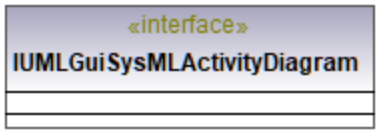
parameter	name	direction	type	type modifier	multiplicity	default

	nX	**in**	**int**
	nY	**in**	**int**
	return	**return**	**void**

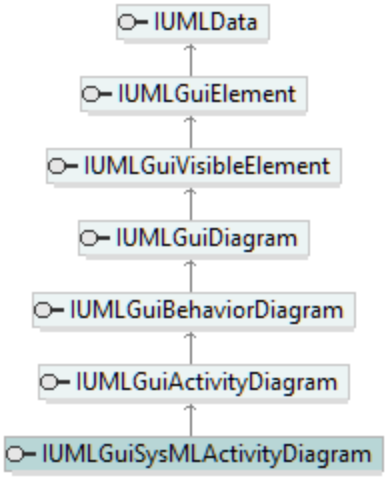
16.4.3.6.45 UModelAPI - IUMLGuiSysMLActivityDiagram

Interface **IUMLGuiSysMLActivityDiagram**

diagram



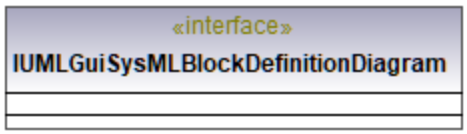
hierarchy



16.4.3.6.46 UModelAPI - IUMLGuiSysMLBlockDefinitionDiagram

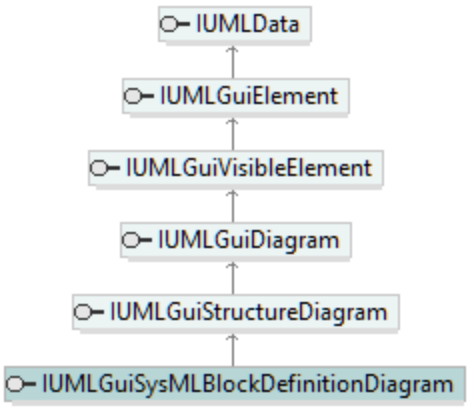
Interface **IUMLGuiSysMLBlockDefinitionDiagram**

diagram



 © 2019-2025 Altova GmbH

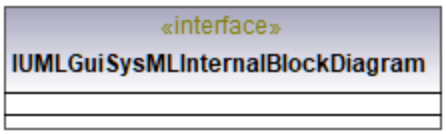
hierarchy



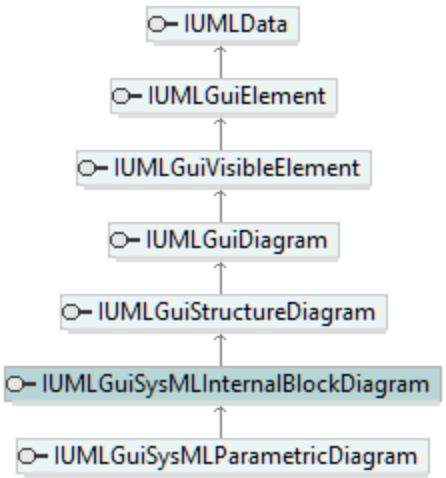
16.4.3.6.47 UModelAPI - IUMLGuiSysMLInternalBlockDiagram

Interface **IUMLGuiSysMLInternalBlockDiagram**

diagram



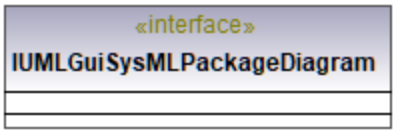
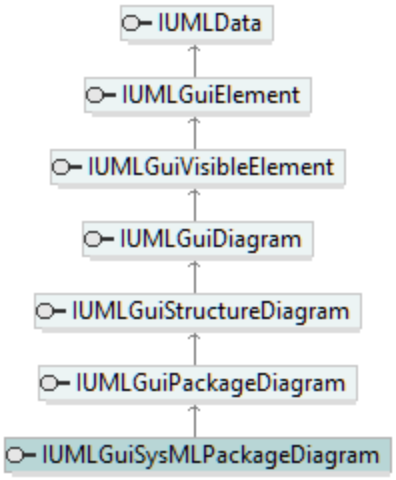
hierarchy



UML documentation generated by UModel UML Editor http://www.altova.com/umodel Wed Sep 25 10:33:04 2024

16.4.3.6.48 UModelAPI - IUMLGuiSysMLPackageDiagram

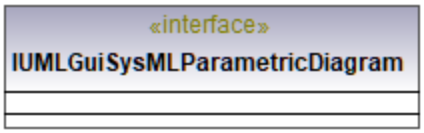
Interface **IUMLGuiSysMLPackageDiagram**

diagram	
hierarchy	

UML documentation generated by UModel UML Editor http://www.altova.com/umodel Wed Sep 25 10:33:04 2024

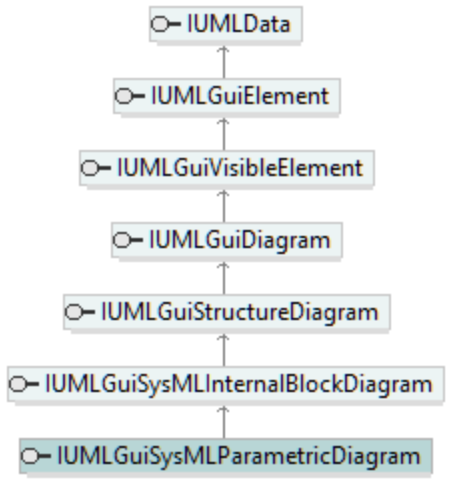
16.4.3.6.49 UModelAPI - IUMLGuiSysMLParametricDiagram

Interface **IUMLGuiSysMLParametricDiagram**

diagram	

 © 2019-2025 Altova GmbH

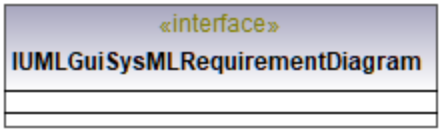
hierarchy



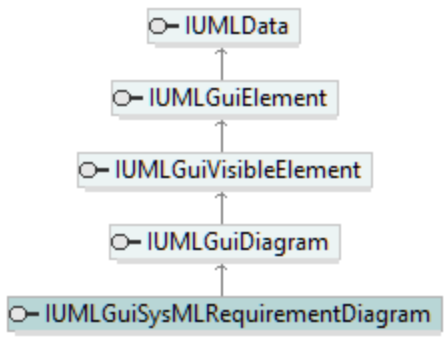
16.4.3.6.50 UModelAPI - IUMLGuiSysMLRequirementDiagram

Interface **IUMLGuiSysMLRequirementDiagram**

diagram



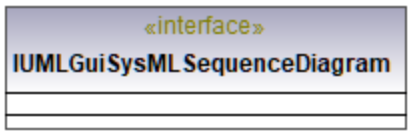
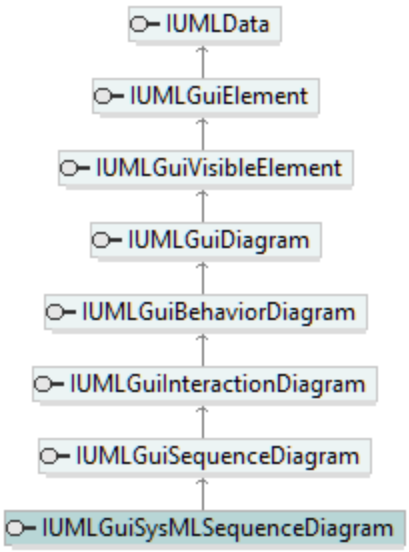
hierarchy



© 2019-2025 Altova GmbH

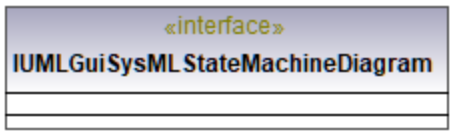
16.4.3.6.51 UModelAPI - IUMLGuiSysMLSequenceDiagram

Interface **IUMLGuiSysMLSequenceDiagram**

diagram	
hierarchy	

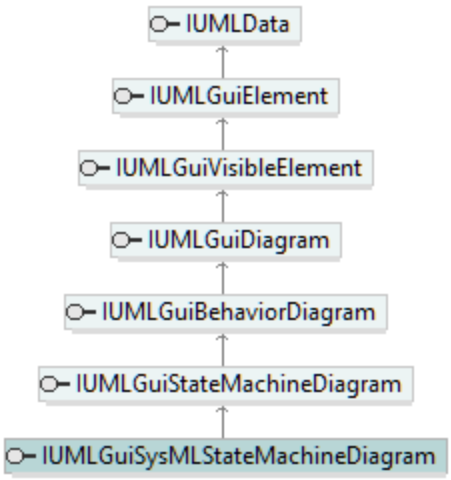
16.4.3.6.52 UModelAPI - IUMLGuiSysMLStateMachineDiagram

Interface **IUMLGuiSysMLStateMachineDiagram**

diagram	

 © 2019-2025 Altova GmbH

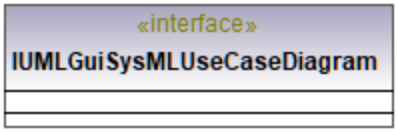
hierarchy



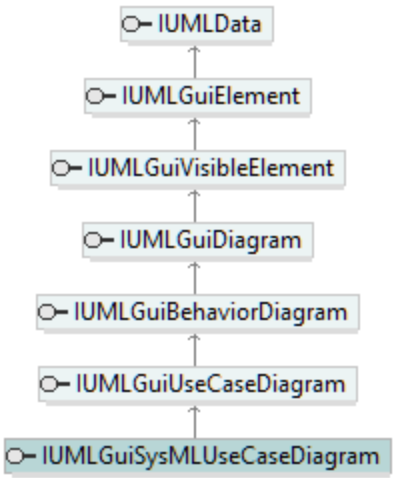
16.4.3.6.53 UModelAPI - IUMLGuiSysMLUseCaseDiagram

Interface **IUMLGuiSysMLUseCaseDiagram**

diagram



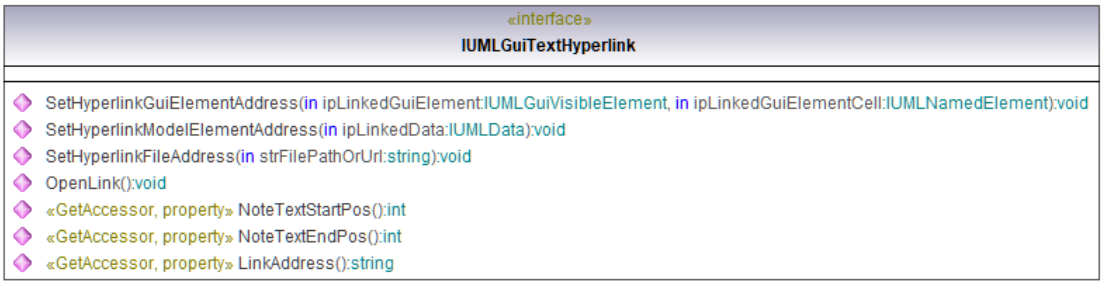
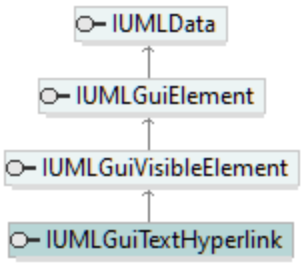
hierarchy



UML documentation generated by UModel UML Editor http://www.altova.com/umodel Wed Sep 25 10:33:04 2024

16.4.3.6.54 UModelAPI - IUMLGuiTextHyperlink

Interface **IUMLGuiTextHyperlink**

diagram	
hierarchy	
typedElements	Interface IUMLDataAll 911 Interface IUMLGuiDiagram 1211 Interface IUMLGuiNote 1228 Operation InsertOwnedGuiTextHyperlinkAt 939 Operation InsertOwnedGuiTextHyperlinkAt 1214 Operation InsertOwnedGuiTextHyperlinkAt 1229
documentation	Text Hyperlinks store an URL together with a begin and an end number referencing positions in some text. Any text between a such a begin and end position is displayed as hyperlink and triggers UModel to open the URL when clicked. This is used in IUMLGuiNote 1228 to create hyperlinks inside of the text comment for example.

