User and Reference Manual

Altova UModel 2007 User & Reference Manual

Published: 2007

Table of Contents

Chapter 1

UModel

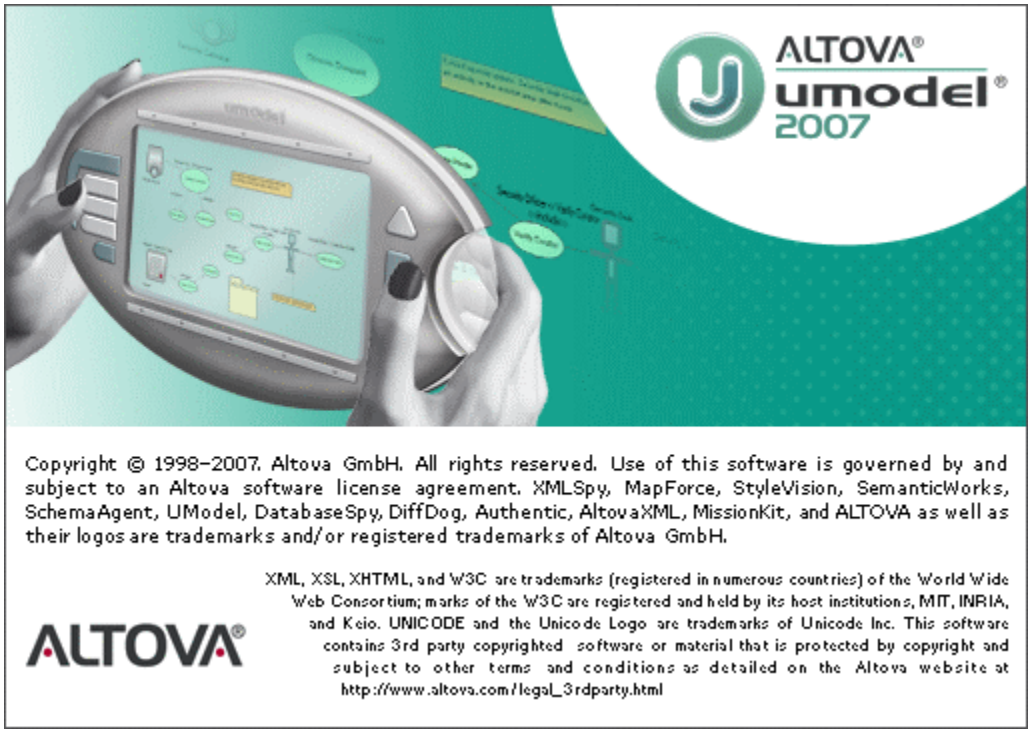
1 UModel

UModel™ 2007 is an affordable UML modeling application with a rich visual interface and superior usability features to help level the UML learning curve, and includes many high-end functions to empower users with the most practical aspects of the UML 2.1.1 specification.

UModel™ 2007 supports:

- all 13 UML 2.1.1 modeling diagrams
- XML Schema diagrams
- import of Java and C# binaries
- hyperlinking of diagrams and modeling elements
- context sensitive entry helpers
- syntax coloring in diagrams
- cascading styles
- customizable design elements
- unlimited Undo and Redo
- sophisticated Java and C# code generation from models
- reverse engineering of existing Java, C# source code
- complete round-trip processing allowing code and model merging
- XMI version 2.1 for UML 2.0 & 2.1 - model import and export
- generation of UModel project documentation

These capabilities allow developers, including those new to software modeling, to quickly leverage UML to enhance productivity and maximize their results.



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.

Chapter 2

Introducing UModel

2 Introducing UModel

The UML is a complete modeling language but does not discuss, or prescribe, the methodology for the development, code generation and round-trip engineering processes. UModel has therefore 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.
- Code, or model merging can be achieved 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.
- Code generation is customizable: the code-generation in UModel is based on SPL templates and is, therefore, completely customizable. Customizations are automatically recognized during code generation.
- Code generation and reverse-engineering currently support Java versions 1.4.x and 5.0, as well as C# versions 1.2 and 2.0. A single project can support both Java and C# code simultaneously.
- Support for UML templates and generics.
- XML Metadata Interchange (XMI version 2.1) for UML 2.0 or 2.1.1.
- When adding properties, or operations UModel provides in-place entry helpers to choose types, protection levels, and all other manner of properties that are also available in industrial-strength IDEs such as XMLSpy, Visual Studio .Net or Eclipse.
- Syntax-coloring in diagrams makes UML diagrams more attractive and intuitive.
- Modeling elements and their properties (font, colors, borders etc.) are completely customizable in an hierarchical fashion at the project, node/line, element family and element level.
- Customizable actors can be defined in use-case diagrams to depict terminals, or any other symbols.
- Modeling elements can be searched for by name in the Model diagram tab, Model Tree pane, Messages and Documentation windows.
- Class, or object associations, dependencies, generalizations etc. can be found/highlighted in model diagrams through the context menu.
- The unlimited levels of Undo/Redo track not only content changes, but also all style changes made to any model element.

Please note:

This document does not attempt to describe, or explain, the Unified Modeling Language (UML); it describes how to use the UModel modeling application, to model code and achieve round-trip engineering results.

Chapter 3

UModel tutorial

3 UModel tutorial

This tutorial describes, and follows, the general sequence used when creating a modeling project in UModel.

The major portion of the tutorial deals with the forward-engineering process, i.e. using UModel to create UML diagrams and generate code as the precursor to the round-trip engineering sections that follow. The round-trip engineering sections, describe the process from both code and model vantage points.

The tutorial describes the following UML diagrams, and how to manipulate the various modeling elements within them. The following diagrams and follow-on tasks are discussed:

Forward engineering process:

- Use cases
- Class diagrams
- Object diagrams
- Component diagrams
- Deployment diagrams

Round-trip process (model - code - model)

- Code generation from UModel
- Add a new operation to the external code
- Merge the external code back into UModel.

Round-trip process (code - model - code)

- Import code produced by XMLSpy from a directory (or from a project file)
- Add a new class to the generated model in UModel
- Merge the updated project with the external code.

The examples used in the **tutorial** are available in the default installation path/folder **c:\Program Files\Altova\UModel2007\UModelExamples\Tutorial\.**

BankView-start.ump
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 at this stage. Working through the tutorial adds new packages, model diagrams and many other elements that will acquaint you with the ease with which you can model applications using UModel. Please note that the syntax check function reports errors and warnings on this file, the tutorial shows you how to resolve these issues.

BankView-finish.ump
is the UModel project file that constitutes final state of the tutorial sample, if you have worked through it step by step. This project file is the one used when generating code and synchronizing it with UModel.

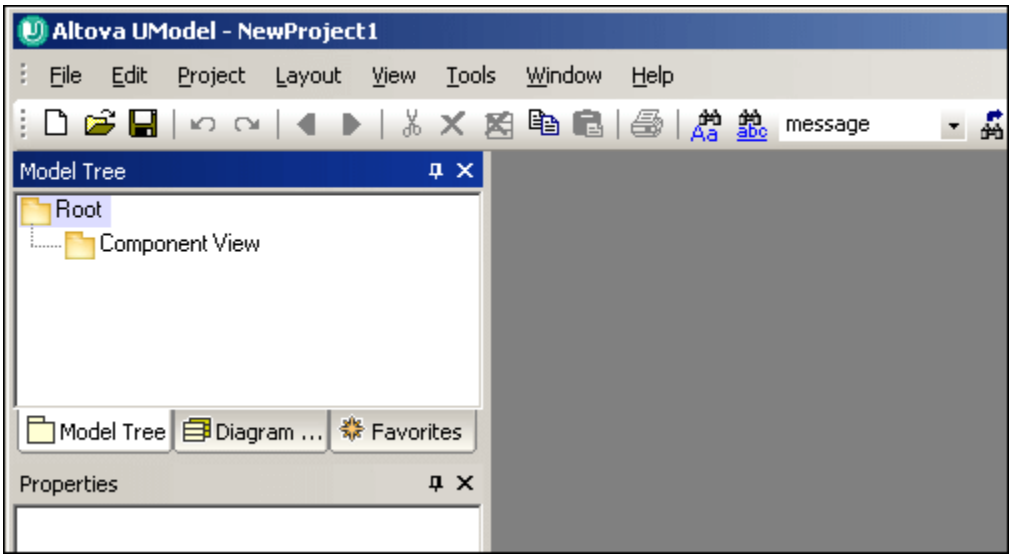
- The **OrgChart.zip** file supplied in the folder is used for the round-trip engineering process. Please unzip it in the ...**UModelExamples** folder before starting the section.

Additional example files for both **Java** and **C#** programming languages are also available in the same directory, i.e. **Bank_Java**.ump, **Bank_CSharp**.ump and **Bank_MultiLanguage**.ump. These project files also contain Sequence diagrams which are described later in this documentation.

3.1 Starting UModel

Having installed UModel on your computer:

1. Start UModel by double-clicking the UModel icon on your desktop, or use the **Start | All Programs** menu to access the UModel program.
   UModel is started with a default project "NewProject1" visible in the interface.

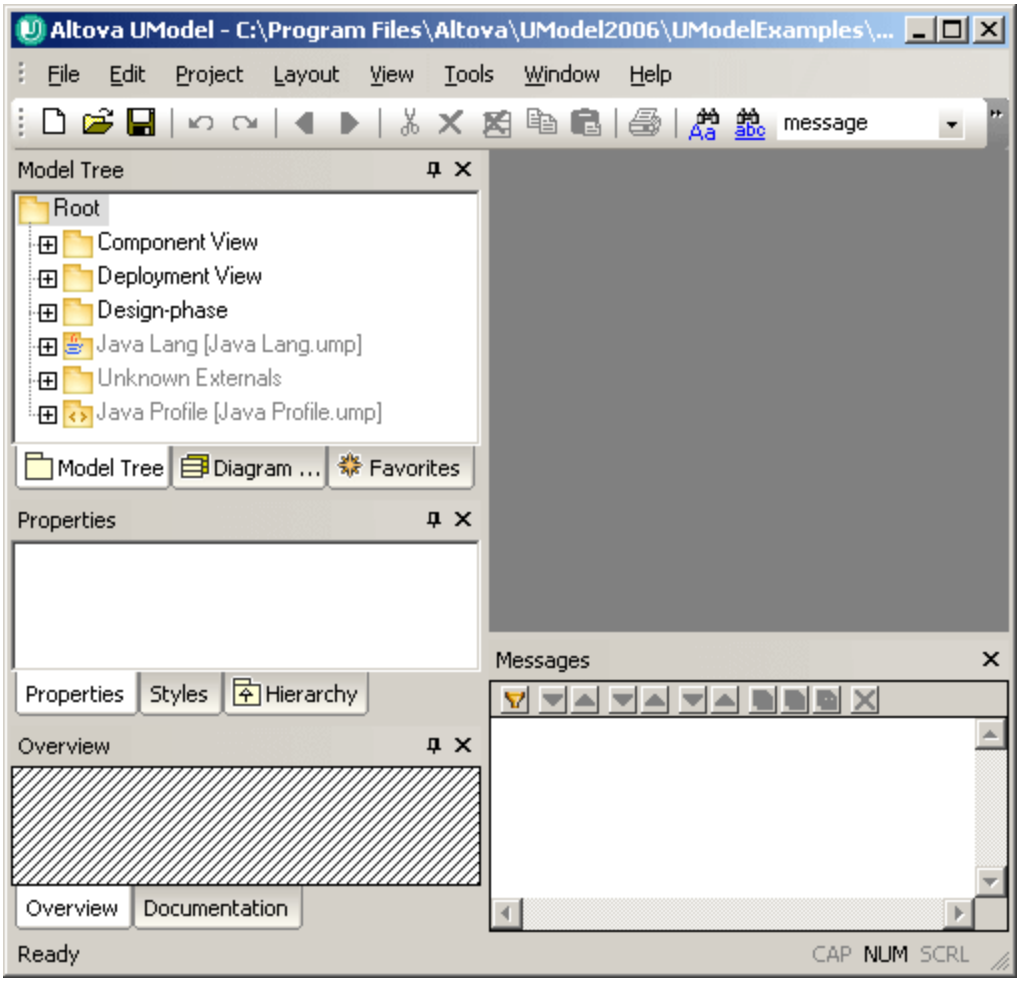


   Note the major parts of the user interface: the three panes on the left hand side and the empty diagram pane at right.

   Two default packages are visible in the Model Tree tab, "Root" and "Component View". These two packages cannot be deleted or renamed in a project.

To open the BankView-start project:

1. Select the menu option **File | Open** and navigate to the ...**UModelExamples** folder of UModel.
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.



The Model Tree pane supplies you with various views of your modeling project:

- The **Model Tree** tab contains and displays all modeling elements of your UModel project. Elements can be directly manipulated in this tab using the standard editing keys as well as drag and drop.
- The **Diagram Tree** tab allows you 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** tab is a user-definable repository of modeling elements. Any type of modeling element can be placed in this tab using the "Add to Favorites" command of the context menu.

The Properties pane supplies you with two views of specific model properties:

- The **Properties** tab displays the properties of the currently selected element in the Model Tree pane or in the Diagram tab. Element properties can defined or updated in this tab.
- The **Styles** tab 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 Overview pane displays two tabs:

- The **Overview** tab, which displays an outline view of the currently active diagram
- The **Documentation** tab which allows you to document your classes on a per-class basis.

Modeling element icon representation in the Model Tree

Package types:

UML Package
Java namespace root package
C# namespace root package
XML Schema root package
Java, C#, code package (package declarations are created when code is generated)

Diagram types:

Activity diagram
Class diagram
Communication diagram
Component diagram
Composite Structure diagram
Deployment diagram
Interaction Overview diagram
Object diagram
Package diagram
Sequence diagram
State Machine diagram
Timing diagram
Use Case diagram
XML Schema diagram

Element types:

An element that is currently visible in the active diagram is displayed with a blue dot at its base. In this case a class element.

Class Instance/Object
Class instance slot

Class
Property
Operation
Parameter

Actor (visible in active use case diagram)
Use Case
Component
Node
Artifact
Interface

Relations (/package)
Constraints

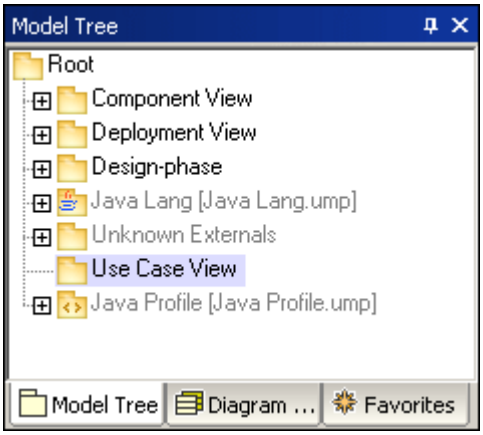
3.2 Use cases

The aim of this tutorial section is to:

- 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 size elements in the diagram tab.

To add a new package to a project:

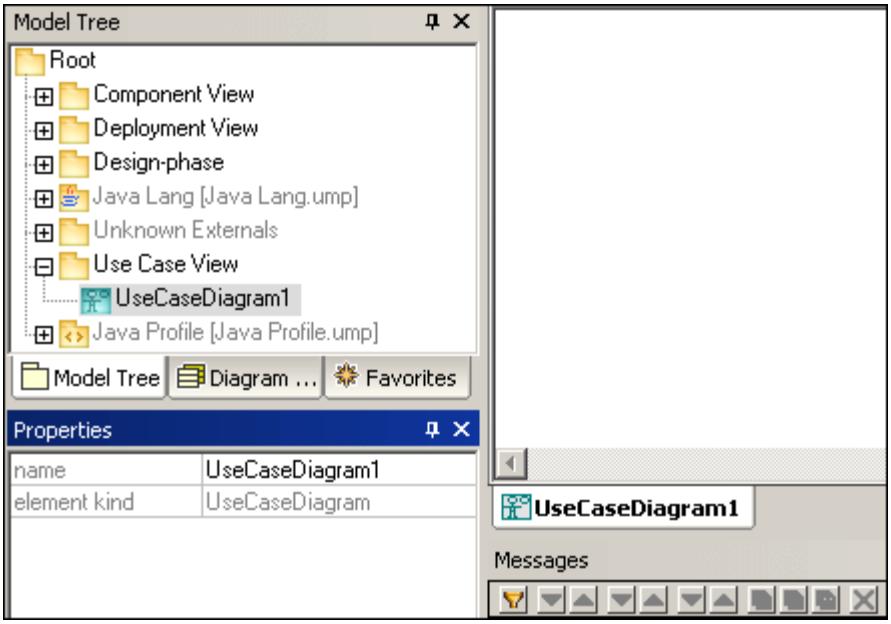
1. Right click the **Root** package in the Model Tree tab, and select **New | Package**.
2. Enter the name of the new package e.g. **Use Case View**, and press Enter.



Please see Packages for more information on packages and their properties.

Adding a diagram to a package:

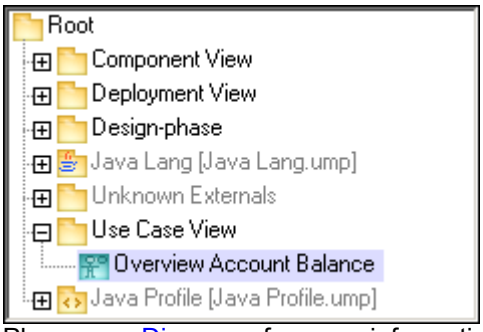
1. Right click the previously created Use Case View package.
2. Select **New | UseCase Diagram**.



A Use Case diagram has now been added to the package in the Model Tree view, and a diagram tab has been created in the diagram pane. A default name has been

provided automatically.

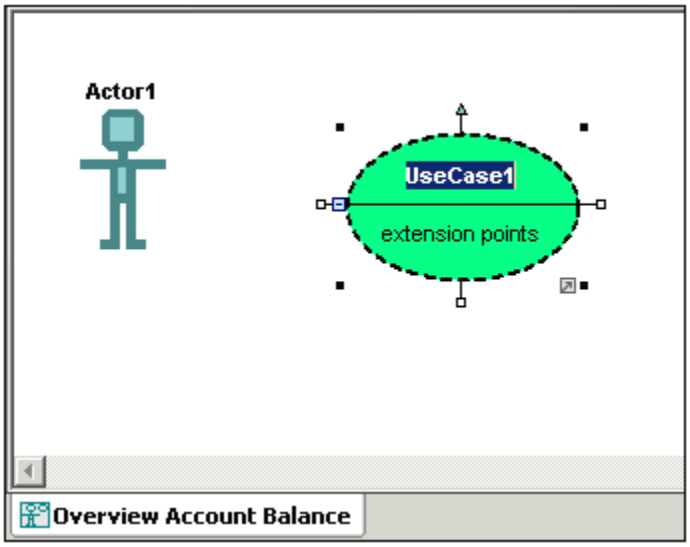
3. Double click the supplied name, in the Model Tree tab, change it to "Overview Account Balance", and press Enter to confirm.



Please see Diagrams for more information on diagrams and their properties.

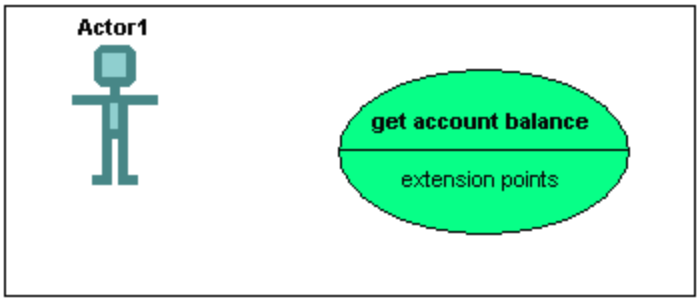
Adding Use case elements to the Use Case diagram:

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 icon  in the icon bar and click in the diagram tab 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 tab.



3. Change the title to "get account balance", press Enter to confirm. Double click the title if it is deselected.

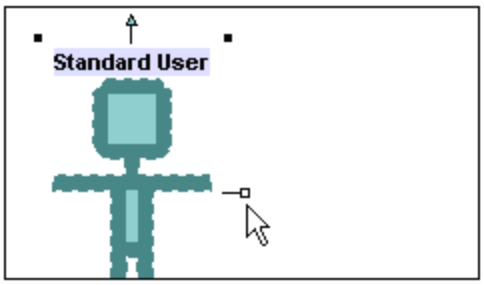
   Note that the use case is automatically resized to adjust to the text length.



Model elements have various connection handles and other items used to manipulate it.

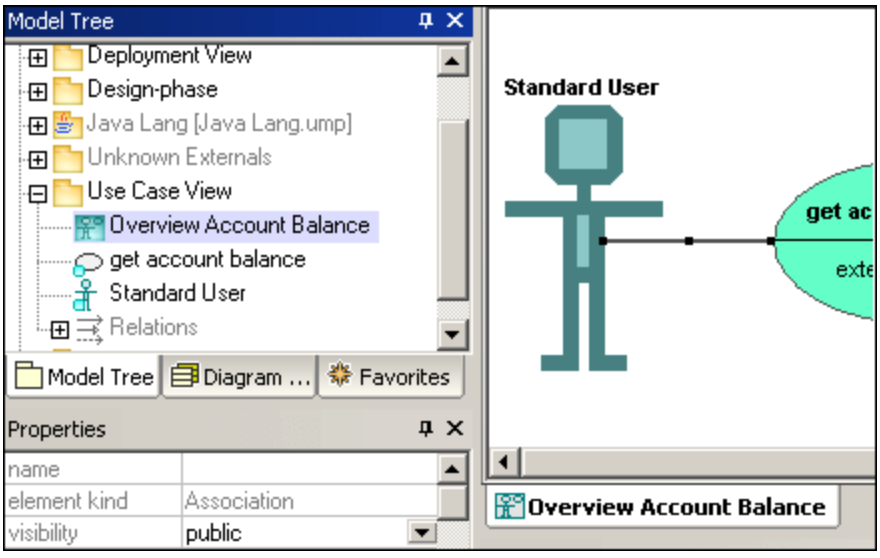
Manipulating UModel elements: handles and compartments

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.



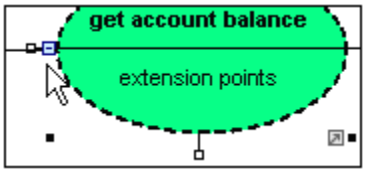
3. Click the handle, drag the Association line to the right, and drop it on the "get account balance" use case.

   An association has now been created between the actor and the use case. The association properties are also visible in the Properties tab. The new association has been added to Model Tree under the Relations item of the Use Case View package.

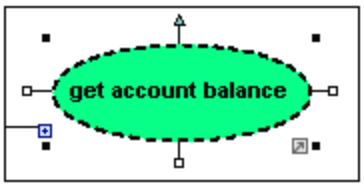


4. Click the use case and drag it to the right to reposition it.
   The association properties are visible on the association object.
5. Click the use case to select it, then click the collapse icon on the left hand edge of the

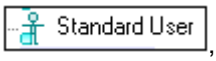
use case ellipse.



The **extension points** compartment is now hidden.