Operation **IUMLGuiTextHyperlink::LinkAddress**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**string**			

Operation **IUMLGuiTextHyperlink::NoteTextEndPos**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**int**			

Operation **IUMLGuiTextHyperlink::NoteTextStartPos**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**int**			

Operation **IUMLGuiTextHyperlink::OpenLink**

 © 2019-2025 Altova GmbH

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**void**			

Operation **IUMLGuiTextHyperlink::SetHyperlinkFileAddress**

parameter	name	direction	type	type modifier	multiplicity	default
	strFilePathOrUrl	**in**	**string**			
	return	**return**	**void**			

Operation **IUMLGuiTextHyperlink::SetHyperlinkGuiElementAddress**

parameter	name	direction	type	type modifier	multiplicity	default
	ipLinkedGuiElemein nt		**IUMLGuiVisibleEl ement** 1260			
	ipLinkedGuiElemein ntCell		**IUMLNamedElem ent** 1117			
	return	**return**	**void**			

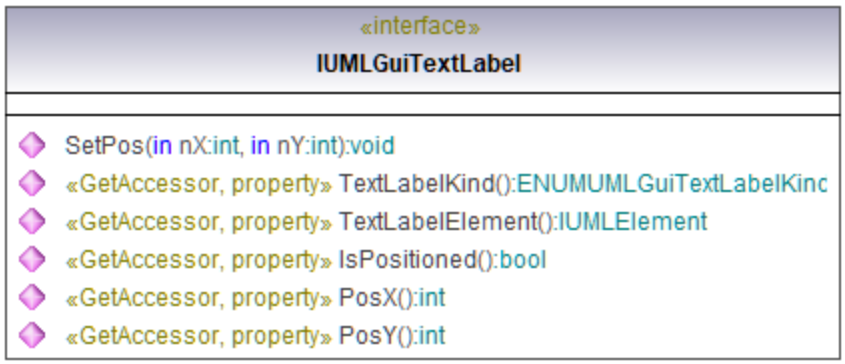
Operation **IUMLGuiTextHyperlink::SetHyperlinkModelElementAddress**

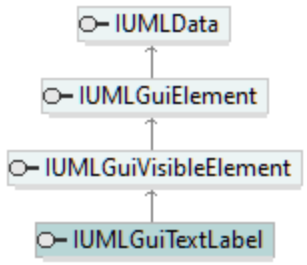
parameter	name	direction	type	type modifier	multiplicity	default
	ipLinkedData	**in**	**IUMLData** 904			
	return	**return**	**void**			

16.4.3.6.55 UModelAPI - IUMLGuiTextLabel

Interface **IUMLGuiTextLabel**

diagram



hierarchy	
typedElements	Interface **IUMLDataAll** 911 Operation **GetTextLabelText** 929 **IsTextLabelVisible** 950 **SetTextLabelVisible** 972 Interface **IUMLGuiTextLabelWaypoint** 1253 Operation **GetTextLabelText** 1253 **IsTextLabelVisible** 1253 **SetTextLabelVisible** 1254
documentation	A text label is an graphical object displaying additional data at the begin, end or at the center of a line. The IUMLGuiTextLabel 1251 interface provides access to this element. The text label can reference an UML Element using the TextLabelElement 1252 property and has a TextLabelKind 1252 which affects what text is shown in the text label.

Operation **IUMLGuiTextLabel::IsPositioned**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IUMLGuiTextLabel::PosX**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**int**			

Operation **IUMLGuiTextLabel::PosY**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**int**			

Operation **IUMLGuiTextLabel::SetPos**

parameter	name	direction	type	type modifier	multiplicity	default
	nX	**in**	**int**			
	nY	**in**	**int**			
	return	**return**	**void**			

Operation **IUMLGuiTextLabel::TextLabelElement**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLElement** 1050			

Operation **IUMLGuiTextLabel::TextLabelKind**

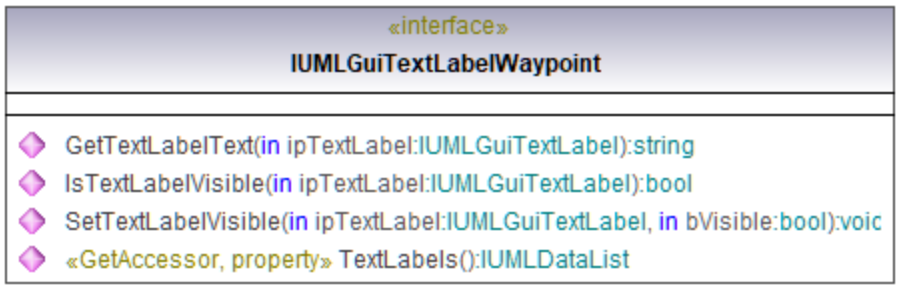
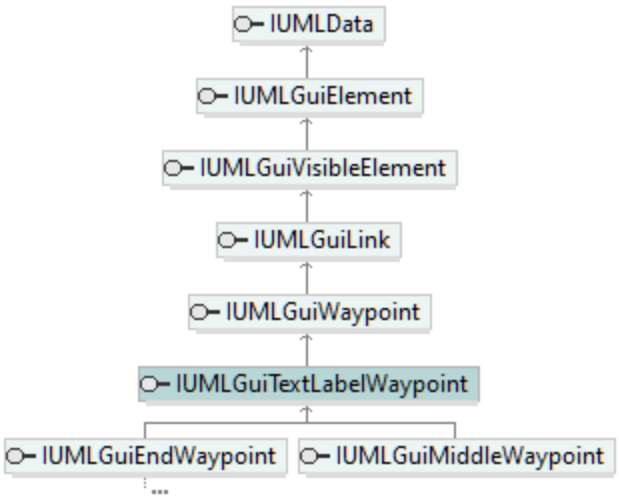
parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**ENUMUMLGuiTextLabelKind** 1268			

© 2019-2025 Altova GmbH

UML documentation generated by **UModel** UML Editor **http://www.altova.com/umodel** Wed Sep 25 10:33:04 2024

16.4.3.6.56 UModelAPI - IUMLGuiTextLabelWaypoint

Interface **IUMLGuiTextLabelWaypoint**

diagram	
hierarchy	
documentation	A text label waypoint is a special waypoint as part of a IUMLGuiLineLink[1222] which can have one or more text labels associated with it. Text label waypoints can usually appear at the begin, end or in the middle of a line. The waypoint stores not only the text labels it shows, but also the visibility of each text label.

Operation **IUMLGuiTextLabelWaypoint::GetTextLabelText**

parameter	name	direction	type	type modifier	multiplicity	default
	ipTextLabel	**in**	**IUMLGuiTextLabel**[1251]			
	return	**return**	**string**			

Operation **IUMLGuiTextLabelWaypoint::IsTextLabelVisible**

parameter	name	direction	type	type modifier	multiplicity	default
	ipTextLabel	**in**	**IUMLGuiTextLabel**[1251]			

	return	**return**	**bool**

Operation **IUMLGuiTextLabelWaypoint::SetTextLabelVisible**

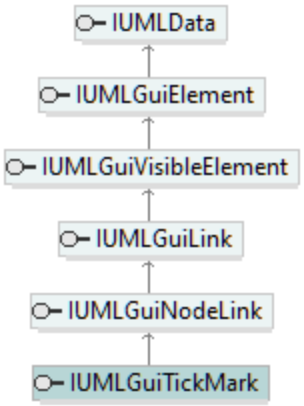
parameter	name	direction	type	type modifier	multiplicity	default
	ipTextLabel	**in**	**IUMLGuiTextLabel** 1251			
	bVisible	**in**	**bool**			
	return	**return**	**void**			

Operation **IUMLGuiTextLabelWaypoint::TextLabels**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			
documentation	Returns a list of all text labels of this waypoint. Contains only elements of type (or subtype of) IUMLGuiTextLabel 1251.					

16.4.3.6.57 UModelAPI - IUMLGuiTickMark

Interface **IUMLGuiTickMark**

diagram	
hierarchy	
typedElements	Interface **IUMLGuiTimingDiagram** 1255 Operation **AddUMLGuiTickMark** 1255
documentation	A tick mark is a special graphical item appearing on the border of lifelines on timing diagrams. It represents a certain point in time and is displayed as short vertical line.

 © 2019-2025 Altova GmbH

	It has a Value[1255] property which is displayed as text below the vertical line.

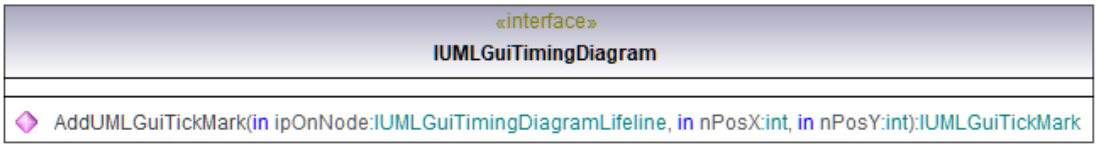
Operation **IUMLGuiTickMark::Value**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**string**			

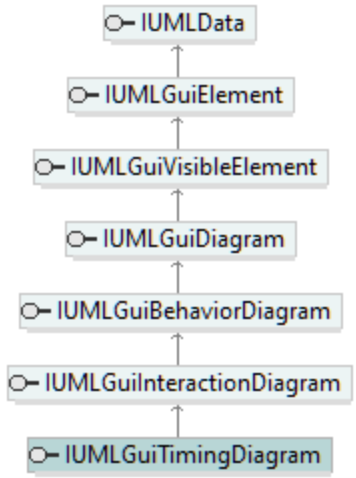
16.4.3.6.58 UModelAPI - IUMLGuiTimingDiagram

Interface **IUMLGuiTimingDiagram**

diagram



hierarchy



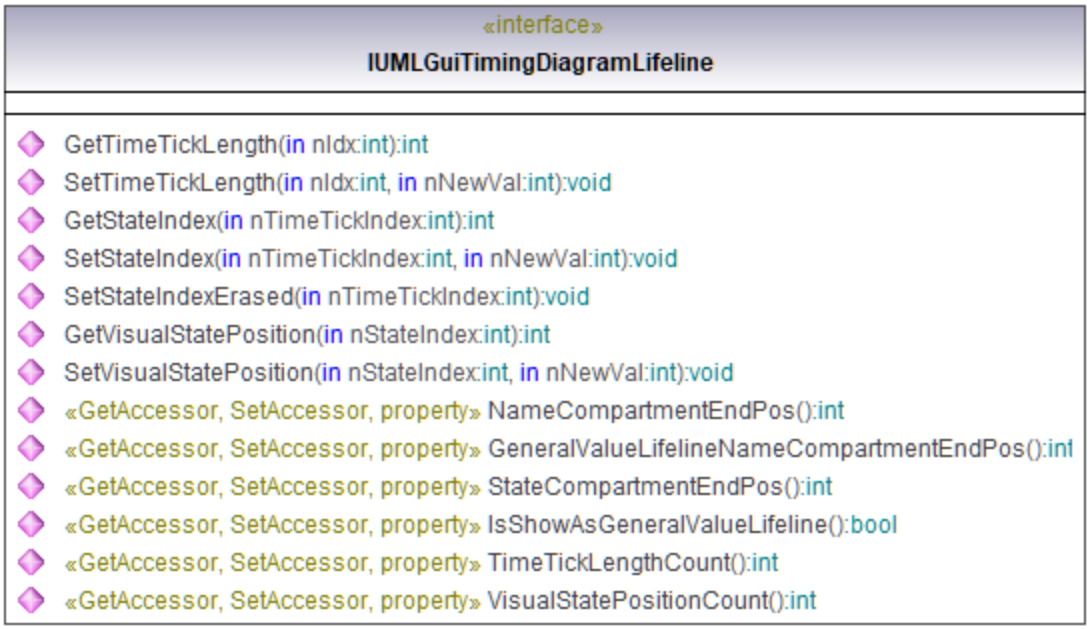
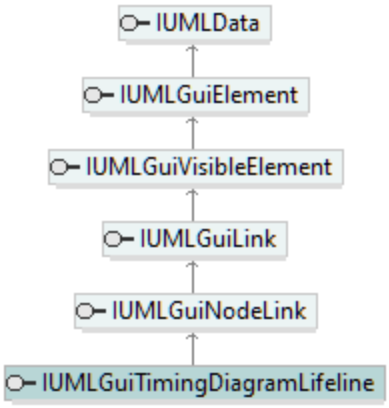
Operation **IUMLGuiTimingDiagram::AddUMLGuiTickMark**

parameter	name	direction	type	type modifier	multiplicity	default
	ipOnNode	**in**	**IUMLGuiTimingDiagramLifeline**[1256]			
	nPosX	**in**	**int**			
	nPosY	**in**	**int**			
	return	**return**	**IUMLGuiTickMark**[1254]			

UML documentation generated by UModel UML Editor http://www.altova.com/umodel Wed Sep 25 10:33:04 2024

16.4.3.6.59 UModelAPI - IUMLGuiTimingDiagramLifeline

Interface **IUMLGuiTimingDiagramLifeline**

diagram	
hierarchy	
typedElements	Interface IUMLGuiTimingDiagram 1255 Operation AddUMLGuiTickMark 1255
documentation	A IUMLGuiTimingDiagramLifeline 1256 is the graphical representation of a lifeline on a timing diagram. This type of lifeline has several options to display its data and provides access to these through its numerous properties.