Please note:

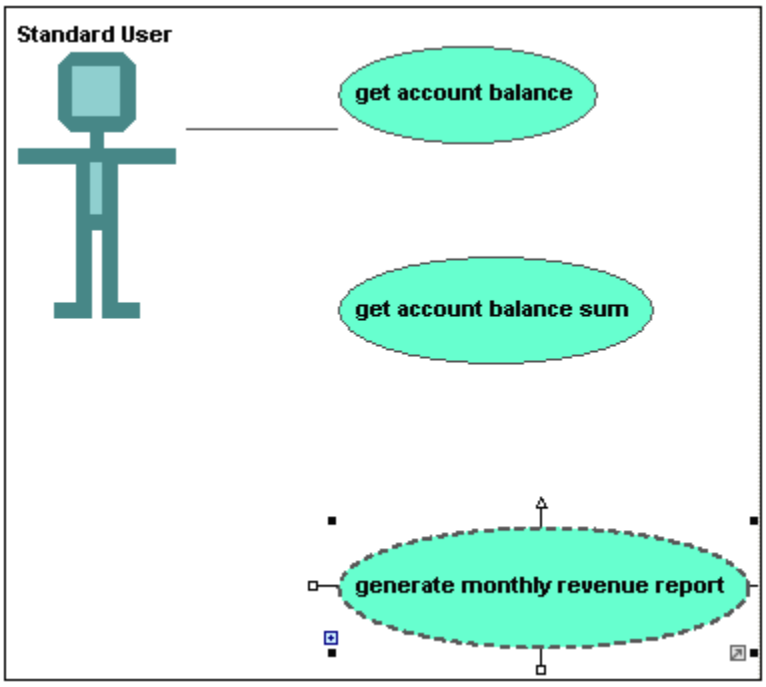
A blue dot next to an element icon 

, in the Model Tree tab, signifies that the element is visible in the current diagram tab. Resizing the actor adjusts the text field which can be multi line. A line break can be inserted into the text using CTRL+Enter.

Finishing up the use case diagram:

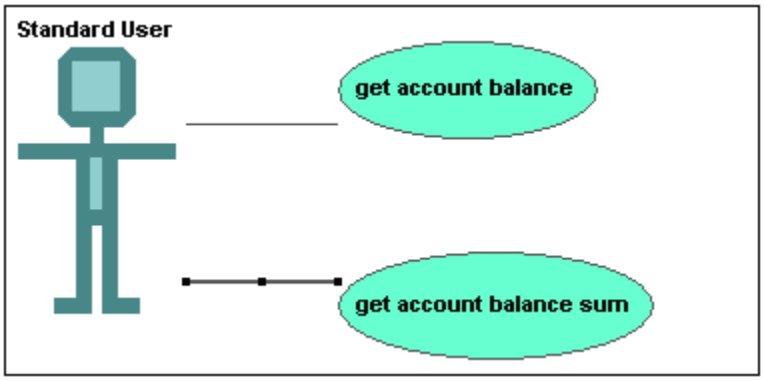
Using the methods discussed above:

1. Click the Use Case icon in the icon bar and **simultaneously** hold down the CTRL keyboard key.
2. Click at two different vertical positions in the diagram tab 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 on the collapse icon of each use case to hide the extensions compartment.



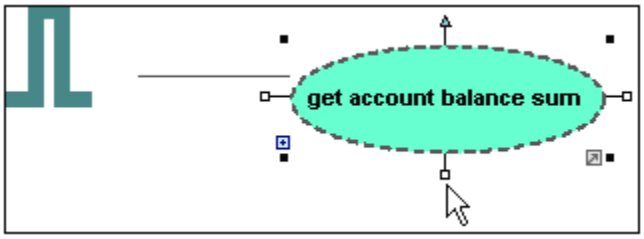
5. Click the actor and use the association handle to create an association between

Standard user and "get account balance sum".

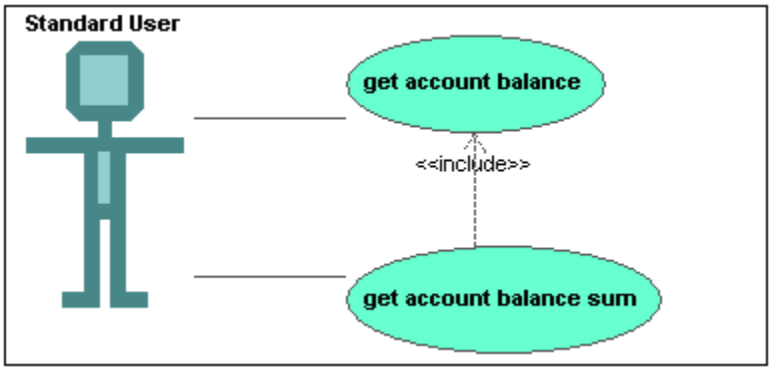


To create an "Include" dependency between use cases (creating a subcase):

1. 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".



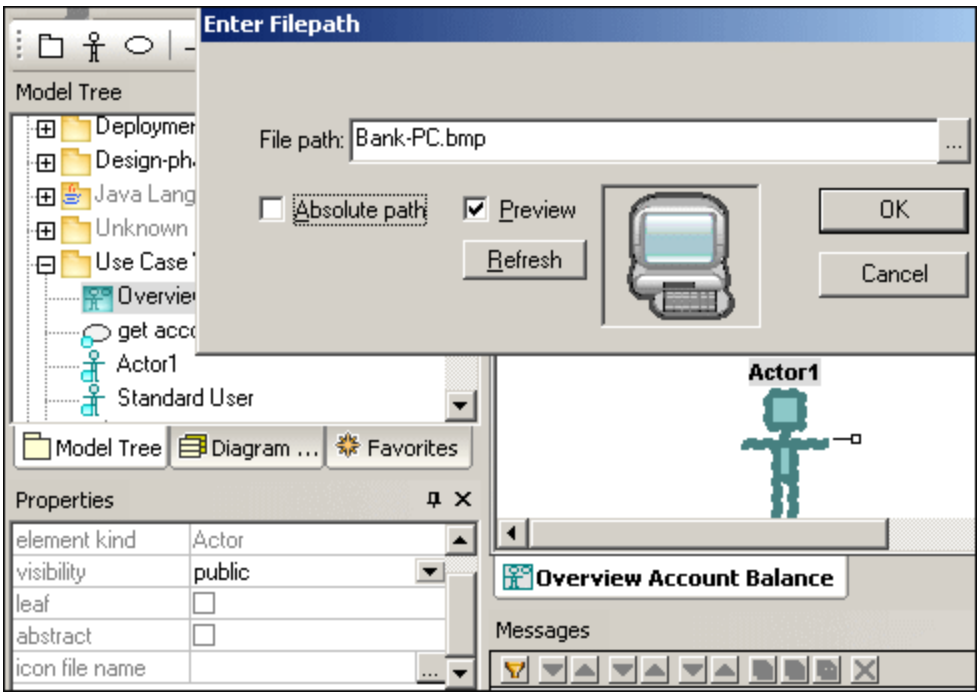
An "include" dependency is created, and the include stereotype is displayed on the dotted arrow.



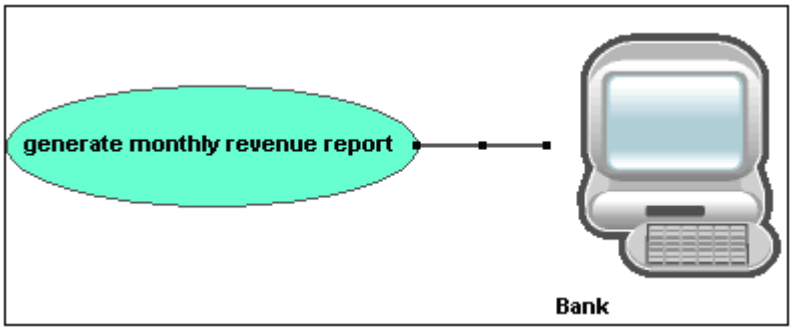
Inserting user-defined actors:

The actor in the "generate monthly revenue report" use case is not a person, but an automated batch job run by a Bank computer.

1. Insert an actor into the diagram using the Actor icon in the icon bar.
2. Rename the actor to Bank.
3. Move the cursor over to the Properties tab, and click the browse ... icon next to the "icon file name" entry.
4. Click the Browse icon to select the user-defined bitmap, Bank-PC.bmp.
5. Deselect the "Absolute Path" check box to make the path relative. Preview displays a preview of the selected file in the dialog box.



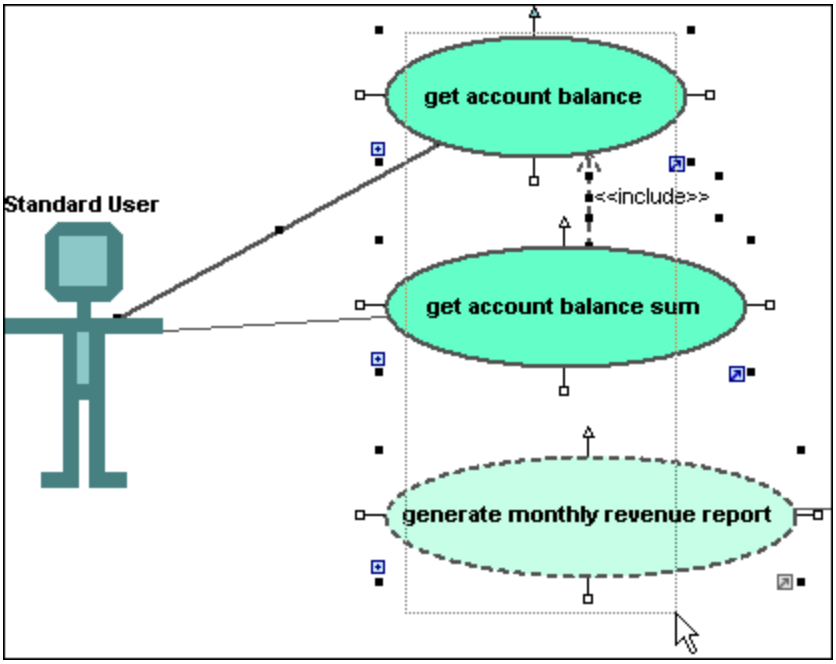
6. Click OK to confirm the settings and insert the new actor.
7. Move the new Bank actor to the right of the lowest use case.
8. Click the Association icon  in the icon bar and drag from the Bank actor to the "generate monthly revenue report" use case.
   This is an alternative method of creating an association.



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

Aligning and adjusting the size of elements:

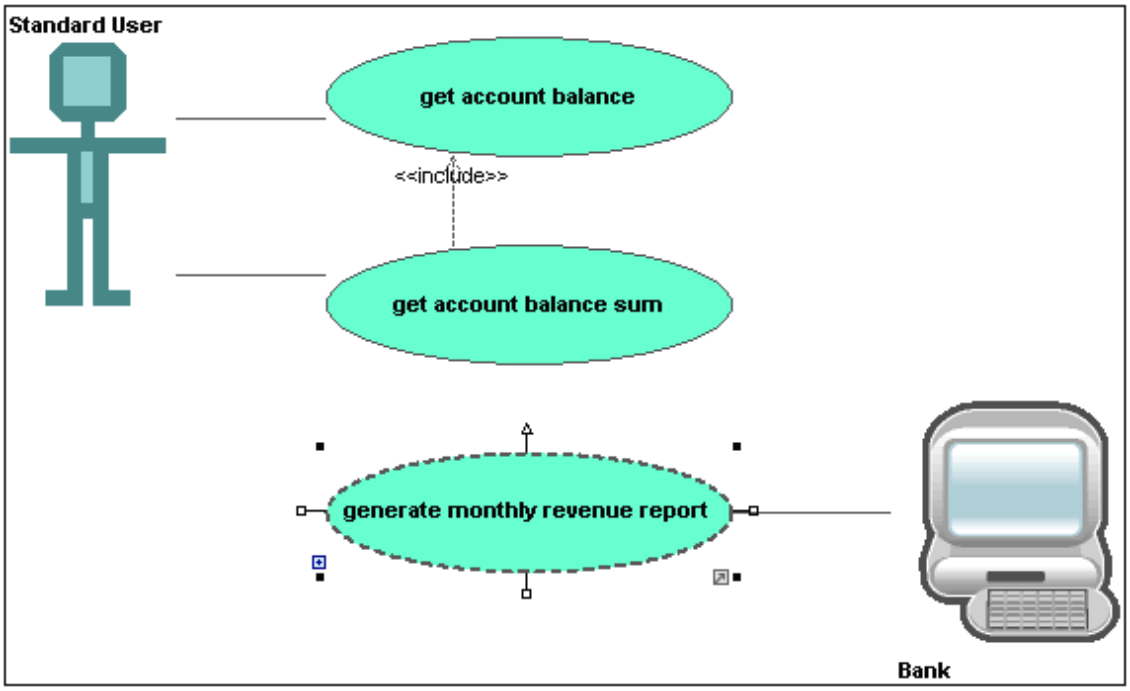
1. Create a selection marquee by dragging on the diagram background, making sure that you encompass all three use cases starting from the top.
   Note that the last use case to be marked, is shown in a dashed outline in the diagram, as well as in the Overview window.



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

2. Click the Make same size icon  in the title bar.
3. Click the Center Horizontally icon  to line up all the ovals.
   The use case elements are all centered and of the same size.

Please note:
You can also use the CTRL key to select multiple elements.



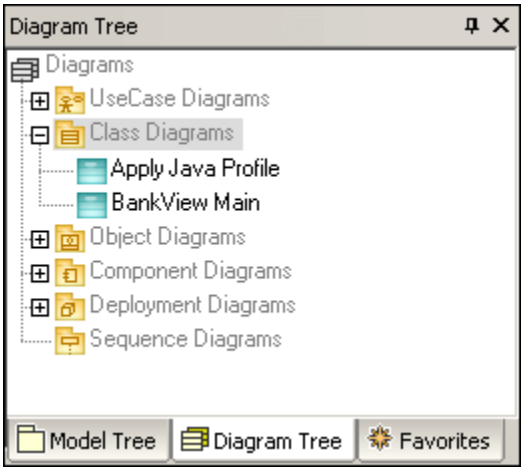
3.3 Class Diagrams

The aim of this tutorial section is to:

- Add a new abstract class called Account, as well as attributes and operations
- Create a **composite** association from Bank to Account

To open a different diagram in UModel:

1. Click the Diagram Tree tab.
2. Expand the Class Diagrams package to see its contents.

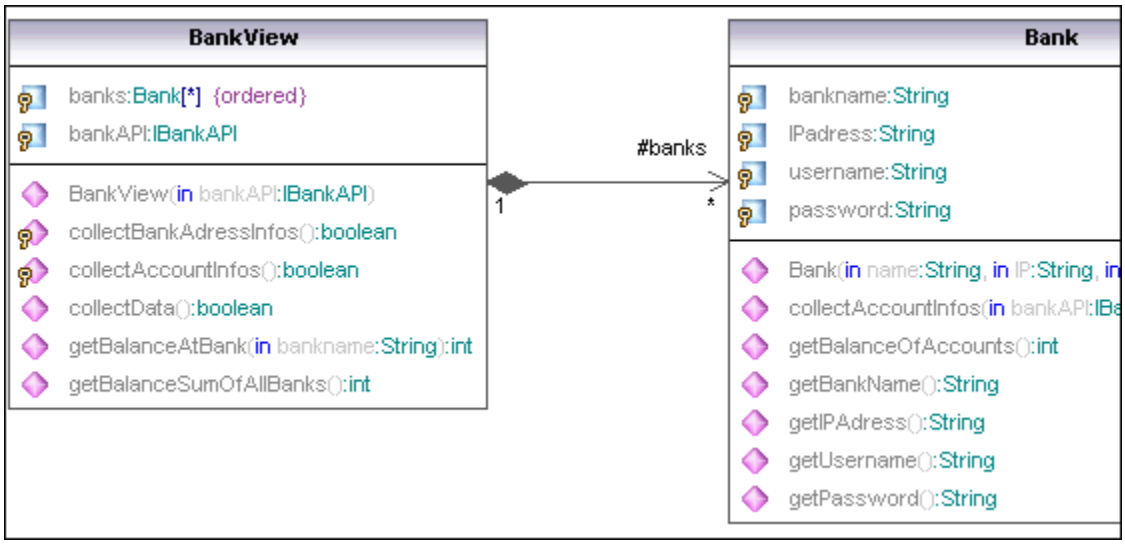


All class diagrams contained in the project are displayed.

3. Double click the **BankView Main** diagram icon.
   The Class diagram appears as a tab in the working area.

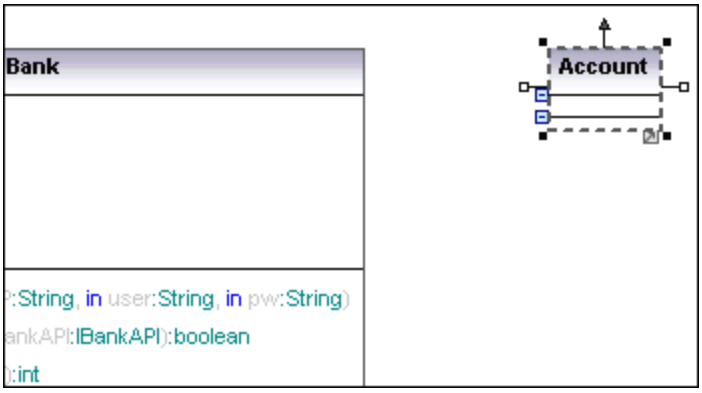
Please note:
You could of course, double click the Class diagram icon in the Model Tree tab below the BankView package to achieve the same thing.

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



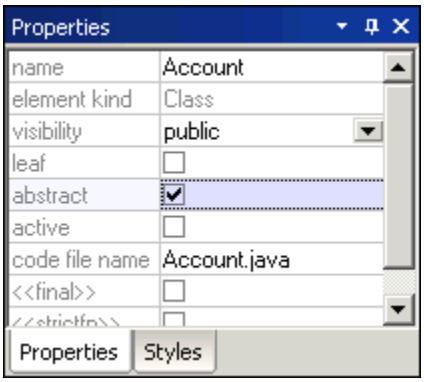
To add a new class and define it as abstract:

1. Click the class icon  in the icon bar, then click to the right of the Bank class to insert it.
2. Change the Class1 name to e.g. "**Account**", press Enter to confirm, (double click the name if it becomes deselected).

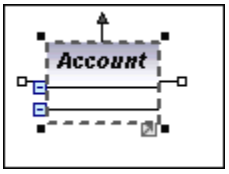


Note that the Properties tab displays the current class properties.

3. Click the "**abstract**" check box in the Properties pane to make the class abstract.
4. Click in the "code file name" text box, and enter **Account.java** to define the Java class.

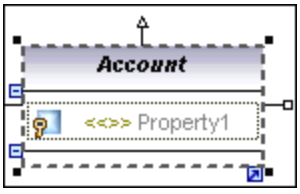


The class title is now displayed in italic, which is the identifying characteristic of abstract classes.

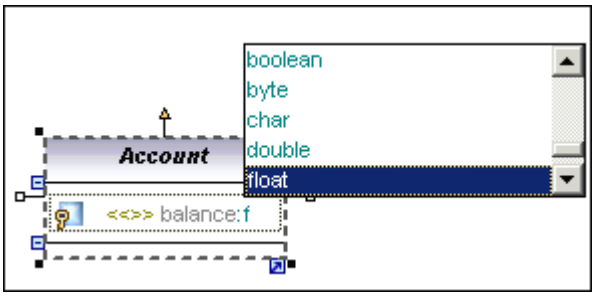


To add properties to a class:

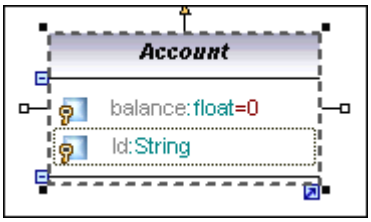
1. Right click the Account class and select **New | Property,** or press the **F7** key**.**
   A default property "Property1" is inserted with stereotype identifiers << >>.



2. Enter the Property name "**balance**", and then add a colon character "**:**".
   A drop-down list containing all valid types is displayed.
3. Enter the "**f**" character through the keyboard, and press Enter to insert the return value datatype "float".
   Please note that drop-down lists are case sensitive!

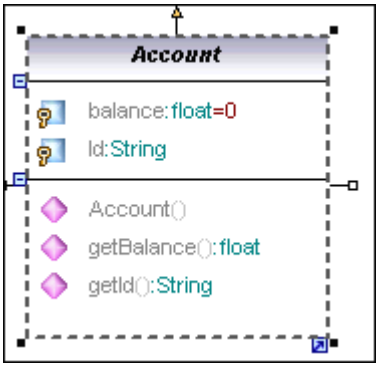


4. Continue on the same line by appending "**=0**" to define the default value.
5. Press the **F7** keyboard key to add a second property to the class.
6. Enter **Id**: and select **String** from the drop-down list.



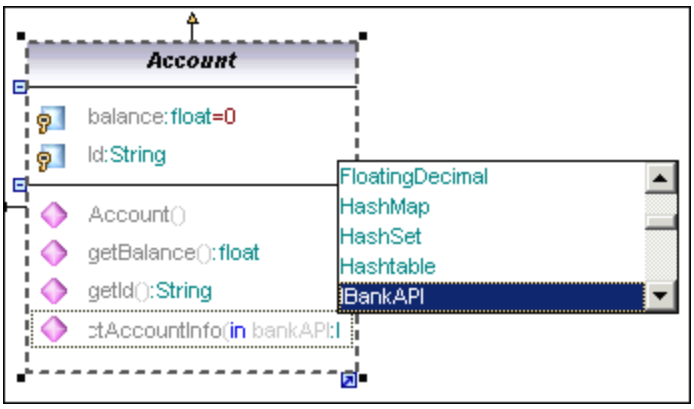
To add operations to a class:

1. Right click the Account class and select **New | Operation**, or press the **F8** key.
2. Enter **Account()** as the constructor.
   Using the method described above:
3. Add two more operations namely **getBalance:float** and **getId:String**.

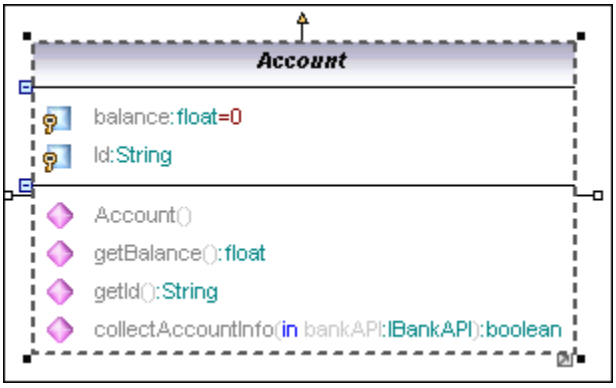


Using the **autocomplete function** while defining operations:

4. Create another operation, using F8, **collectAccountInfo** and enter the open parenthesis character "**(**".
   Entering the "i" character opens the drop-down list allowing you to select one of the operation direction parameters: in, inout, or out.
5. Select "in" from the drop-down list, enter a "space" character, and continue editing on the same line.
6 Enter "bankAPI" and then a colon.
7. Select **IBankAPI** from the drop-down list, add the close parenthesis character "**)**", and enter a colon ":".



8. Press the "b" key to select the boolean datatype, then Enter to insert it.
9. Press Enter to end the definition.



Please note:

Clicking the **visibility icon** to the left of an operation, or property, opens a drop-down list enabling you to change the visibility status.

Deleting class properties and operations from a class diagram:

1. Press F8 then Enter, to add a default operation "Operation1" in the Account class.
2. Click Operation1 and press the Del. key to delete it.