Operation **IUMLGuiTimingDiagramLifeline::GeneralValueLifelineNameCompartmentEndPos**

parameter	name	direction	type	type modifier	multiplicity	default

 © 2019-2025 Altova GmbH

	return	**return**	**int**

Operation **IUMLGuiTimingDiagramLifeline::GetStateIndex**

parameter	name	direction	type	type modifier	multiplicity	default
	nTimeTickIndex	**in**	**int**			
	return	**return**	**int**			

Operation **IUMLGuiTimingDiagramLifeline::GetTimeTickLength**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	return	**return**	**int**			

Operation **IUMLGuiTimingDiagramLifeline::GetVisualStatePosition**

parameter	name	direction	type	type modifier	multiplicity	default
	nStateIndex	**in**	**int**			
	return	**return**	**int**			

Operation **IUMLGuiTimingDiagramLifeline::IsShowAsGeneralValueLifeline**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**bool**			

Operation **IUMLGuiTimingDiagramLifeline::NameCompartmentEndPos**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**int**			

Operation **IUMLGuiTimingDiagramLifeline::SetStateIndex**

parameter	name	direction	type	type modifier	multiplicity	default
	nTimeTickIndex	**in**	**int**			
	nNewVal	**in**	**int**			
	return	**return**	**void**			

Operation **IUMLGuiTimingDiagramLifeline::SetStateIndexErased**

parameter	name	direction	type	type modifier	multiplicity	default
	nTimeTickIndex	**in**	**int**			
	return	**return**	**void**			

Operation **IUMLGuiTimingDiagramLifeline::SetTimeTickLength**

parameter	name	direction	type	type modifier	multiplicity	default
	nIdx	**in**	**int**			
	nNewVal	**in**	**int**			
	return	**return**	**void**			

Operation **IUMLGuiTimingDiagramLifeline::SetVisualStatePosition**

parameter	name	direction	type	type modifier	multiplicity	default
	nStateIndex	**in**	**int**			
	nNewVal	**in**	**int**			
	return	**return**	**void**			

Operation **IUMLGuiTimingDiagramLifeline::StateCompartmentEndPos**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**int**			

Operation **IUMLGuiTimingDiagramLifeline::TimeTickLengthCount**

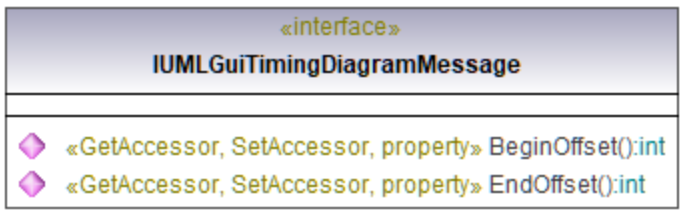
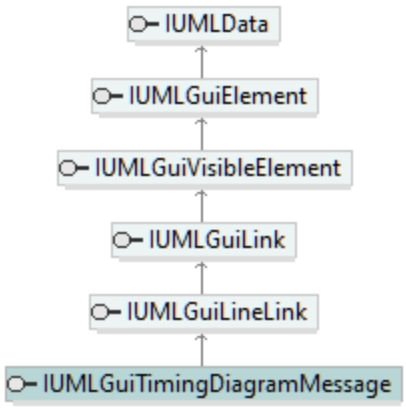
parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**int**			

Operation **IUMLGuiTimingDiagramLifeline::VisualStatePositionCount**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**int**			

16.4.3.6.60 UModelAPI - IUMLGuiTimingDiagramMessage

Interface **IUMLGuiTimingDiagramMessage**

diagram	
hierarchy	
documentation	A IUMLGuiTimingDiagramMessage 1258 is a line usually connecting two IUMLGuiTimingDiagramLifeline 1256 s. For each lifeline on one of its ends, it stores its position from the start of the state or general value compartement.

Operation **IUMLGuiTimingDiagramMessage::BeginOffset**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**int**			

 © 2019-2025 Altova GmbH

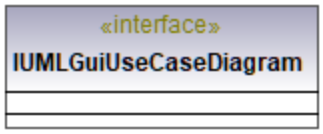
Operation **IUMLGuiTimingDiagramMessage::EndOffset**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**int**			

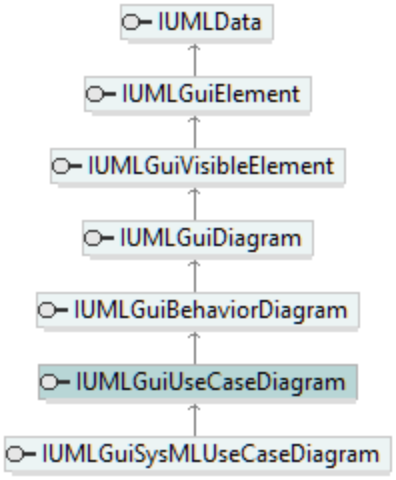
16.4.3.6.61 UModelAPI - IUMLGuiUseCaseDiagram

Interface **IUMLGuiUseCaseDiagram**

diagram



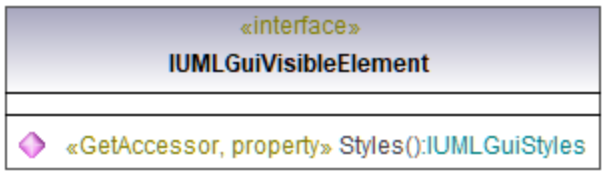
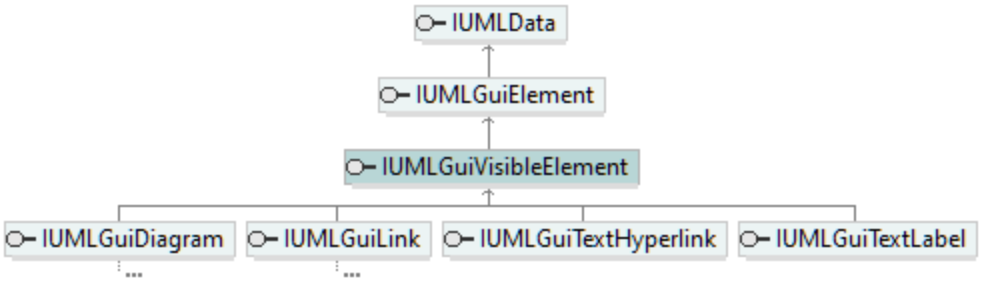
hierarchy



© 2019-2025 Altova GmbH

16.4.3.6.62 UModelAPI - IUMLGuiVisibleElement

Interface **IUMLGuiVisibleElement**

diagram	
hierarchy	
typedElements	Interface **IUMLCommentTextHyperlink** 1026 Interface **IUMLDataAll** 911 Interface **IUMLGuiTextHyperlink** 1250 Interface **IUMLHyperlink2GuiElement** 1077 Interface **IUMLNamedElement** 1117 Operation **SetHyperlinkGuiElementAddress** 1027 Operation **InsertOwnedHyperlink2GuiElementAt** 940 **LinkedGuiElement** 951 **SetHyperlinkGuiElementAddress** 966 Operation **SetHyperlinkGuiElementAddress** 1251 Operation **LinkedGuiElement** 1078 Operation **InsertOwnedHyperlink2GuiElementAt** 1118
documentation	The IUMLGuiVisibleElement 1260 is the base interface for most visible elements which can be placed on UModel diagrams. Visible elements have a style with which it is possible to influence its color and/or shape, depending on the actual type of the visible element.

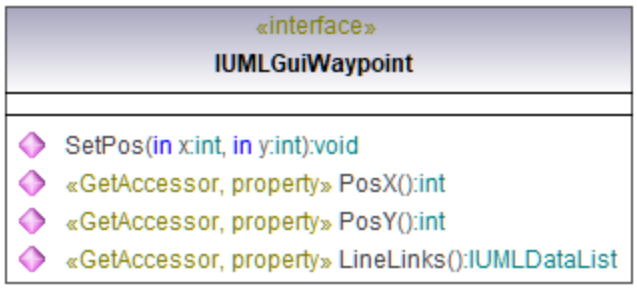
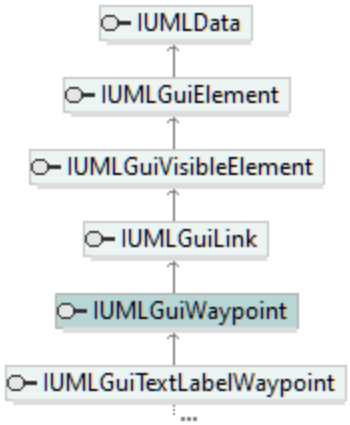
Operation **IUMLGuiVisibleElement::Styles**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLGuiStyles** 1241			

© 2019-2025 Altova GmbH

16.4.3.6.63 UModelAPI - IUMLGuiWaypoint

Interface **IUMLGuiWaypoint**

diagram	
hierarchy	
typedElements	Interface **IUMLDataAll** 911 Interface **IUMLGuiLineLink** 1222 Operation **InsertWaypointAt** 944 Operation **InsertWaypointAt** 1223
documentation	A IUMLGuiWaypoint 1261 is a part of a IUMLGuiLineLink 1222 which defines the position of one point of a line. There are several subtypes of this interface which for example make it possible to attach text labels or lines to a waypoint or line.

Operation **IUMLGuiWaypoint::LineLinks**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**IUMLDataList** 906			

Operation **IUMLGuiWaypoint::PosX**

parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**int**			

Operation **IUMLGuiWaypoint::PosY**

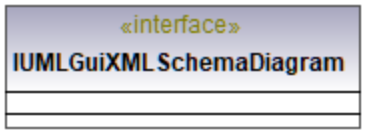
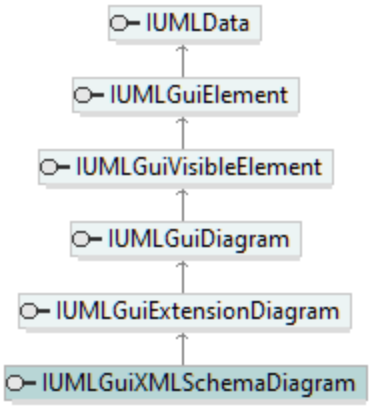
parameter	name	direction	type	type modifier	multiplicity	default
	return	**return**	**int**			

Operation **IUMLGuiWaypoint::SetPos**

parameter	name	direction	type	type modifier	multiplicity	default
	x	**in**	**int**			
	y	**in**	**int**			
	return	**return**	**void**			

16.4.3.6.64 UModelAPI - IUMLGuiXMLSchemaDiagram

Interface **IUMLGuiXMLSchemaDiagram**

diagram	
hierarchy	

16.4.3.7 Events

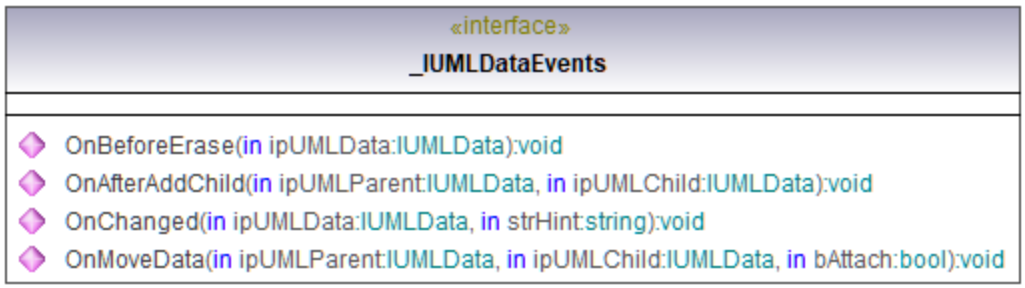
This is a list of all events sent by the UModel API on `UMLData` level.

See also How to Use UMLData Events and Event Filters 766.

 © 2019-2025 Altova GmbH

16.4.3.7.1 UModelAPI - _IUMLDataEvents

Interface **_IUMLDataEvents**

diagram	

Operation **_IUMLDataEvents::OnAfterAddChild**

parameter	name	direction	type	type modifier	multiplicity	default
	ipUMLParent	**in**	**IUMLData** 904			
	ipUMLChild	**in**	**IUMLData** 904			
	return	**return**	**void**			

Operation **_IUMLDataEvents::OnBeforeErase**

parameter	name	direction	type	type modifier	multiplicity	default
	ipUMLData	**in**	**IUMLData** 904			
	return	**return**	**void**			

Operation **_IUMLDataEvents::OnChanged**

parameter	name	direction	type	type modifier	multiplicity	default
	ipUMLData	**in**	**IUMLData** 904			
	strHint	**in**	**string**			
	return	**return**	**void**			
documentation	**strHint** is for future use only!					

Operation **_IUMLDataEvents::OnMoveData**

parameter	name	direction	type	type modifier	multiplicity	default
	ipUMLParent	**in**	**IUMLData** 904			
	ipUMLChild	**in**	**IUMLData** 904			
	bAttach	**in**	**bool**			
	return	**return**	**void**			

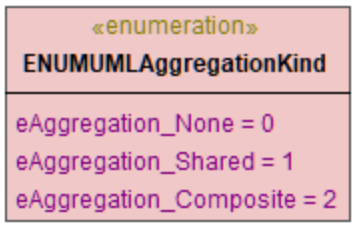
UML documentation generated by **UModel** UML Editor **http://www.altova.com/umodel** Wed Sep 25 10:33:04 2024

16.4.3.8 Enumerations

This is a list of all enumerations used by the UModel API on `UMLData` level. If your scripting environment does not support enumerations use the number-values instead.