   A delete prompt appears asking if you want to delete the element from the project. Click Yes to delete Operation1 from the **class** as well as from the **project**.

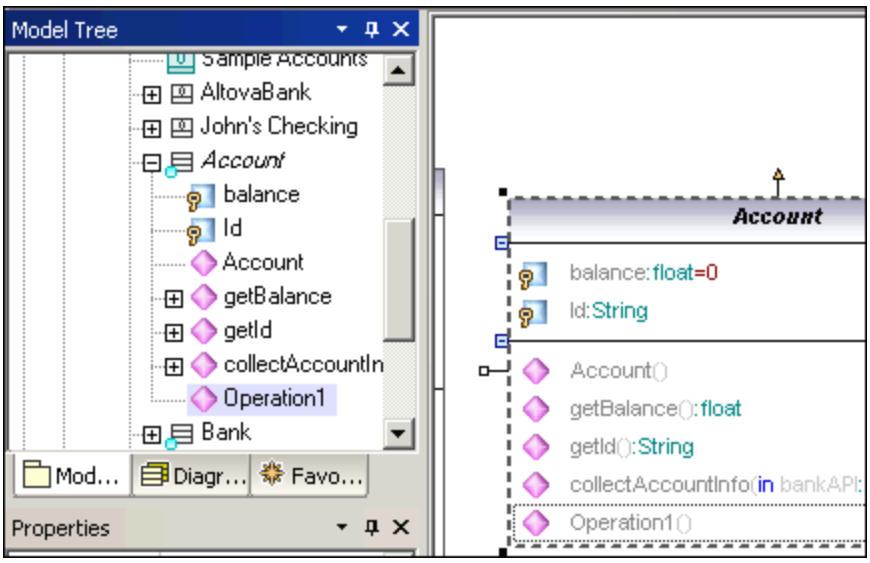
Please note:

If you only want to delete the operation from the class in the diagram, but **not** from the **project,** press the **CTRL + Del**. key.

Deleting (finding) class properties and options from the Model Tree:

Properties and options can also be deleted directly from the Model Tree. To do this safely, it is important to first find the correct property. Assuming you have inserted "Operation1" in the Account class (press F8, then Enter to insert):

1. Right click Operation1 in the Account class.
2. Select the option "**Select in Model Tree**".
   The Operation1 item is now highlighted under *Account* in the Model Tree tab.



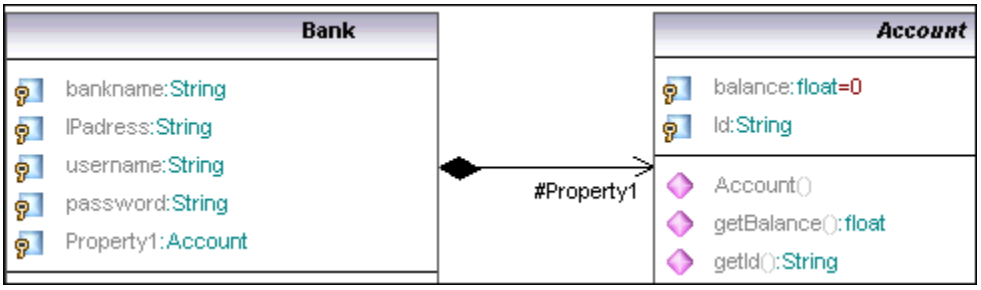
3. Press the **Del** key to delete the operation from the **class** and **project!**

Please note:

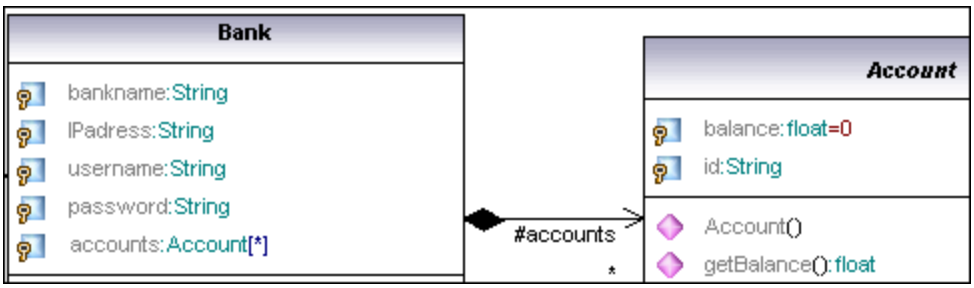
A delete prompt appears asking if you want to delete the element from the project. Click Yes to delete Operation1 from the **class** as well as from the **project**.. Undo can correct any number of mishaps at any time.

Creating an composition association between the Bank and Account classes:

1. Click the Composition icon  in the title bar, 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 an composite association arrow joins the two classes.



2. Double click the new **Property1** entry 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, and
4. Enter the open square bracket character "**[**" and select "*****" from the dropdown list, to define the **multiplicity**, and press Enter to confirm.



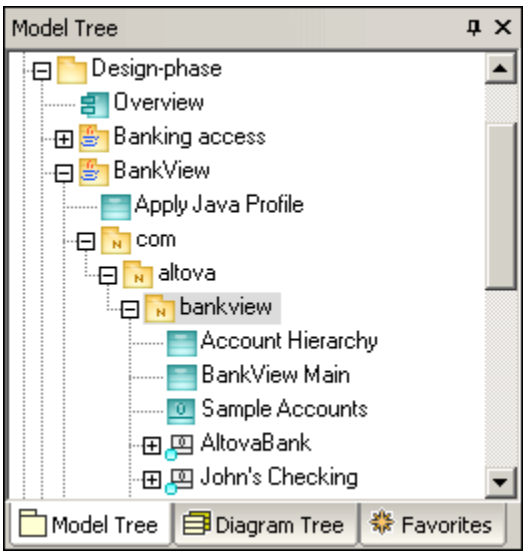
3.3.1 Creating derived classes

The aim of this tutorial section is to:

- Add a new **Class diagram** called Account Hierarchy to the project
- Insert existing classes, and create a new Savings account class
- Create three **derived** classes of the abstract base class Account, using Generalizations

To create a new Class Diagram:

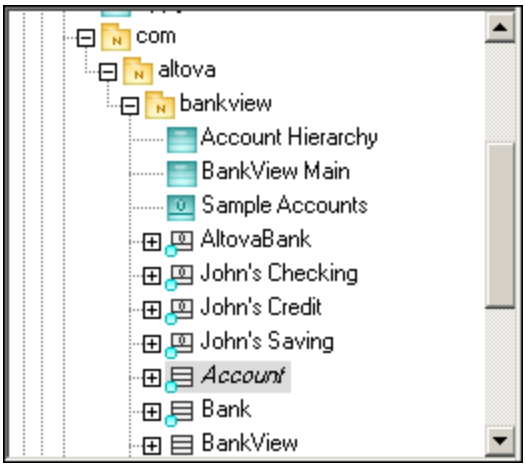
1. Right click the bankview **package** (under **Design-phase | BankView | com | altova**) in the Model Tree tab, and select **New | Class Diagram.**
2. Double click the new ClassDiagram1 entry and rename it to "Account Hierarchy", and press Enter to confirm.



The Account Hierarchy tab is now visible in the working area.

Inserting existing classes into a diagram:

1. Click the *Account* **class** in the BankView **package** (under **com | altova | bankview**), and

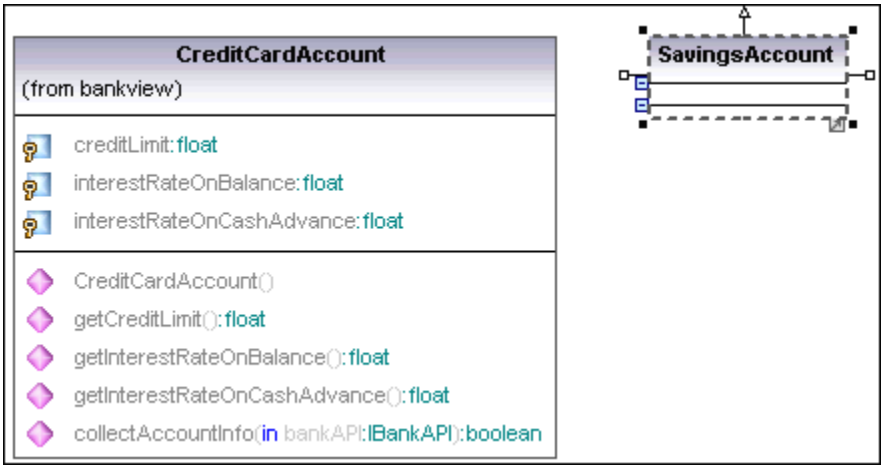


2. Drag it into the Account Hierarchy tab.
3. Click the **CheckingAccount** class (of the same package) and drag it into the tab.
4. Place the class below and to the left of the Account class.

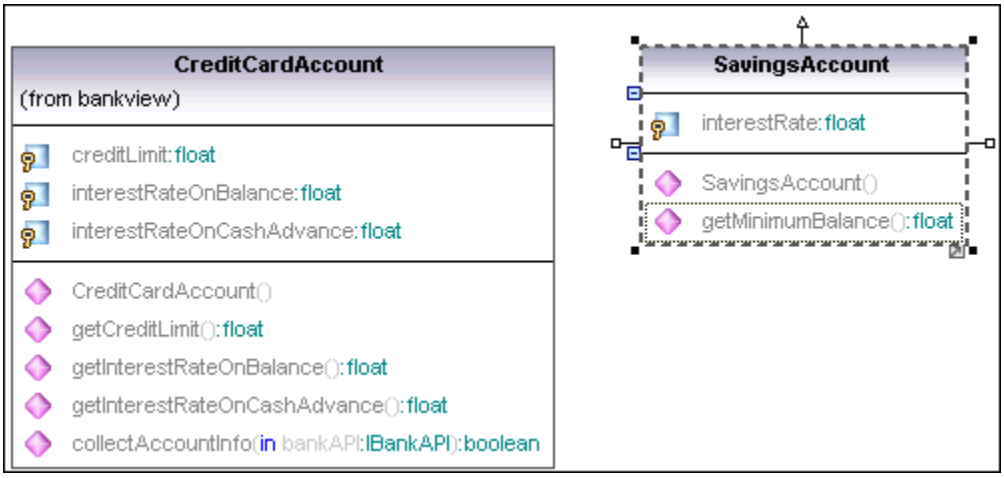
5. Use the same method to insert the **CreditCardAccount** class. Place it to the right of the CheckingAccount class.

Adding a new class:

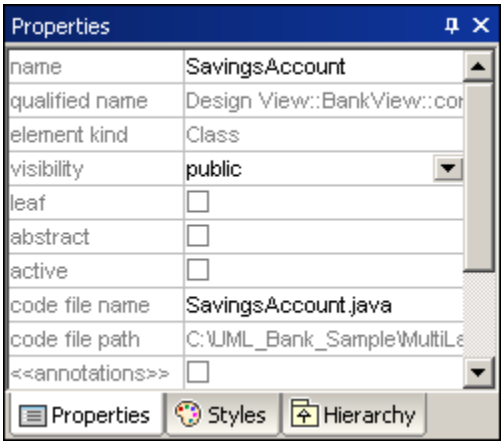
1. Right click the diagram background (to the right of CreditAccountClass) and select **New | Class**.
   A new class is automatically added to the correct package, i.e. BankView which contains the current class diagram **Account Hierarchy**.
2. Double click the class name and change it to **SavingsAccount**.



3. Press the **F7** key to add a new **property**.
4. Enter "**interestRate**", then a colon, and press "**f**" to select the float datatype from the dropdown list and press Enter twice to select and confirm the entry.
5. Press **F8** and add the **operation**/constructor **SavingsAccount()**.
6. Use the same method, F8, to add the operation **getMinimumBalance:float**.