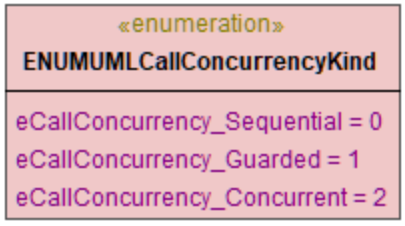
16.4.3.8.1 UModelAPI - ENUMUMLAggregationKind

Enumeration **ENUMUMLAggregationKind**

diagram	
typedElements	Interface IUMLDataAll 911 Interface IUMLProperty 1147 Operation Aggregation 914 Operation Aggregation 1148

16.4.3.8.2 UModelAPI - ENUMUMLCallConcurrencyKind

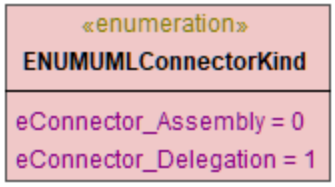
Enumeration **ENUMUMLCallConcurrencyKind**

diagram	
typedElements	Interface IUMLBehavioralFeature 1004 Interface IUMLDataAll 911 Operation Concurrency 1005 Operation Concurrency 918

 © 2019-2025 Altova GmbH

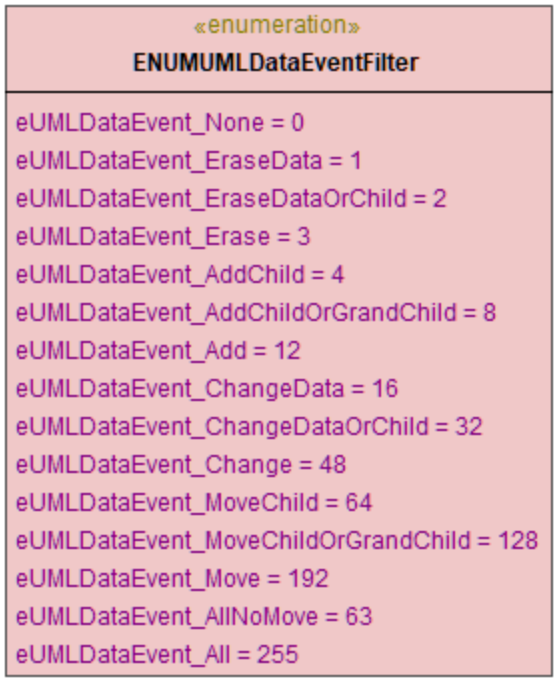
16.4.3.8.3 UModelAPI - ENUMUMLConnectorKind

Enumeration **ENUMUMLConnectorKind**

diagram	
typedElements	Interface **IUMLConnector** 1033 Interface **IUMLDataAll** 911 Operation **ConnectorKind** 1033 Operation **ConnectorKind** 918

16.4.3.8.4 UModelAPI - ENUMUMLDataEventFilter

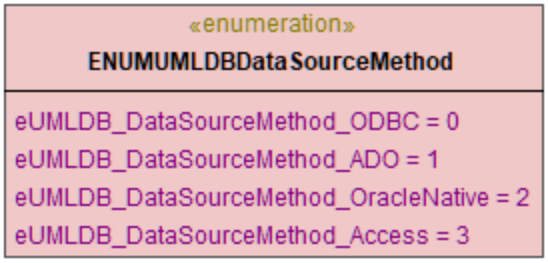
Enumeration **ENUMUMLDataEventFilter**

diagram	

© 2019-2025 Altova GmbH

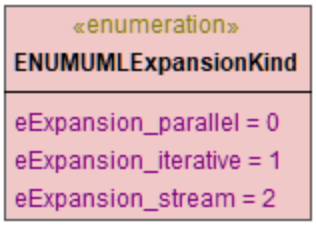
16.4.3.8.5 UModelAPI - ENUMUMLDBDataSourceMethod

Enumeration **ENUMUMLDBDataSourceMethod**

diagram	
typedElements	Interface **IImportDatabaseDlg** 855 Operation **SelectNewDataSourceByConnectionString** 855

16.4.3.8.6 UModelAPI - ENUMUMLExpansionKind

Enumeration **ENUMUMLExpansionKind**

diagram	
typedElements	Interface **IUMLDataAll** 911 Operation **Mode** 954 Interface **IUMLExpansionRegion** 1065 Operation **Mode** 1066

16.4.3.8.7 UModelAPI - ENUMUMLGuiStyleKind

Enumeration **ENUMUMLGuiStyleKind**

diagram	*The diagram is not included because of page size constraints; however, it is available in the HTML version of the manual (https://www.altova.com/manual/en/umodelprofessional/2025.2/).*
typedElements	Interface **IUMLGuiStyle** 1240 Operation **Kind** 1240

 © 2019-2025 Altova GmbH

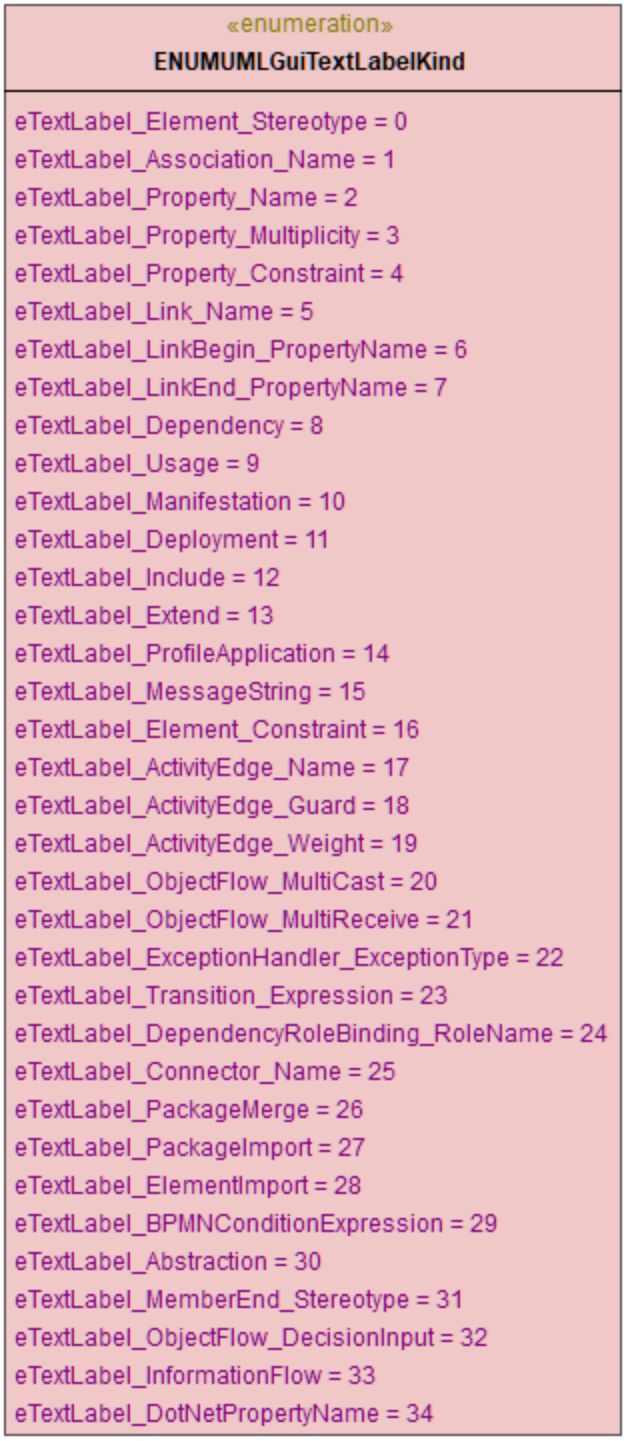
	Interface **IUMLGuiStyles** 1241	Operation **GetName** 1241 **GetStyle** 1241 **GetUsedValue** 1242 **GetValue** 1242 **SetValue** 1242

© 2019-2025 Altova GmbH

16.4.3.8.8 UModelAPI - ENUMUMLGuiTextLabelKind

Enumeration **ENUMUMLGuiTextLabelKind**

diagram



typedElements

Interface **IUMLDataAll** 911

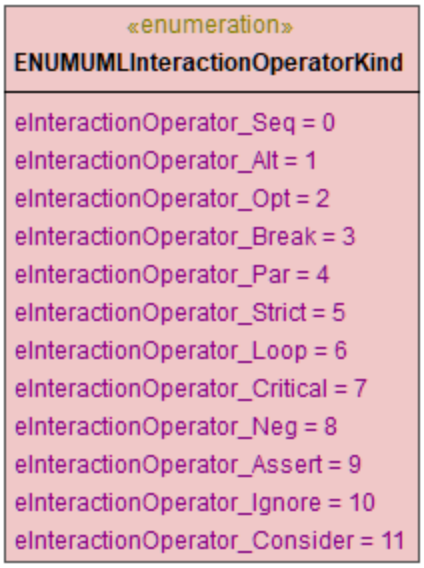
Operation **TextLabelKind** 976

© 2019-2025 Altova GmbH

	Interface **IUMLGuiTextLabel** 1251	Operation **TextLabelKind** 1252

16.4.3.8.9 UModelAPI - ENUMUMLInteractionOperatorKind

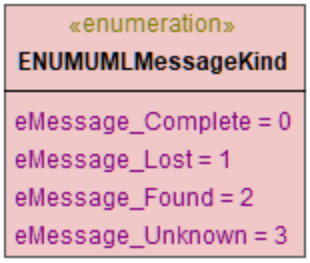
Enumeration **ENUMUMLInteractionOperatorKind**

diagram	
typedElements	Interface **IUMLCombinedFragment** 1024 — Operation **InteractionOperator** 1024 Interface **IUMLDataAll** 911 — Operation **InteractionOperator** 944

© 2019-2025 Altova GmbH

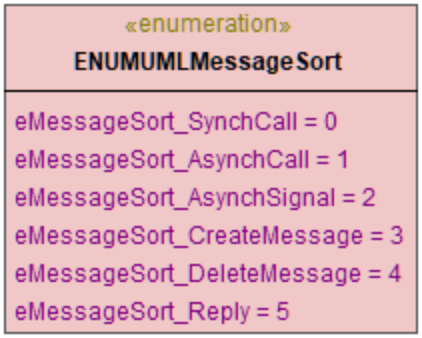
16.4.3.8.10 UModelAPI - ENUMUMLMessageKind

Enumeration **ENUMUMLMessageKind**

diagram	
typedElements	Interface **IUMLDataAll** 911 Interface **IUMLMessage** 1111 Operation **MessageKind** 953 Operation **MessageKind** 1112

16.4.3.8.11 UModelAPI - ENUMUMLMessageSort

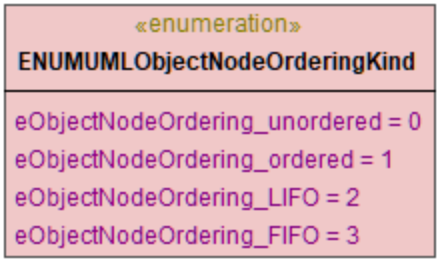
Enumeration **ENUMUMLMessageSort**

diagram	
typedElements	Interface **IUMLDataAll** 911 Interface **IUMLMessage** 1111 Operation **MessageSort** 953 Operation **MessageSort** 1112

 © 2019-2025 Altova GmbH

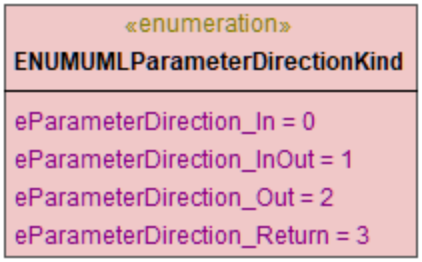
16.4.3.8.12 UModelAPI - ENUMUMLObjectNodeOrderingKind

Enumeration **ENUMUMLObjectNodeOrderingKind**

diagram	
typedElements	Interface **IUMLDataAll** 911 Interface **IUMLObjectNode** 1124 Operation **Ordering** 956 Operation **Ordering** 1125

16.4.3.8.13 UModelAPI - ENUMUMLParameterDirectionKind

Enumeration **ENUMUMLParameterDirectionKind**

diagram	
typedElements	Interface **IUMLDataAll** 911 Interface **IUMLParameter** 1140 Operation **Direction** 921 Operation **Direction** 1141

16.4.3.8.14 UModelAPI - ENUMUMLPredefinedElement

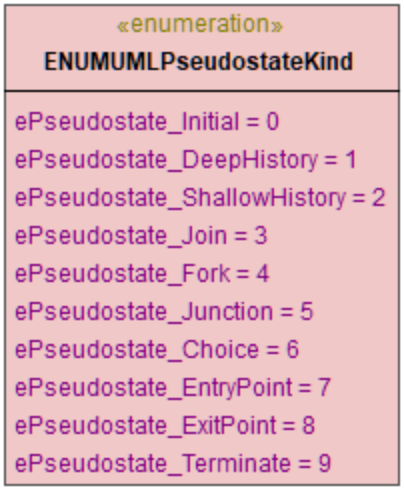
Enumeration **ENUMUMLPredefinedElement**

diagram	*The diagram is not included because of page size constraints; however, it is available in the HTML version of the manual (https://www.altova.com/manual/en/umodelprofessional/2025.2/).*

typedElements	Interface IUMLDataAll 911	Operation ApplyPredefinedStereotype 915 FindPredefinedOwnedElement 926 GetStereotypeApplicationForPredefinedStereotype 928 IsPredefinedStereotypeApplied 948 SetPredefinedTaggedValueAt 970 UnapplyPredefinedStereotype 977
	Interface IUMLElement 1050	Operation ApplyPredefinedStereotype 1051 FindPredefinedOwnedElement 1052 GetStereotypeApplicationForPredefinedStereotype 1052 IsPredefinedStereotypeApplied 1052 UnapplyPredefinedStereotype 1053
	Interface IUMLStereotypeApplication 1170	Operation SetPredefinedTaggedValueAt 1171
documentation	**Deprecated:** ePredefined_Java_finalStereotypeOfProperty ePredefined_Java_finalStereotypeOfOperation ePredefined_Java_finalStereotypeOfClass	

16.4.3.8.15 UModelAPI - ENUMUMLPseudostateKind

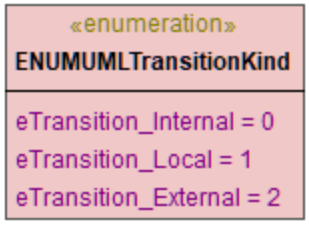
Enumeration **ENUMUMLPseudostateKind**

diagram	
typedElements	Interface IUMLDataAll 911 Interface IUMLPseudostate 1153 | Operation PseudostateKind 963 Operation PseudostateKind 1153

 © 2019-2025 Altova GmbH

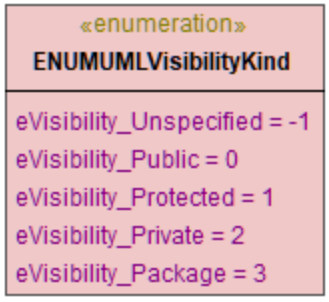
16.4.3.8.16 UModelAPI - ENUMUMLTransitionKind

Enumeration **ENUMUMLTransitionKind**

diagram	
typedElements	Interface IUMLDataAll 911 Interface IUMLTransition 1187 Operation TransitionKind 977 Operation TransitionKind 1188

16.4.3.8.17 UModelAPI - ENUMUMLVisibilityKind

Enumeration **ENUMUMLVisibilityKind**

diagram	
typedElements	Interface ILocalOptionsEditing 873 Interface IUMLDataAll 911 Interface IUMLElementImport 1054 Interface IUMLNamedElement 1117 Interface IUMLPackageImport 1138 Operation OperationsDefaultVisibility 873 PropertiesDefaultVisibility 874 Operation Visibility 979 Operation Visibility 1054 Operation Visibility 1119 Operation Visibility 1138

© 2019-2025 Altova GmbH

17 SPL Reference

This section gives you an overview of SPL (Spy Programming Language), code generator's template language. It is assumed that you have prior programming experience, and are familiar with operators, functions, variables and classes, as well as the basics of object-oriented programming - which is used heavily in SPL.

The templates used by UModel are supplied in the applications's `UModelSPL` folder. You can use these files as a guide to developing your own templates.

How code generator works

Code is generated on the basis of the template files (`.spl` files) and the object model provided by UModel. The template files contain the code of the target programming language combined with SPL instructions for creating files, reading information from the object model, and performing calculations.

The template file is interpreted by the code generator and outputs the source-code files of the target language/s (that is, the non-compiled code files) and any other relevant project file or template-dependent file.

© 2019-2025 Altova GmbH

17.1 Basic SPL structure

An SPL file contains literal text to output, interspersed with code generator instructions.

Code generator instructions are enclosed in square brackets '**[**' and '**]**'. Multiple statements can be included in a bracket pair. Additional statements have to be separated by a new line or a colon '**:**'.

Valid examples are:

```
[$x = 42
$x = $x + 1]
```

or

```
[$x = 42: $x = $x + 1]
```

Adding text to files

Text not enclosed by **[** and **]**, is written directly to the current output file.

To output literal square brackets, escape them with a backslash: **\[** and **\]**; to output a backslash use ****.

Comments

Comments inside an instruction block always begin with a **'** character, and terminate on the next line, or at a block close character **]**.

17.2 Variables

Any non-trivial SPL file will require variables. Some variables are predefined by the code generator, and new variables may be created simply by assigning values to them.

The **$** character is used when **declaring** or **using** a variable, a variable name is always prefixed by **$**. Variable names are **case sensitive**.

Variables types:

- integer - also used as boolean, where 0 is false and everything else is true
- string
- object - provided by UModel
- iterator - see foreach[1287] statement

Variable types are declared by first assignment:

```
[$x = 0]
```

x is now an integer.

```
[$x = "teststring"]
```

x is now treated as a string.

Strings

String constants are always enclosed in double quotes, like in the example above. **\n** and **\t** inside double quotes are interpreted as newline and tab, **\"** is a literal double quote, and \\ is a backslash. String constants can also span multiple lines.

String concatenation uses the **&** character:

```
[$BasePath = $outputpath & "/" & $JavaPackageDir]
```

Objects

Objects represent the information contained in the UModel project. Objects have **properties**, which can be accessed using the **.** operator. It is not possible to create new objects in SPL (they are predefined by the code generator, derived from the input), but it is possible to assign objects to variables.

Example:

```
class [=$class.Name]
```

This example outputs the word "class", followed by a space and the value of the **Name** property of the **$class** object.

© 2019-2025 Altova GmbH

The following table shows the relationship between UML elements their SPL equivalents along with a short description.

Predefined variables

UML element	SPL property	Multiplicity	UML Attribute / Association	UModel Attribute / Association	Description
BehavioralFeature	isAbstract		isAbstract:Boolean		
BehavioralFeature	raisedException	*	raisedException:Type		
BehavioralFeature	ownedParameter	*	ownedParameter:Parameter		
BehavioredClassifier	interfaceRealization	*	interfaceRealization:InterfaceRealization		
Class	ownedOperation	*	ownedOperation:Operation		
Class	nestedClassifier	*	nestedClassifier:Classifier		
Classifier	namespace	*		namespace:Package	packages with code language <<namespace>> set
Classifier	rootNamespace	*		project root namespace:String	VB only - root namespace
Classifier	generalization	*	generalization:Generalization		
Classifier	isAbstract		isAbstract:Boolean		
ClassifierTemplateParameter	constrainingClassifier	*	constrainingClassifier		
Comment	body		body:String		
DataType	ownedAttribute	*	ownedAttribute:Property		
DataType	ownedOperation	*	ownedOperation:Operation		
Element	kind			kind:String	
Element	owner	0..1	owner:Element		
Element	appliedStereotype	*		appliedStereotype:StereotypeApplication	applied stereotypes

UML element	SPL property	Multiplicity	UML Attribute / Association	UModel Attribute / Association	Description
Element	ownedComment	*	ownedComment:Comment		
ElementImport	importedElement	1	importedElement:PackageableElement		
Enumeration	ownedLiteral	*	ownedLiteral:EnumerationLiteral		
Enumeration	nestedClassifier	*		nestedClassifier::Classifier	
Enumeration	interfaceRealization	*		interfaceRealization:Interface	
EnumerationLiteral	ownedAttribute	*		ownedAttribute:Property	
EnumerationLiteral	ownedOperation	*		ownedOperation:Operation	
EnumerationLiteral	nestedClassifier	*		nestedClassifier:Classifier	
Feature	isStatic		isStatic:Boolean		
Generalization	general	1	general:Classifier		
Interface	ownedAttribute	*	ownedAttribute:Property		
Interface	ownedOperation	*	ownedOperation:Operation		
Interface	nestedClassifier	*	nestedClassifier:Classifier		
InterfaceRealization	contract	1	contract:Interface		
MultiplicityElement	lowerValue	0..1	lowerValue:ValueSpecification		
MultiplicityElement	upperValue	0..1	upperValue:ValueSpecification		
NamedElement	name		name:String		
NamedElement	visibility		visibility:VisibilityKind		
NamedElement	isPublic			isPublic:Boolean	visibility <public>
NamedElement	isProtected			isProtected:Boolean	visibility <protected>
NamedElement	isPrivate			isPrivate:Boolean	visibility <private>

© 2019-2025 Altova GmbH

UML element	SPL property	Multiplicity	UML Attribute / Association	UModel Attribute / Association	Description
NamedElement	isPackage			isPackage:Boolean	visibility <package>
NamedElement	namespacePrefix			namespacePrefix:String	XSD only - namespace prefix when exists
NamedElement	parseableName			parseableName:String	CSharp, VB only - name with escaped keywords (@)
Namespace	elementImport	*	elementImport:ElementImport		
Operation	ownedReturnParameter	0..1		ownedReturnParameter:Parameter	parameter with direction return set
Operation	type	0..1		type	type of parameter with direction return set
Operation	ownedOperationParameter	*		ownedOperationParameter:Parameter	all parameters excluding parameter with direction return set
Operation	implementedInterface	1		implementedInterface:Interface	CSharp only - the implemented interface
Operation	ownedOperationImplementations	*		implementedOperation:OperationImplementation	VB only - the implemented interfaces/operations
OperationImplementation	implementedOperationOwner	1		implementedOperationOwner:Interface	interface implemented by the operation
OperationImplementation	implementedOperationName			name:String	name of the implemented operation
OperationImplementation	implementedOperationParseableName			parseableName:String	name of the implemented operation with escaped keywords
Package	namespace	*		namespace:Package	packages with code language <<namespace>> set
PackageableElement	owningPackage	0..1		owningPackage	set if owner is a package

UML element	SPL property	Multiplicity	UML Attribute / Association	UModel Attribute / Association	Description
PackageableElement	owningNamespace Package	0..1		owningNamespace Package:Package	owning package with code language <<namespace>> set
Parameter	direction		direction:Parameter DirectionKind		
Parameter	isIn			isIn:Boolean	direction <in>
Parameter	isInOut			isInOut:Boolean	direction <inout>
Parameter	isOut			isOut:Boolean	direction <out>
Parameter	isReturn			isReturn:Boolean	direction <return>
Parameter	isVarArgList			isVarArgList:Boolean	true if parameter is a variable argument list
Parameter	defaultValue	0..1	defaultValue:Value Specification		
Property	defaultValue	0..1	defaultValue:Value Specification		
RedefinableElement	isLeaf		isLeaf:Boolean		
Slot	name			name:String	name of the defining feature
Slot	values	*	value:ValueSpecification		
Slot	value			value:String	value of the first value specification
StereotypeApplication	name			name:String	name of applied stereotype
StereotypeApplication	taggedValue	*		taggedValue:Slot	first slot of the instance specification
StructuralFeature	isReadOnly		isReadOnly		
StructuredClassifier	ownedAttribute	*	ownedAttribute:Property		
TemplateBinding	signature	1	signature:Template Signature		
TemplateBinding	parameterSubstitution	*	parameterSubstitution:TemplateParameterSubstitution		

© 2019-2025 Altova GmbH

UML element	SPL property	Multiplicity	UML Attribute / Association	UModel Attribute / Association	Description
TemplateParameter	paramDefault			paramDefault:String	template parameter default value
TemplateParameter	ownedParameteredElement	1	ownedParameteredElement:ParameterableElement		
TemplateParameterSubstitution	parameterSubstitution			parameterSubstitution:String	Java only - code wildcard handling
TemplateParameterSubstitution	parameterDimensionCount			parameterDimensionCount:Integer	code dimension count of the actual parameter
TemplateParameterSubstitution	actual	1	OwnedActual:ParameterableElement		
TemplateParameterSubstitution	formal	1	formal:TemplateParameter		
TemplateSignature	template	1	template:TemplateableElement		
TemplateSignature	ownedParameter	*	ownedParameter:TemplateParameter		
TemplateableElement	isTemplate			isTemplate:Boolean	true if template signature set
TemplateableElement	ownedTemplateSignature	0..1	ownedTemplateSignature:TemplateSignature		
TemplateableElement	templateBinding	*	templateBinding:TemplateBinding		
Type	typeName	*		typeName:PackageableElement	qualified code type names
TypedElement	type	0..1	type:Type		
TypedElement	postTypeModifier			postTypeModifier:String	postfix code modifiers
ValueSpecification	value			value:String	string value of the value specification

Adding a prefix to attributes of a class during code generation

You might need to prefix all new attributes with the "m_" characters in your project.