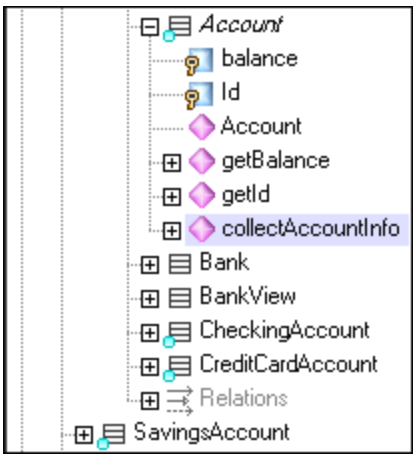
7. Click in the "code file name" text box, in the Properties tab, and enter **SavingsAccount.java** to define the Java code class.



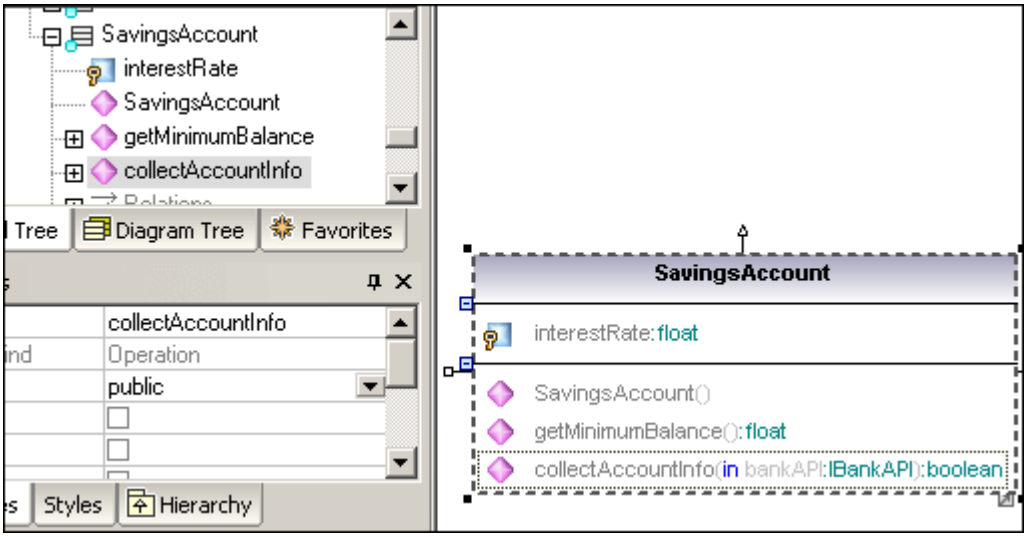
Reusing/copying existing Properties/Operations:
Properties and operations can be directly copied, or moved, from one class to another. This can be achieved using drag and drop, as well as the standard keyboard shortcuts:

- within a **class** in the diagram tab
- between **different** classes in the diagram tab
- in the Model Tree view
- between different UML diagrams, by dropping the copied data onto a different diagram tab.
  Please see "Cut, copy and paste in UModel Diagrams" for more information.

1. Expand the ***Account*** class in the **Model Tree**.
2. Right click the **collectAccountInfo** operation and select **Copy**.



3. Right click the **SavingsAccount** class in the Model Tree 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.

Please note:

You can use the Copy/Paste keyboard shortcuts (CTRL X, C, or V), as well as drag and drop in the Model Tree to achieve the same effect. You might have to disable the sort options to drop the operation between specific items.

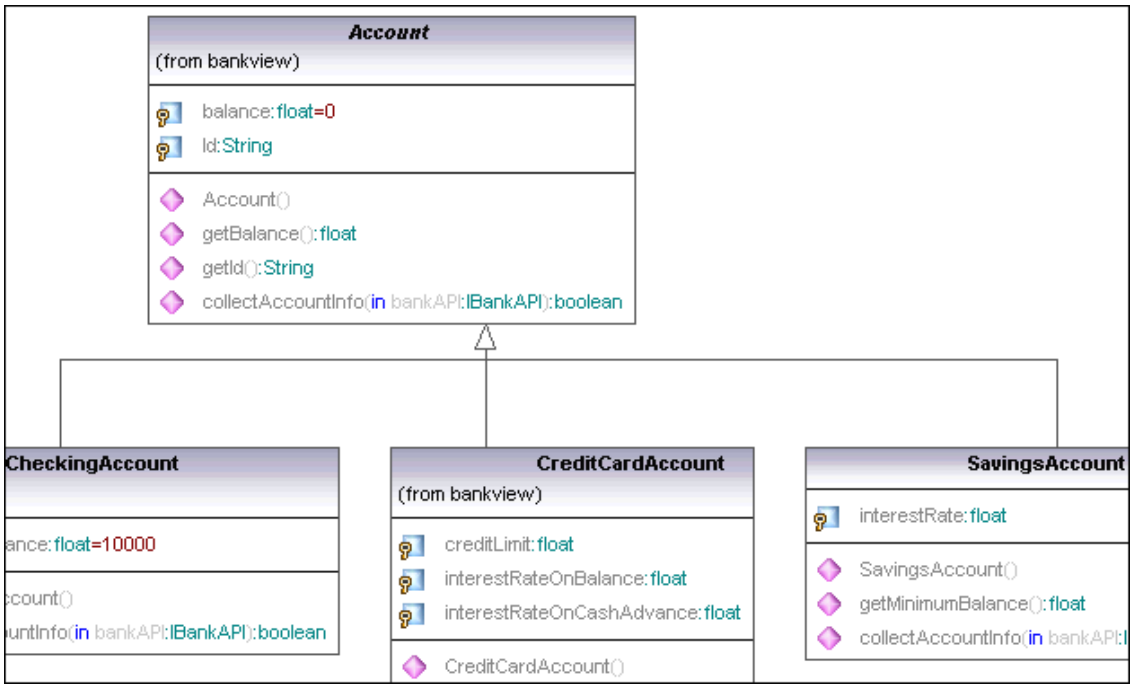
Creating derived classes - Generalization/Specialization:

At this point the class diagram contains the abstract class, Account, as well as three specific Account classes. We now want to define, or create a generalization/specialization relationship between Account and the specific classes i.e. to create three derived concrete classes.

1. Click the Generalization icon  in the icon bar and **hold down** the **CTRL** key.
2. Drag from **CreditCardAccount** (the class in the middle) and drop on the Account class.
3. Drag from the **CheckingAccount** class and drop the **arrowhead** of the previously created generalization.
4. Drag from the **SavingsAccount** class and drop the arrowhead of the previously created

generalization: release the CTRL key at this point.

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



3.4 Object Diagrams

The aim of this tutorial section is to:

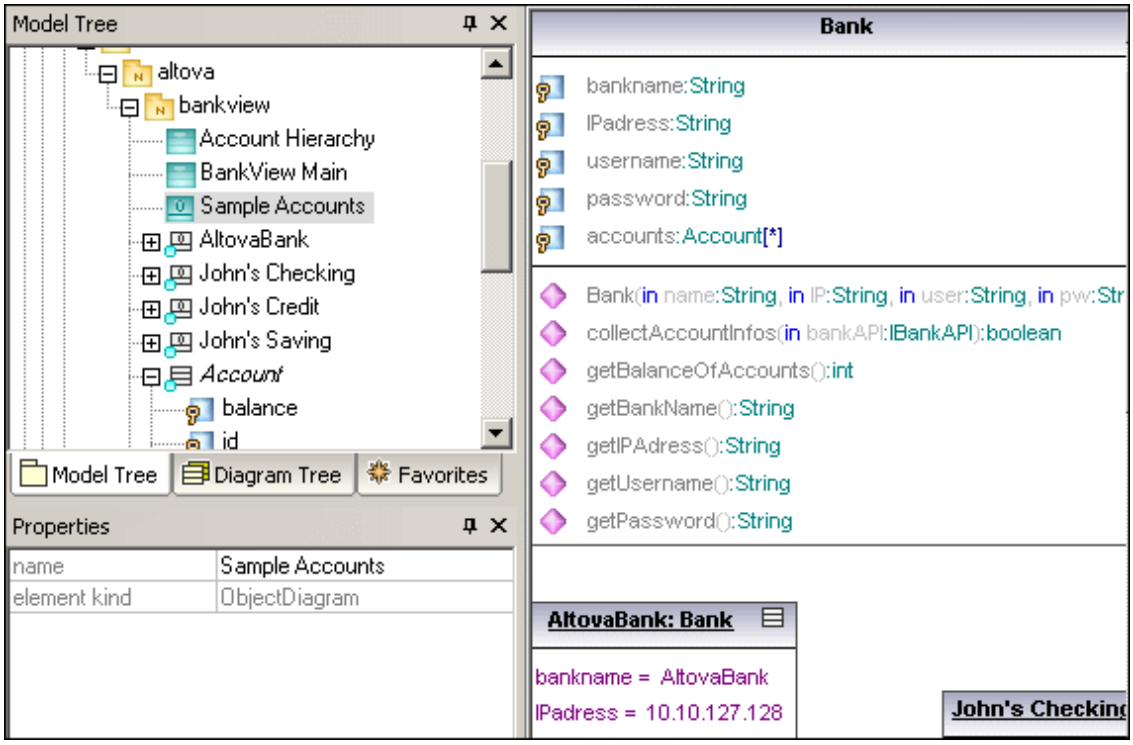
- Show how **class** and **object** diagrams can be combined in one diagram, to give you a snapshot of the objects at a given point of time
- Create **Objects/Instances** and define the relationships between them
- Format association/links
- Enter real-life data into objects/instances

To open the Object diagram:

1. Double click the **Sample Accounts** diagram icon under the **bankview** package (or under Object Diagrams in the Diagram Tree tab).

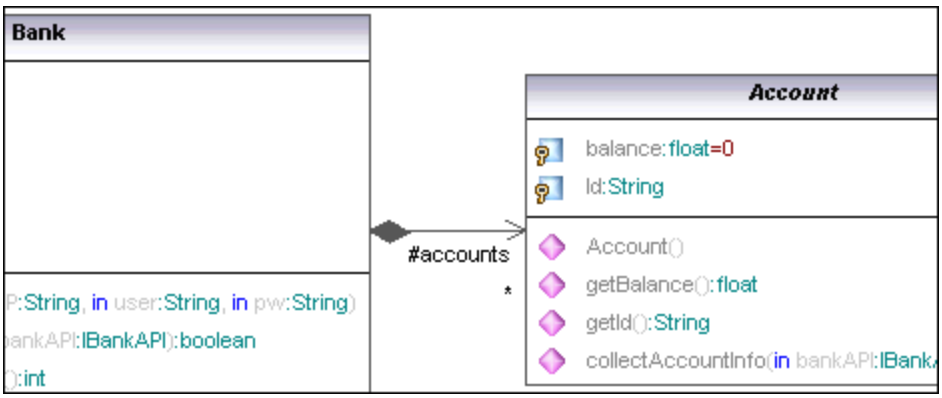
   The Bank class and two related objects/instances are displayed in the object diagram.

   AltovaBank:Bank is the object/instance of the Bank class, while **John's checking: CheckingAccount** is an instance of the class CheckingAccount.