All new coding elements are written using the SPL templates. For example, if you open **UModelSPL\C#[Java]\Default\Attribute.spl**, you can change the way the name is written. Namely, you can replace

```
write $Property.name
```

with

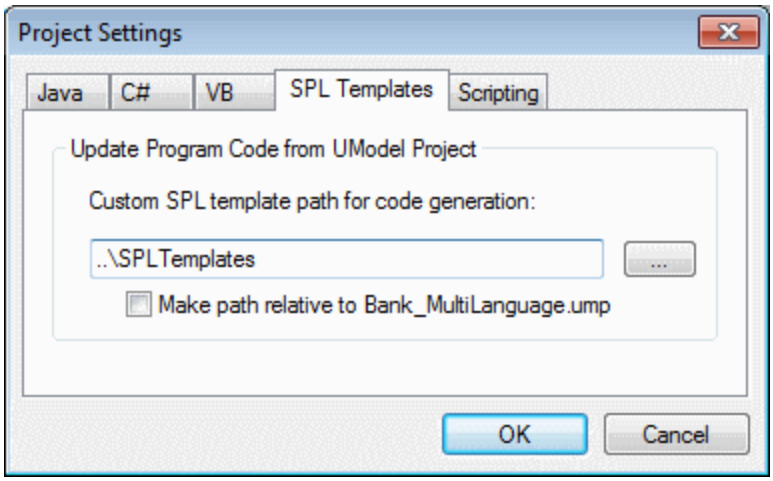
```
write "m_" & $Property.name
```

It is highly recommended that you immediately update your model from code after code generation, to ensure that code and model are synchronized.

Note: As previously mentioned, copy the SPL templates one directory higher (i.e. above the **default** directory to **UModelSPL\C#**) before modifying them. This ensures that they are not overwritten when you install a new version of UModel. Please make sure that the "user-defined override default" check box is activated in the **Code from Model** tab of the "Synchronization Settings" dialog box.

SPL Templates

SPL templates can be specified per UModel project using the menu option **Project | Project Settings** (as shown in the screenshot below). Relative paths are also supported. Templates which are not found in the specified directory, are searched for in the local default directory.



Global objects

$Options	an object holding global options:
	generateComments:bool generate doc comments (true/false)
$Indent	a string used to indent generated code and represent the current nesting level
$IndentStep	a string, used to indent generated code and represent one nesting level
$NamespacePrefix	XSD only – the target namespace prefix if present

© 2019-2025 Altova GmbH

String manipulation routines

```
integer Compare(s)
```

The return value indicates the lexicographic relation of the string to s (case sensitive):

<0:	the string is less than s
0:	the string is identical to s
>0:	the string is greater than s

```
integer CompareNoCase(s)
```

The return value indicates the lexicographic relation of the string to s (case insensitive):

<0:	the string is less than s
0:	the string is identical to s
>0:	the string is greater than s

```
integer Find(s)
```

Searches the string for the first match of a substring s. Returns the zero-based index of the first character of s or -1 if s is not found.

```
string Left(n)
```

Returns the first n characters of the string.

```
integer Length()
```

Returns the length of the string.

```
string MakeUpper()
```

Returns a string converted to upper case.

```
string MakeUpper(n)
```

Returns a string, with the first n characters converted to upper case.

```
string MakeLower()
```

Returns a string converted to lower case.

```
string MakeLower(n)
```

Returns a string, with the first n characters converted to lower case.

```
string Mid(n)
```

Returns a string starting with the zero-based index position n

```
string Mid(n,m)
```

Returns a string starting with the zero-based index position n and the length m

```
string RemoveLeft(s)
```

Returns a string excluding the substring s if Left(s.Length()) is equal to substring s.

```
string RemoveLeftNoCase(s)
```

Returns a string excluding the substring s if Left(s.Length()) is equal to substring s (case insensitive).

```
string RemoveRight(s)
```

Returns a string excluding the substring s if Right(s.Length()) is equal to substring s.

```
string RemoveRightNoCase(s)
```

Returns a string excluding the substring s if Right(s.Length()) is equal to substring s (case insensitive).

```
string Repeat(s,n)
```

Returns a string containing substring s repeated n times.

```
string Right(n)
```

Returns the last n characters of the string.

© 2019-2025 Altova GmbH

17.3 Operators

Operators in SPL work like in most other programming languages.

List of SPL operators in descending precedence order:

.	Access object property
( )	Expression grouping
true	boolean constant "true"
false	boolean constant "false"
&	String concatenation
-	Sign for negative number
not	Logical negation
*	Multiply
/	Divide
%	Modulo
+	Add
-	Subtract
<=	Less than or equal
<	Less than
>=	Greater than or equal
>	Greater than
=	Equal
<>	Not equal
and	Logical conjunction (with short circuit evaluation)
or	Logical disjunction (with short circuit evaluation)
=	Assignment

17.4 Conditions

SPL allows you to use standard "if" statements. The syntax is as follows:

```
if condition
   statements
else
   statements
endif
```

or, without else:

```
if condition
   statements
endif
```

Note: There are no round brackets enclosing the condition.

As in any other programming language, conditions are constructed with logical and comparison operators[1285].

Example:

```
[if $namespace.ContainsPublicClasses and $namespace.Prefix <> ""]
   whatever you want ['inserts whatever you want, in the resulting file]
[endif]
```

Switch

SPL also contains a multiple choice statement.

Syntax:

```
switch $variable
   case X:
      statements
   case Y:
   case Z:
      statements
   default:
      statements
endswitch
```

The case labels must be constants or variables.

The switch statement in SPL does not fall through the cases (as in C), so there is no need for a "break" statement.

© 2019-2025 Altova GmbH

17.5 Collections and foreach

Collections and iterators

A collection contains multiple objects - like a ordinary array. Iterators solve the problem of storing and incrementing array indexes when accessing objects.

Syntax:

```
foreach iterator in collection
   statements
next
```

Example:

```
[foreach $class in $classes
   if not $class.IsInternal
      ]  class [=$class.Name];
[  endif
next]
```

Example 2:

```
[foreach $i in 1 To 3
      Write "// Step " & $i & "\n"
      ' Do some work
next]
```

Foreach steps through all the items in $classes, and executes the code following the instruction, up to the **next** statement, for each of them.

In each iteration, **$class** is assigned to the next class object. You simply work with the class object instead of using, classes[i]->Name(), as you would in C++.

All collection iterators have the following additional properties:

Index	The current index, starting with 0
IsFirst	true if the current object is the first of the collection (index is 0)
IsLast	true if the current object is the last of the collection

Example:

```
[foreach $enum in $facet.Enumeration
   if not $enum.IsFirst
      ], [
```

```
   endif
   ]"[=$enum.Value]"[
next]
```

Collection manipulation routines:

collection **SortByName**(bAscending)

returns a collection whose elements are sorted by name (case sensitive) in ascending or descending order.

collection **SortByNameNoCase**(bAscending)

returns a collection whose elements are sorted by name (case insensitive) in ascending or descending order

Example:

```
$SortedNestedClassifier = $Class.nestedClassifier.SortByNameNoCase( true )
```

collection **SortByKind**(bAscending)

returns a collection whose elements are sorted by kind names (e.g. “Class”, “Interface”,…) in ascending or descending order.

collection **SortByKindAndName**(bAscendingKind, bAscendingName)

returns a collection whose elements are sorted by kind (e.g. “Class”, “Interface”,…) in ascending or descending order and if the kinds are equal by name (case sensitive in ascending or descending order)

collection **SortByKindAndNameNoCase**(bAscending)

returns a collection whose elements are sorted by kind (e.g. “Class”, “Interface”,…) in ascending or descending order and if the kinds are equal by name (case insensitive in ascending or descending order)

© 2019-2025 Altova GmbH

17.6 Subroutines

Code generator supports subroutines in the form of procedures or functions.

Features:

- By-value and by-reference passing of values
- Local/global parameters (local within subroutines)
- Local variables
- Recursive invocation (subroutines may call themselves)

17.6.1 Subroutine declaration

Subroutines

Syntax example:

```
Sub SimpleSub()

       ... lines of code
 EndSub
```

- **Sub** is the keyword that denotes the procedure.
- **SimpleSub** is the name assigned to the subroutine.
- Round **parenthesis** can contain a parameter list.
- The code block of a subroutine starts immediately after the closing parameter parenthesis.
- **EndSub** denotes the end of the code block.

Note: Recursive or cascaded subroutine **declaration** is not permitted, i.e. a subroutine may not contain another subroutine.

Parameters

Parameters can also be passed by procedures using the following syntax:

- All parameters must be variables
- Variables must be prefixed by the **$** character
- Local variables are defined in a subroutine
- Global variables are declared explicitly, outside of subroutines
- Multiple parameters are separated by the comma character "**,**" within round parentheses
- Parameters can pass values

Parameters - passing values

Parameters can be passed in two ways, by value and by reference, using the keywords **ByVal** and **ByRef** respectively.

Syntax:

```
' define sub CompleteSub()
[Sub CompleteSub( $param, ByVal $paramByValue, ByRef $paramByRef )
] ...
```

- **ByVal** specifies that the parameter is passed by value. Note that most objects can only be passed by reference.
- **ByRef** specifies that the parameter is passed by reference. This is the default if neither ByVal nor ByRef is specified.

Function return values

To return a value from a subroutine, use the **return** statement. Such a function can be called from within an expression.

Example:

```
' define a function
[Sub MakeQualifiedName( ByVal $namespacePrefix, ByVal $localName )
if $namespacePrefix = ""
  return $localName
else
  return $namespacePrefix & ":" & $localName
endif
EndSub
]
```

17.6.2 Subroutine invocation

Use **call** to invoke a subroutine, followed by the procedure name and parameters, if any.

```
Call SimpleSub()
```

or

```
Call CompleteSub( "FirstParameter", $ParamByValue, $ParamByRef )
```

Function invocation

To invoke a function (any subroutine that contains a **return** statement), simply use its name inside an expression. Do not use the **call** statement to call functions. Example:

```
$QName = MakeQualifiedName($namespace, "entry")
```


© 2019-2025 Altova GmbH

18 License Information

This section contains information about:

- the distribution of this software product
- software activation and license metering
- the license agreement governing the use of this product

Please read this information carefully. It is binding upon you since you agreed to these terms when you installed this software product.

To view the terms of any Altova license, go to the Altova Legal Information page at the Altova website.

18.1 Electronic Software Distribution

This product is available through electronic software distribution, a distribution method that provides the following unique benefits:

- You can evaluate the software free-of-charge for 30 days before making a purchasing decision. *(Note: Altova MobileTogether Designer is licensed free of charge.)*
- Once you decide to buy the software, you can place your order online at the Altova website and get a fully licensed product within minutes.
- When you place an online order, you always get the latest version of our software.
- The product package includes an onscreen help system that can be accessed from within the application interface. The latest version of the user manual is available at www.altova.com in (i) HTML format for online browsing, and (ii) PDF format for download (and to print if you prefer to have the documentation on paper).

30-day evaluation period

After downloading this product, you can evaluate it for a period of up to 30 days free of charge. About 20 days into the evaluation period, the software will start to remind you that it has not yet been licensed. The reminder message will be displayed once each time you start the application. If you would like to continue using the program after the 30-day evaluation period, you must purchase a product license, which is delivered in the form of a license file containing a key code. Unlock the product by uploading the license file in the Software Activation dialog of your product.

You can purchase product licenses at https://shop.altova.com/.

Helping Others within Your Organization to Evaluate the Software

If you wish to distribute the evaluation version within your company network, or if you plan to use it on a PC that is not connected to the Internet, you may distribute only the installer file, provided that this file is not modified in any way. Any person who accesses the software installer that you have provided must request their own 30-day evaluation license key code and after expiration of their evaluation period, must also purchase a license in order to be able to continue using the product.

© 2019-2025 Altova GmbH

18.2 Software Activation and License Metering

As part of Altova's Software Activation, the software may use your internal network and Internet connection for the purpose of transmitting license-related data at the time of installation, registration, use, or update to an Altova-operated license server and validating the authenticity of the license-related data in order to protect Altova against unlicensed or illegal use of the software and to improve customer service. Activation is based on the exchange of license related data such as operating system, IP address, date/time, software version, and computer name, along with other information between your computer and an Altova license server.

Your Altova product has a built-in license metering module that further helps you avoid any unintentional violation of the End User License Agreement. Your product is licensed either as a single-user or multi-user installation, and the license-metering module makes sure that no more than the licensed number of users use the application concurrently.

This license-metering technology uses your local area network (LAN) to communicate between instances of the application running on different computers.

Single license

When the application starts up, as part of the license metering process, the software sends a short broadcast datagram to find any other instance of the product running on another computer in the same network segment. If it doesn't get any response, it will open a port for listening to other instances of the application.

Multi-user license

If more than one instance of the application is used within the same LAN, these instances will briefly communicate with each other on startup. These instances exchange key-codes in order to help you to better determine that the number of concurrent licenses purchased is not accidentally violated. This is the same kind of license metering technology that is common in the Unix world and with a number of database development tools. It allows Altova customers to purchase reasonably-priced concurrent-use multi-user licenses.

We have also designed the applications so that they send few and small network packets so as to not put a burden on your network. The TCP/IP ports (2799) used by your Altova product are officially registered with the IANA (*see the IANA Service Name Registry for details*) and our license-metering module is tested and proven technology.

If you are using a firewall, you may notice communications on port 2799 between the computers that are running Altova products. You are, of course, free to block such traffic between different groups in your organization, as long as you can ensure by other means, that your license agreement is not violated.

Note about certificates

Your Altova application contacts the Altova licensing server (`link.altova.com`) via HTTPS. For this communication, Altova uses a registered SSL certificate. If this certificate is replaced (for example, by your IT department or an external agency), then your Altova application will warn you about the connection being insecure. You could use the replacement certificate to start your Altova application, but you would be doing this at your own risk. If you see a *Non-secure connection* warning message, check the origin of the certificate and consult your IT team (who would be able to decide whether the interception and replacement of the Altova certificate should continue or not).

If your organization needs to use its own certificate (for example, to monitor communication to and from client machines), then we recommend that you install Altova's free license management software, Altova LicenseServer, on your network. Under this setup, client machines can continue to use your organization's certificates, while Altova LicenseServer can be allowed to use the Altova certificate for communication with Altova.

© 2019-2025 Altova GmbH

18.3 Altova End-User License Agreement

- The Altova End-User License Agreement is available here: https://www.altova.com/legal/eula
- Altova's Privacy Policy is available here: https://www.altova.com/privacy

18.4 Packaging License Files with UModel Installer

If you want to perform a silent installation of UModel, you may want to modify the MSI database so that it includes your license file(s). This way, the installer will not only install the product but also license it. For details about how to achieve this, download this ZIP file from the Altova website and open the PDF document in it.

© 2019-2025 Altova GmbH

Index

.

.NET 5,
as UModel profile, 164
importing types from binaries, 101
support, 13
.NET Core, 13
importing assemblies, 209
.NET Framework, 164
importing assemblies, 209

3

3-way project,
merge, 276

A

Abstract,
class, 31
Activation box,
Execution Specification, 354
Activity,
Add diagram to transition, 316
Add operation, 316
Add to state, 316
BPMN, 442
create branch / merge, 301
diagram elements, 303
diagram SysML, 481
icons, 638
Activity diagram, 297
inserting elements, 298
Actor,
customize, 22
user-defined, 22
Add, 622
diagram to package, 22
new project, 153
package to project, 22
project to source control, 622
to source control, 622
ADO,
as data connection interface, 507
setting up a connection, 514
ADO.NET,
setting up a connection, 519
Align,
elements when dragging, 22
snap lines when dragging, 685
All,
expand / collapse, 387
Annotation,
text - BPMN, 454
Application object, 753
Artifact,
add to node, 59
BPMN, 454
manifest, 59
Association,
aggregate/composite, 31
as relationship, 136
between classes, 31
BPMN, 451
changing the properties of, 139
creating, 136, 139
object links, 46
reflexive associations, 139
show typed property, 283
use case, 22
viewing, 139
Association qualifier,
creating, 139
Associations,
viewing, 91
Attribute,
autocompletion window, 685
coloring, 392
show / hide, 387
Autocomplete,
function, 31
Autocompletion,
window on class editing, 685
Autocompletion of data types,
disabling, 134
triggering, 134
Autogenerate,

© 2019-2025 Altova GmbH

Autogenerate,
reply message, 360
Automatially add operation, 316
Azure SQL, 547

B

Ball and socket,
interface notation, 387
Base,
class, 40
Base class,
expand, collapse compartments, 387
multiple instances on diagram, 387
overriding, 387
Batch mode,
creating projects, 106
loading projects, 106
saving projects, 106
Behavioral,
diagrams, 297
Binary files,
import into model, 204
Binding,
template, 283
BPMN, 441
artifacts, 454
association, 451
convert 1.0 to 2.0, 441
BPMN 2.0,
events, 442
flow objects, 442
tasks, 442
Branch,
create in Activity, 301
Business Process Modeling Notation, 441
icons, 655

C

C#,
auto-implemented properties, 177
code generation options, 175
code import options, 194
error handling, 769
generate code, 177
generating code, 170
import attributes, 205
import binary files, 204, 209
importing source code, 192
C++,
error handling, 769
Call,
message, 360
Call message,
go to operation, 360
CallBehavior,
insert, 298
CallOperation,
insert, 298
Catalog,
file - XMLSpy Catalog file, 685
Change provider,
source control, 630
Check In, 620
Check Out, 618
Class,
abstract and concrete, 31
add new, 31
add operations, 31
add properties, 31
associations, 31
base, 40
derived, 40
diagrams, 31
enable autocompletion window, 685
icons, 640
in component diagram, 53
name changes - synchronization, 220
synchronization, 217
syntax coloring, 392
Class diagram, 387
Class name changing,
effect on code file name, 220
Classifier,
constraining, 281
new, 218
renaming, 218
Code, 220
adding code to sequence diagram, 375
default, 685
generate from sequence diagram, 372

© 2019-2025 Altova GmbH

Code, 220
generate sequence diagram from, 759
generate sequence diagram manually, 760
generating sequence diagrams from, 366
Java code and class file names, 220
refactoring, 220
SPL, 1274
synchronization, 217
Code engineering,
errors, 96
from code to model, 73
from model to code, 64
generate ComponentRealizations, 218
information messages, 96
move project file to new location, 153
resolving associatons, 142
tutorial samples, 18
warnings, 96
Collaboration,
Composite Structre diagram, 402
Collapse,
class compartments, 387
Collapsed,
sub process, 450
Collection Association,
creating, 142
prerequisites, 142
resolving to collection templates, 142
Color,
syntax coloring - enable/disable, 392
COM API,
in Scripting Editor, 716
Combined fragment, 356
Command,
add to toolbar/menu, 676
Command line,
creating projects, 106
Generating program code, 101
Importing binary types, 101
Importing source code, 101
loading projects, 106
Reference, 101
saving projects, 106
Synchronizing code and model, 101
Communication,
icons, 641
Communication diagram, 342
generate from Sequence diagram, 343
Compare source files, 628
Compartment,
expand single / multiple, 387
Compatibility,
updating projects, 217
Component,
diagram, 53
icons, 643
insert class, 53
realization, 53
Component diagram, 404
Component view,
as package, 112
ComponentRealizations,
autogeneration, 218
Composite state, 323
add region, 323
Composite Structure,
icons, 642
insert elements, 402
Composite Structure diagram, 401
Composition,
association - create, 31
Concrete,
class, 31
Conditional flow, 451
Connecting objects, 451
Constraining,
classifiers, 281
Containment,
drawing in a diagram, 145
Continuous flow,
modeling / streaming SysML Activity, 481
Convert,
BPMN 1.0 to 2.0, 441
Copyright information, 1291
CR/LF,
for ump file on save, 153
Create,
getter / setter methods, 387
Customize,
actor, 22
toolbar/menu commands, 676

© 2019-2025 Altova GmbH

D

Data object,
BPMN, 454
Database,
configuring for round-trip engineering, 500
importing into UModel, 486, 488
modeling with UModel, 487
updating from the model, 501
Database connection,
setting up, 507
setup examples, 533
starting the wizard, 508
Database drivers,
overview, 510
Default,
project code, 685
SPL templates, 217
Default flow, 451
Delete,
command from toolbar, 676
icon from toolbar, 676
toolbar, 677
Dependencies,
viewing, 91
Dependency,
include, 22
usage, 53
Deployment,
diagram, 59
icons, 644
Deployment diagram, 404
Derived,
class, 40
Diagram, 405
- Activity, 297
- BMPN, 441
- Communication, 342
- Component, 404
- Composite structure, 401
- Deployment, 404
- Interaction Overview, 346
- Object, 405
- Package, 405
- Sequence, 351
- State machine, 314
- Timing, 378
- Use Case, 342
Add activity to transition, 316
add to Favorites, 88
adding code to sequence diagram, 375
Additional - XML schema, 424
Class, 387
database - importing, 486
finding unused elements, 116
generate code from sequence diagram, 372
generate Package dependency diagram, 405
icon reference, 83
icons, 637
ignore elem. from inluded files, 685
inserting elements into, 110
multiple instances of class, 387
quick scroll, 93
save as png, 660
save open diagrams with project, 685
Sequence diagrams SysML, 482
State Machine SysML, 483
styles, 90
Use case SysML, 484
viewing an outline of, 93
XML Schema, 424
Diagram - sequence,
generate from code, 759
generate manually from code, 760
generate sequence diagram from code, 760
Diagram Tree window, 87
Diagram type,
identifying, 98
Diagrams, 296
adding layers to, 132
behavioral, 297
changing the appearance of, 128
changing the size of, 128
creating, 98, 124
deleting from project, 128
fit into window, 135
generating, 125
generating from Hierarchy window, 91
opening, 127
structural, 387
viewing inside a project, 87
zoom in/out, 135
Directory,

© 2019-2025 Altova GmbH

Directory,
change project location, 153
ignoring on merge, 685
Disable source control, 615
Distribution,
of Altova's software products, 1291, 1292
Documentation,
adding to elements, 121
generate from UML project, 285
generating source code with, 121
importing from source code, 121
Documentation window, 94
Download source control project, 612
Drid,
snap lines while dragging, 22
DurationConstraint,
Timing diagram, 384

E

Edit menu,
commands, 662
Element,
add to Favorites, 88
styles, 90
ElementImport,
viewing, 91
Elements,
adding to a diagram, 110
adding to the model, 83, 109
aligning within a diagram, 130
applying custom images to, 122
autolayout, 130
changing properties of, 89
changing the appearance of, 122
constraining, 117
copying, 112
deleting from diagram, 113
deleting from project, 113
documenting, 94, 121
finding, 114
finding in a diagram, 116
hyperlinking, 118
ignore from include files, 685
insert State Machine, 315
moving, 112
moving between layers, 132
renaming, 112
replacing, 114
resizing, 130
Enable source control, 615
End User License Agreement, 1291, 1295
Enhance,
performance, 169
Entry point,
add to submachine, 323
Error handling,
general description, 769
Errors,
during code engineering, 96
Evaluation period,
of Altova's software products, 1291, 1292
Event,
BPMN, 442
Event/Stimulus,
Timing diagram, 383
Exception,
Adding raised exception, 387
Execution specification,
lifeline, 354
Exit point,
add to submachine, 323
Expand,
all class compartments, 387
Expanded,
sub process, 448
Export,
UModel projects to XMI, 568
External applications,
opening from UModel, 678

F

Favorites window,
adding to, 88
removing from, 88
Fetch file,
source control, 616
File,
merging project files, 276
open from URL, 660
ump, 153

© 2019-2025 Altova GmbH

File DSN,
setting up, 528
File menu,
commands, 660
Find,
diagrams, 114
elements, 114
text, 114
Firebird,
Connecting through JDBC, 533
Connecting through ODBC, 534
Flow, 451
conditional, 451
default, 451
message, 451
sequence, 451
Flow objects, 442
Folders,
get in source control, 617
Forward engineering, 64

G

Gate,
sequence diagram, 359
Gateways,
BPMN 2.0, 442
Complex Gateway (Decision/Merge), 442
Data Based Exclusive Gateway (XOR), 442
Event Based Exclusive Gateway (XOR), 442
Inclusive Gateway (OR), 442
Parallel Gateway (AND), 442
General Value lifeline,
Timing diagram, 379
Generalization,
as relationship, 110, 136
creating, 136
Generalizations,
viewing, 91
Generalize,
specialize, 40
Generate,
ComponentRealizations automatically, 218
reply message automatically, 360
Sequence dia from Communication, 343
sequence diagram from code, 759
UML project documentation, 285
Generate manually,
sequence diagram from code, 760
Generated documentation,
options, 289
Get,
getter / setter methods, 387
Get file,
source control, 616
Get folders,
source control, 617
Get latest version, 616
Goto,
lifeline, 354
Grid,
snap lines, 685
Group,
BPMN, 454

H

Help menu,
commands, 700
Hide,
show - slot, 387
Hierarchy diagram,
levels shown in documentation, 285
Hierarchy window, 91
History,
show, 626
Hotkeys,
assigning, 682
deleting, 682
HRESULT,
and error handling, 769
Hyperlinks,
in documentation text, 121

I

IBM DB2,
connecting through JDBC, 536
connecting through ODBC, 538
IBM DB2 for i,

© 2019-2025 Altova GmbH

IBM DB2 for i,
connecting through JDBC, 540
connecting through ODBC, 541
IBM Informix,
connecting through JDBC, 544
Icon,
Activity, 638
add to toolbar/menu, 676
Business Process Modeling Notation, 655
class, 640
Communication, 641
component, 643
Composite Stucture, 642
deployment, 644
Interaction Overview, 645
object, 646
Package, 647
Sequence, 650
show large, 684
State machine, 651
Timing, 652
use case, 653
XML Schema, 654
Icons,
visibility, 387
Ignore,
directories, 685
elements in list, 685
Images,
using as element background, 122
Import,
SQL database, 486
XMI to UModel, 568
Include,
.NET Framework, 164
dependency, 22
UModel project, 164
Insert, 298
action (CallBehavior), 298
action (CallOperation), 298
Composite Stucture elements, 402
Interaction Overview elements, 347
Package diagram elements, 407
simple state, 316
Timing diagram elements, 379
Instance,
diagram, 46
multiple class, and display of, 387
object, 46
Intelligent,
autocomplete, 31
Interaction operand, 356
multi-line, 356
Interaction operator,
defining, 356
Interaction Overview,
icons, 645
inserting elements, 347
Interaction Overview diagram, 346
Interaction use, 359