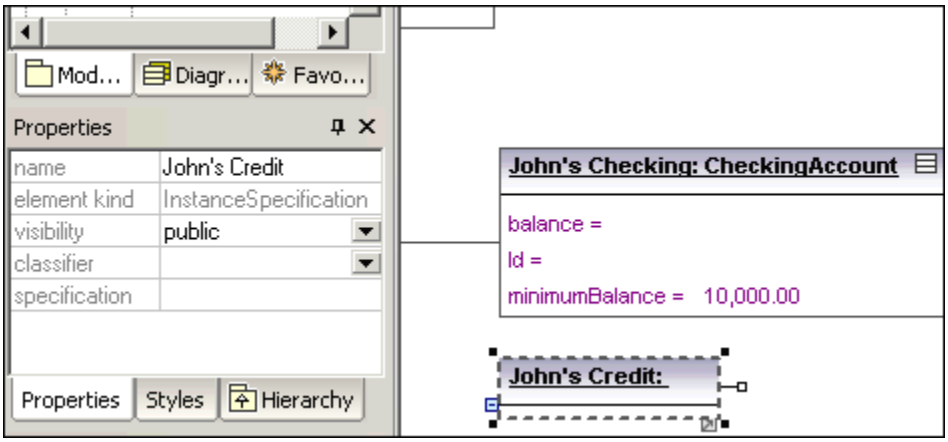
Inserting a class into an Object diagram:

- Click the *Account* class icon *Account* in the **Model Tree**, and drag it into the "Sample Accounts" tab.
  The composite association defined previously, in BankView Main diagram, is automatically created.



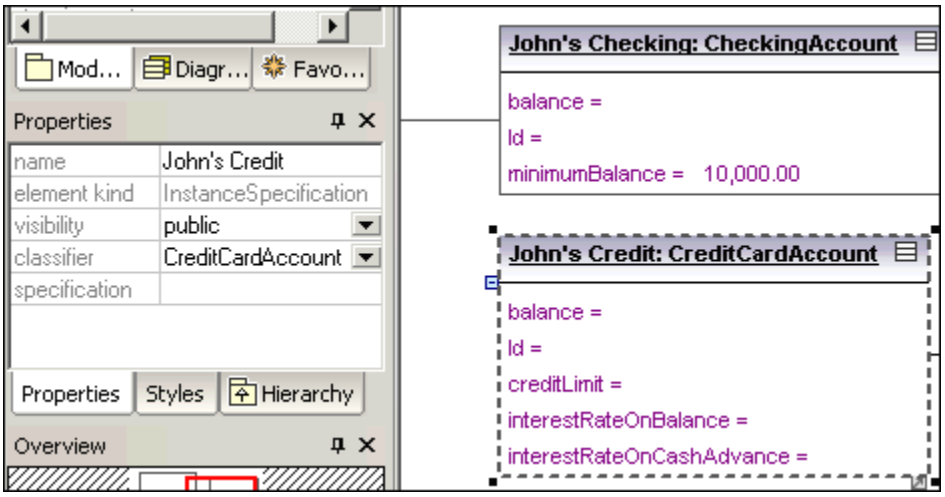
To add a new object/instance by selecting its type:

1. Click the **InstanceSpecification** icon  in the icon bar, then click under the John's Checking object in the diagram tab.
2. Change the name of the instance to **John's Credit**, and press Enter.



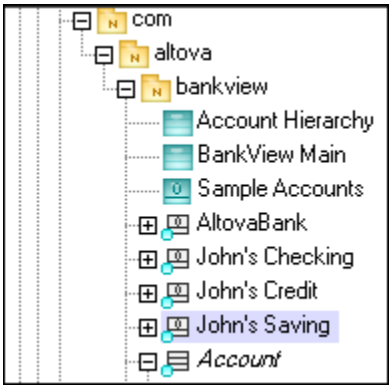
While the instance is active, all its properties are visible in the Properties tab.

3. Click the **classifier** combo box and select the entry **CreditCardAccount** from the drop-down list.



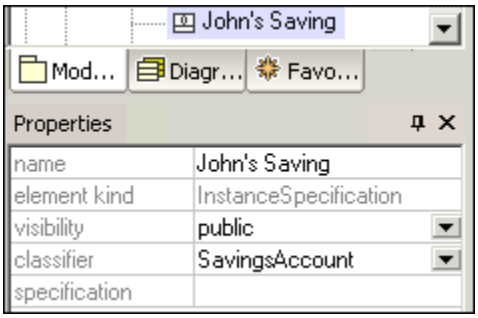
To add a new object in the Model Tree view (then insert it into a diagram):

1. Right click the **bankview** package in the **Model Tree tab**, and select **New | InstanceSpecification**.
2. Change the default object name to **John's Saving**, and press Enter to confirm. The new object is added to the package and sorted accordingly.

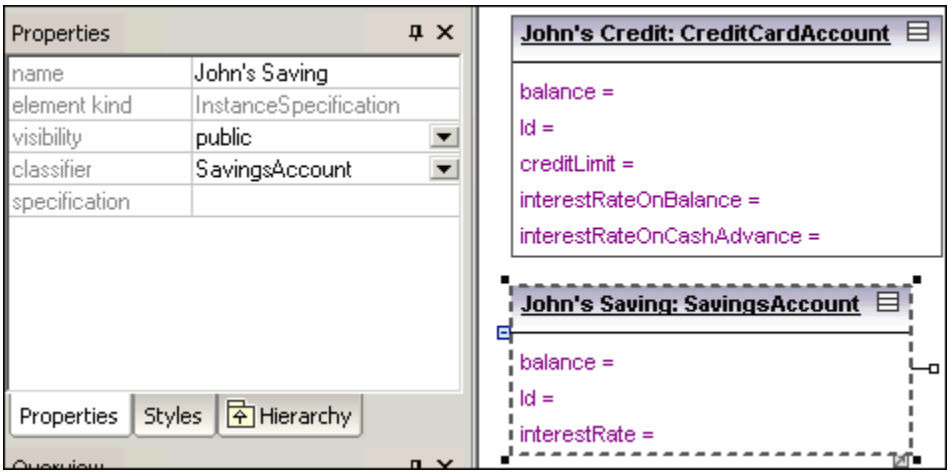


While the object is still selected in the Model Tree tab,

3. Click the **classifier** combo box, in the **Properties** tab, and select **SavingsAccount**.



4. Drag the John's Saving object/instance from the Model Tree tab, into the Sample Accounts tab, placing it below John's credit.

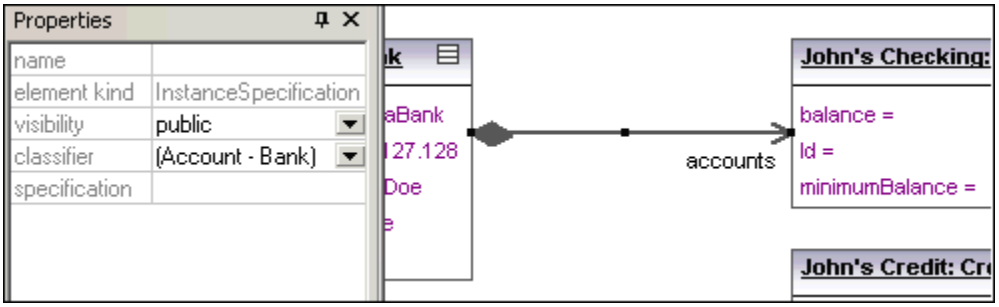


Creating "links" between objects:

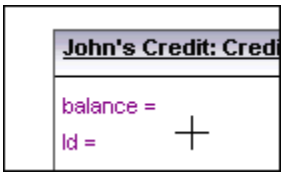
Links are the instances of class associations, and describe the relationships between objects/instances at a fixed moment in time.

1. Click the existing link (association) between the **AltovaBank** and John's Checking.
2. In the Properties tab, click the **classifier** combo box and select the entry **Account** -

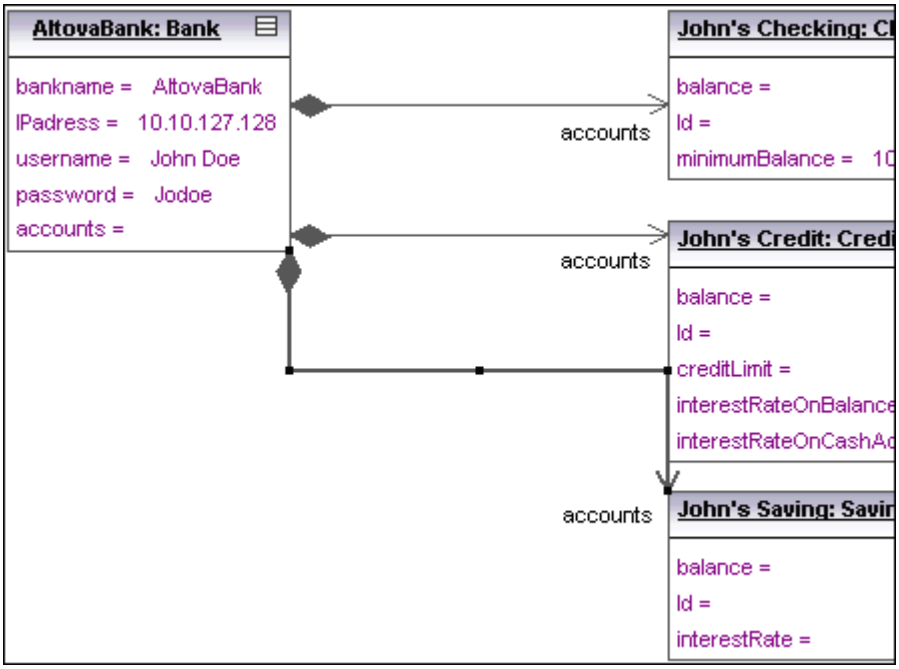
Bank.
The link now changes to a composite association, in accordance with the class definitions.



3. Click the **InstanceSpecification** icon  in the icon bar, and position the cursor over the John's Credit class.
The cursor now appears as a + sign.



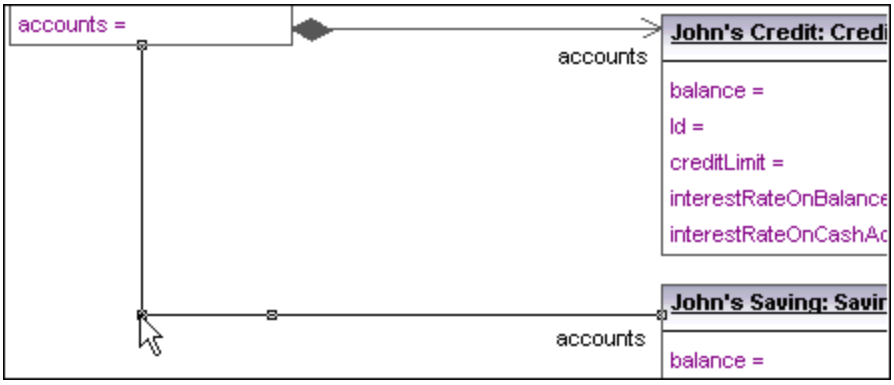
4. Drag from **John's Credit** object to AltovaBank to create a link between the two.
5. Use the **classifier** combo box in the Properties tab to change the link type to **Account - Bank**.
6. Use the method outlined above to create a link between **John's Saving** and AltovaBank.



Please note:
Changes made to the association type in any class diagram, are now automatically updated in the object diagram.

Formatting association/link lines in a diagram:

1. Click the lowest link in the diagram, if not active, and drag the corner connector to the left.
   This allows you to reposition the line both horizontally and vertically.