J

Java,
code and class file names, 220
code generation options, 175
code import options, 194
generating code, 170, 182
import annotations, 205
import binary files, 211
importing source code, 192
JavaScript,
error handling, 769
JDBC,
as data connection interface, 507
connect to Teradata, 564
setting up a connection (Windows), 525
JScript,
scripting with UModel, 707

L

Layer window, 95
Layers,
adding to diagrams, 132
deleting, 132
hiding, 132
locking, 132
showing, 132
Layout menu,
commands, 667
Legal information, 1291

© 2019-2025 Altova GmbH

License, 1295
information about, 1291
License metering,
in Altova products, 1293
Licensing,
package license files with installer, 1296
Lifeline, 354
attributes, 354
General Value, 379
typed property as, 354
Lifelline,
goto, 354
Line,
orthogonal, 53
Line break,
in actor text, 22
Lines,
changing the style of, 137
custom, 137
direct, 137
formatting, 46
moving, 137
orthogonal, 137
snap lines, 685
Links,
in generated documentation, 289
Local project, 612
Location,
move project, 153

M

Macros,
developing, 707, 713
enabling, 719, 730
running, 731
Mail,
send project, 660
Manifest,
artifact, 59
MariaDB,
connect through ODBC, 545
Menu,
add/delete command, 676
Merge,
3-way manual project merge, 278
3-way project merge, 276
create in Activity, 301
ignore directory, 685
projects, 276
Message, 360
arrows, 360
call, 360
create object, 360
go to operation, 360
inserting, 360
moving, 360
numbering, 360
Timing diagram, 385
Message flow, 451
Messages window,
reference, 96
Method,
Add raised exception, 387
Methods,
getter / setter, 387
Microsoft Access,
connecting through ADO, 514, 546
Microsoft Azure SQL, 547
Microsoft SQL Server,
connecting through ADO, 548
connecting through ODBC, 549
Model,
adding elements to, 83, 109
changing class name - effect in Java, 220
Model Tree window,
expanding or collapsing items, 83
exploring the project from, 83
icon reference, 83
showing or hiding items, 83
sorting items, 83
Modeling,
enhance performance, 169
Move,
project, 153
Moving message arrows, 360
Multiline, 22
Multi-line,
actor text, 22
interactionOperand, 356
use case, 22
MySQL,
connecting through ODBC, 554

© 2019-2025 Altova GmbH

N

Name,
region names - hide / show, 323
Native connections,
Azure CosmosDB, 532
CouchDB, 532
MariaDB, 532
MongoDB, 532
PostgreSQL, 532
SQLite, 532
New,
classifier, 218
New line,
in Lifeline, 343
ineractionOperand, 356
Node,
add, 59
add artifact, 59
styles, 90
Numbering,
messages, 360

O

Object,
create message, 360
diagram, 46
icons, 646
links - associations, 46
Object diagram, 405
Object model,
overview, 753
ODBC,
as data connection interface, 507
connect to MariaDB, 545
connect to Teradata, 565
setting up a connection, 528
ODBC Drivers,
checking availability of, 528
OLE DB,
as data connection interface, 507
Open Project,
source control, 612
OpenJDK,
as Java Virtual Machine, 525
importing binaries, 205
Operand,
interaction, 356
Operation,
autocompletion window, 685
Automatically add on Activity, 316
coloring, 392
goto from call message, 360
overriding, 387
reusing, 40
show / hide, 387
template, 283
Operations,
adding, 31
Operator,
interaction, 356
Options,
source control, 685
tools, 685
when generating documentation, 289
Oracle database,
connecting through JDBC, 556
connecting through ODBC, 557
Orthogonal,
line, 53
state, 323
Override,
class operations, 387
default SPL templates, 217
Overview window,
scrolling, 93

P

Package,
default packages, 83
icon reference, 83
icons, 647
Package diagram, 405
generating dependency diagram, 405
insert elements, 407
SysML, 478
PackageImport, 407

© 2019-2025 Altova GmbH

PackageImport, 407
viewing, 91
PackageMerge, 407
viewing, 91
Parameter,
template, 283
Path,
change project location, 153
SPL template path, 1276
Performance,
enhancement, 169
Pool,
swimlane, 453
PostgreSQL,
connecting through ODBC, 559
Pretty print,
in exported XMI files, 568
project on save, 153
Print preview,
options, 660
Process,
collapsed sub process, 450
expanded sub process, 448
Profiles,
applying to a package, 160, 412
built-in, 412
creating, 412
definition, 411
Progress OpenEdge database,
connecting through JDBC, 560
connecting through ODBC, 561
Project, 153
3-way manual merge, 278
3-way merge, 276
add or remove items, 83
add to source control, 622
create, 153
default code, 685
exploring, 83
file - updating, 217
generating documentation, 285
include UModel project, 164
insert package, 153
Merge, 276
modularize, 161
move, 153
open last on start, 685
remove from source control, 624
save - pretty print, 153
save open diagrams, 685
send by mail, 660
split into subprojects, 161
styles, 90
workflow, 153
Project menu,
commands, 664
Project open,
source control, 612
Project syntax,
checking, 96
Properties,
adding, 31
source control, 629
Properties window,
adding custom properties, 89
Property,
coloring, 392
reusing, 40
typed - show, 283
typed as lifeline, 354
Provider,
select, 612

R

Raised exception,
Adding, 387
Realization,
component, 53
generate ComponentRealizations, 218
Refactoring code,
class names - synchronization, 220
Reference, 659
Refresh status,
source control, 630
Region,
add to composite state, 323
Region name,
show / hide, 323
Reject source edits, 620
Relationships,
aggregation, 136
association, 110, 136
changing the style of, 137

© 2019-2025 Altova GmbH

Relationships,
composition, 136
dependency, 136
generalization, 110, 136
realization, 136
viewing, 139
Remove,
from source control, 624
Rename,
classifier, 218
Reply,
message - autogenerate, 360
Requirement diagram,
SysML, 480
Reset,
toolbar & menu commands, 677
Restore,
toolbars and windows, 669
Reverse engineering, 73
Root,
as package, 112
catalog - XMLSpy, 685
package/class synchronization, 217
Run native interface, 630

S

Save,
diagram as image, 660
SC,
syntax coloring, 392
Scripting Editor,
overview, 707, 709
Search,
diagrams, 114
elements, 114
text, 114
Send by mail,
project, 660
Sequence,
diagram SysML, 482
icons, 650
Sequence diagram, 351
adding code to, 375
combined fragment, 356
gate, 359
generate code from, 372
generate from code, 759
generate from Communication diag., 343
generate manually from code, 760
inserting elements, 352
interaction use, 359
lifeline, 354
messages, 360
state invariant, 360
Sequence diagrams,
generating from getters/setters, 371
generating from source code, 366
generating multiple, 371
Sequence flow, 451
Set,
getter / setter methods, 387
Setting,
synchronization, 217
Settings,
source control, 685
Share,
from source control, 625
Shortcut,
show in tooltip, 684
Shortcuts,
assigning, 682
deleting, 682
Show,
hide - slot, 387
hide- region name, 323
property as association, 283
Show differences, 628
Show history, 626
Show/hide,
attributes, operations, 387
Signature,
template, 281, 282
Silent Installation,
modify MSI file, 1296
package license files with installer, 1296
Slot,
show / hide, 387
Snap,
line - when dragging, 685
Snap lines, 22
Socket,
Ball and socket, 387
Software product license, 1295

© 2019-2025 Altova GmbH

Source control,
add to source control, 622
change provider, 630
Check In, 620
Check Out, 618
commands, 612
enable / disable, 615
get file, 616
get latest version, 616
installing a source-control plug-in, 607
open project, 612
options / settings, 685
properties, 629
refresh status, 630
remove from, 624
run native interface, 630
show differences, 628
show history, 626
Undo Check out, 620
Specialize,
generalize, 40
Speed,
enhancememt, 169
Spelling,
checking, 94
SPL, 1274
code blocks, 1275
conditions, 1286
foreach, 1287
subroutines, 1289
templates user-defined, 217
SPL templates,
template path, 1276
SQL,
importing into UModel, 486
SQL Azure, 547
SQL Server,
connecting through ADO, 514
connecting through ADO.NET, 519
connecting via JDBC, 525
Start,
with previous project, 685
State, 323
add activity, 316
composite, 323
define transition between, 316
insert simple, 316
orthogonal, 323
submachine state, 323
State changes,
defining on a timeline, 379
State invariant, 360
State machine,
composite states, regions, 323
diagram elements, 335
diagram SysML, 483
icons, 651
insert elements, 315
states, activities, transitions, 316
State Machine Diagram, 314
Stereotypes,
adding custom icons to, 421
adding custom styles to, 421
adding to the Properties window, 89
applying to elements, 148, 416
creating, 413, 416
definition, 146
example, 416
examples, 146, 411
Structural,
diagrams, 387
Styles,
applying to diagrams, 128
applying to elements, 122
applying to lines, 137
cascading, 122, 128, 137
precedence, 122, 128, 137
Styles window, 90
StyleVision,
customize generated documentation with, 294
customizing generated documentation with, 285
Sub Process,
collapsed, 450
expanded, 448
Submachine state,
add entry/exit point, 323
Subproject,
create from main project, 161
reintegrate into main project, 161
Swimlane,
pool, 453
Sybase,
connecting through JDBC, 563
Symbols,
visibillity icons, 387
Synchronization, 220

© 2019-2025 Altova GmbH

Synchronization, 220
class and code file name, 220
class name changes, 220
settings, 217
Synchronize,
root/package/class, 217
to new location, 153
Syntax coloring, 392
SysML,
activity diagram, 481
Block Definition Diagram, 469
creating diagrams, 468
Internal Block Diagram, 472
introduction, 468
package diagram, 478
Parametric diagram, 477
Requirement diagram, 480
Sequence diagram, 482
State Machine diagram, 483
Use Case diagram, 484
System DSN,
setting up, 528
System hierarchy,
Package diagram SysML, 478

T

Tagged values,
as enumerations, 413, 416
creating, 148, 413
definition, 147
example, 416
examples, 147
showing or hiding, 150
Template,
binding, 283
operation/parameter, 283
signature, 281, 282
Templates,
SPL templates, 1276
user-defined SPL, 217
Teradata,
connect through JDBC, 564
connect through ODBC, 565
Text annotation,
BPMN, 454
Tick mark,
Timing diagram, 382
TimeConstraint,
Timing diagram, 385
Timeline,
defining state changes, 379
Timing,
icons, 652
Timing diagram, 378, 379
DurationConstraint, 384
Event/Stimuls, 383
General Value lifeline, 379
inserting elements, 379
Lifeline, 379
Message, 385
switch between types, 379
Tick mark, 382
TimeConstraint, 385
Timeline, 379
Toolbar,
activate/deactivate, 677
add command to, 676
create new, 677
reset toolbar & menu commands, 677
show large icons, 684
Toolbars,
restore to default, 669
Tools,
options, 685
Tools menu,
adding custom commands to, 678
Tooltip,
show, 684
show shortcuts in, 684
Transition,
Add Activity diagram to, 316
define between states, 316
define trigger, 316
Trigger,
define transition trigger, 316
Tutorial,
sample files, 18
Type,
property - show, 283
Typed,
property - as lifeline, 354

© 2019-2025 Altova GmbH

U

UML,
Diagrams, 296
templates, 281
variables, 1276
visibility icons, 387
UModel,
Introduction, 13
Main features, 13
UModel API,
overview of, 752
UModel diagram icons, 637
UModel Plug-In,
creating with Visual Studio, 733
UModel Plug-in for Visual Studio,
installing, 572
UModel Plug-In Library,
adding reference to, 734
UModel projects,
opening, saving, creating, 19
UModel Type Library,
adding reference to, 736, 771
UMP, 153
change project location, 153
file extension, 153
Undo Check out, 620
Update,
project file, 217
URL,
open file from, 660
Usage,
dependency, 53
Use case,
adding, 22
association, 22
compartments, 22
diagram SysML, 484
icons, 653
multi-line, 22
Use Case diagram, 342
User defined,
actor, 22
User DSN,
setting up, 528
User interface,
configure using plug-in, 744
User-defined,
SPL templates, 217

V

Variables,
UML, 1276
VB.NET,
code generation options, 175
code import options, 194
generating code, 170
import binary files, 204
importing source code, 192
VBScript,
scripting with UModel, 707
Version control,
commands, 612
View,
to multiple instances of element, 387
View menu,
commands, 668
Viewpoints,
Package diagram SysML, 478
Visibility,
icons - selecting, 387
Visual Basic,
error handling, 769
Visual Studio,
addin UModel support to solutions, 573
automatic synchronization of code and model, 578
creating a UModel Plug-In, 733
Integrating UModel as a plug-in, 570
loading/unloading UModel projects, 577
synchronizing code with model, 578

W

Warnings,
during code engineering, 96
Windows,
restore to default, 669
Workflow,

© 2019-2025 Altova GmbH

Workflow,
project, 153
Working directory,
source control, 612

X

XMI,
import and export, 568
XML Schema,
creating diagrams, 430
declare namespace, 430
diagrams, 424
generating from model, 432
icons, 654
importing into a model, 425
modeling, 430, 432

© 2019-2025 Altova GmbH