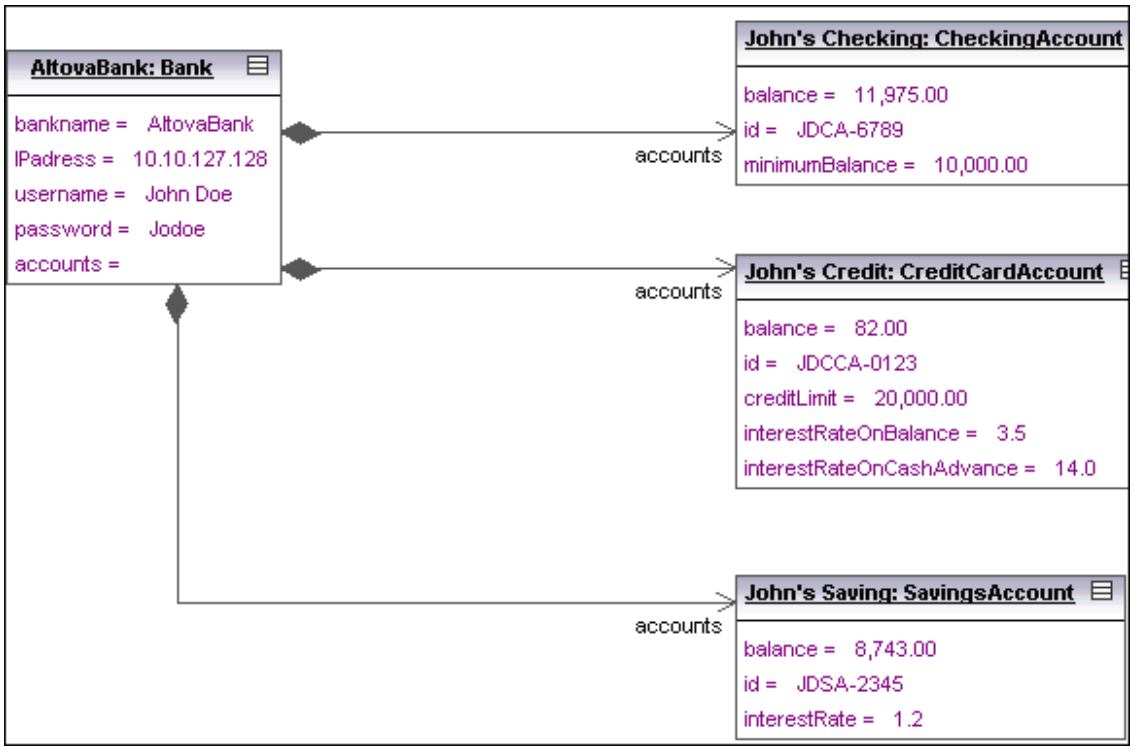


   Use this method to reposition links in the diagram tab.

Entering sample data into objects:

The instance value of an Attribute/Property in an object is called a **slot**.

1. Click in the respective slots of each object and enter sample data.
2. E.g. in **John's Checking** object, double click in the **balance** slot and enter 11,975.00 as the balance.
3. Fill in the rest of the data to give yourself an idea of the current instance state.



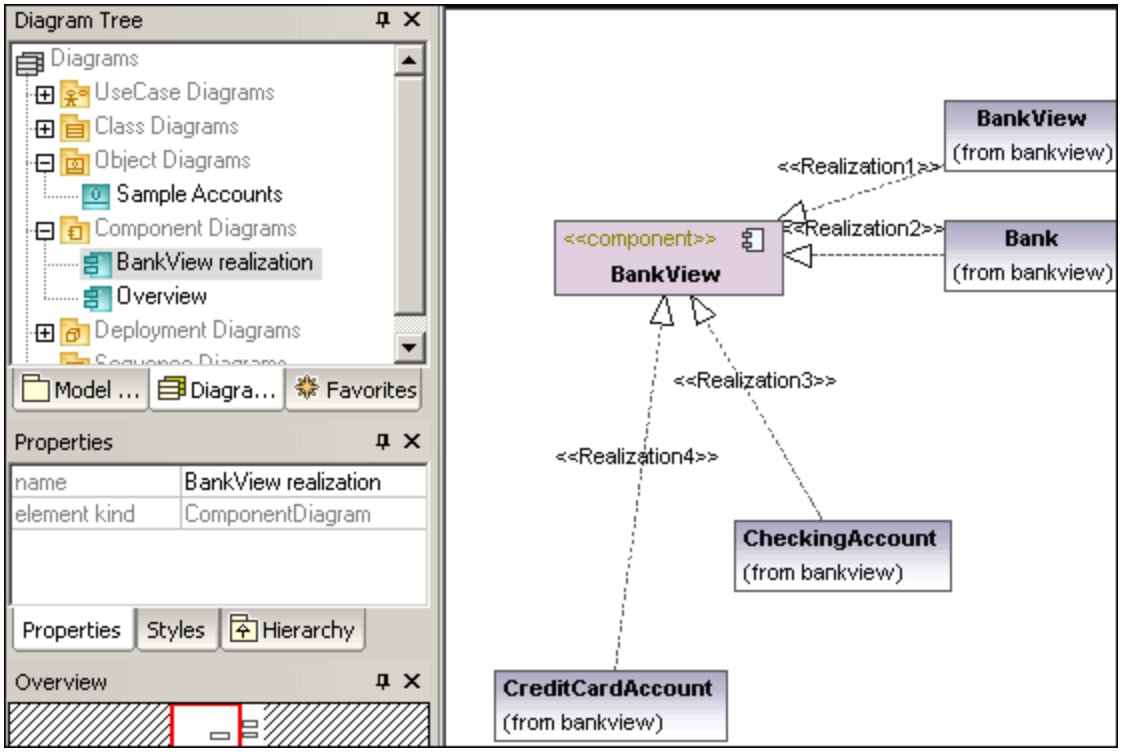
3.5 Component Diagrams

The aim of this tutorial section is to:

- Show how to insert classes into a component diagram
- Create realization dependencies between the classes and the BankView component
- Show how to change line properties
- Insert components into a component diagram, and create usage dependencies to an interface

To open the component diagram:

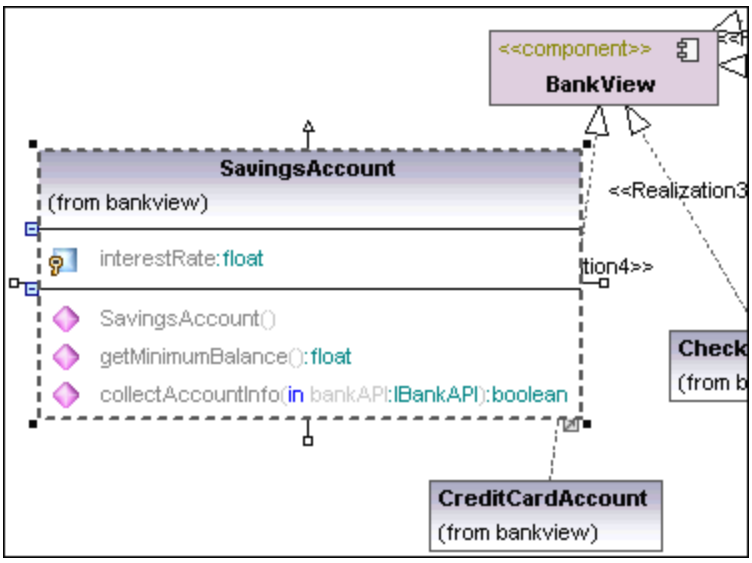
1. Click the Diagram Tree tab, expand the **Component Diagrams** component and double click the "BankView realization" diagram icon.
   The "BankView realization" component diagram is displayed.



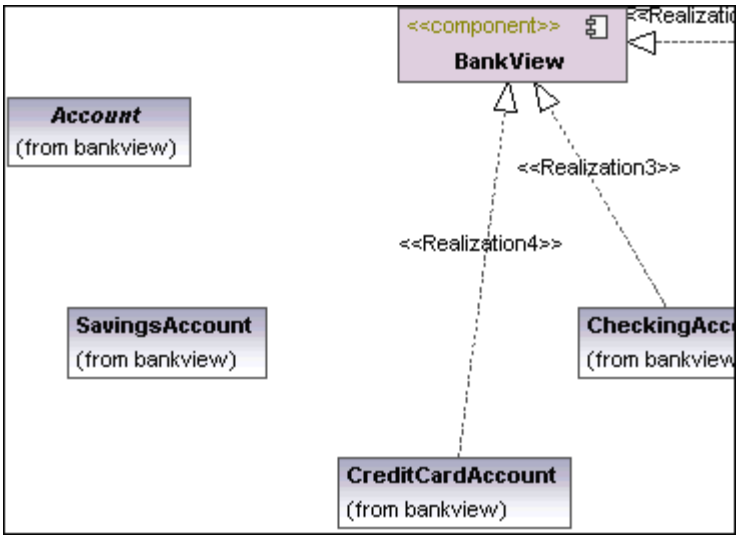
2. Switch back to the Model Tree tab by clicking that tab.

To insert (existing) classes into a component diagram:

1. Locate the **SavingsAccount** class  under the bankview package.
2. Drag it into the component diagram.
   The class is displayed with all its compartments.



3. Click both collapse icons to end up with the only the class name compartment.
4. Use the same method to insert the abstract class **Account**.

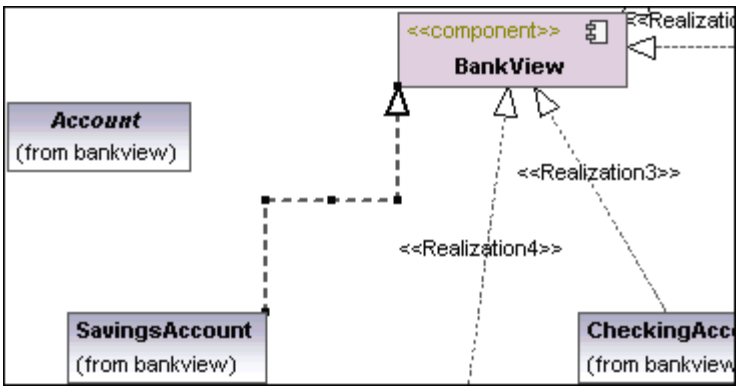


Please note:

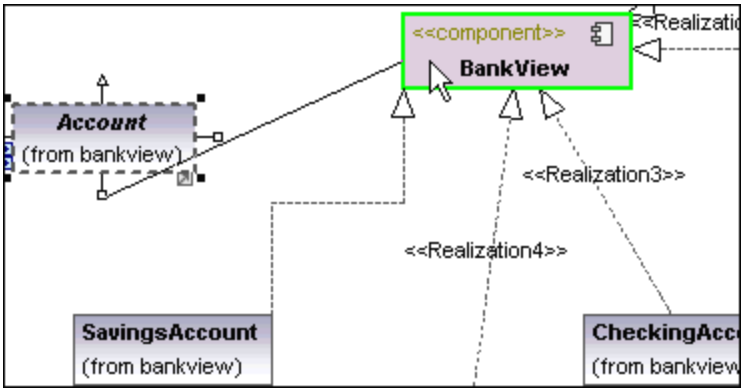
The package containing the inserted class, is displayed in the name compartment in the form "from bankview".

To create Realization dependencies between a class and component:

1. Click the Realization icon  in the icon bar.
2. Drag from **SavingsAccount**, and drop the arrow on the **BankView** component.



3. Click the **ComponentRealization** handle of the Account class (at the base), and drop it on the BankView component.

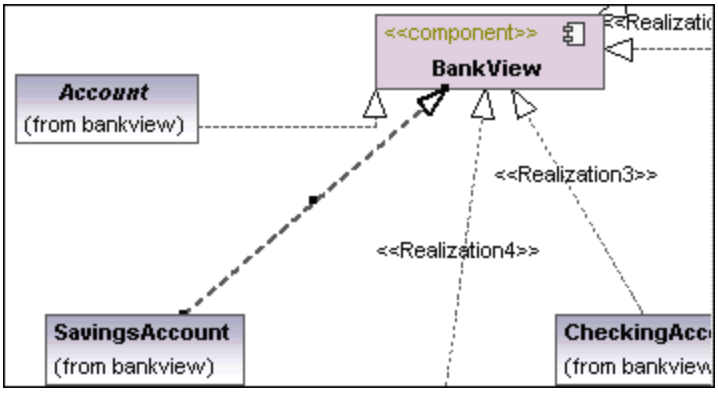


Both of these methods can be used to create realization dependencies. There is another method that allows you to create realization dependencies solely in the Model Tree, please see Round-trip engineering (code - model - code) for more information.

Changing (Realization) line characteristics:

Clicking a dependency or any other type of line in a UModel diagram, activates the line drawing icons in the Layout icon bar.

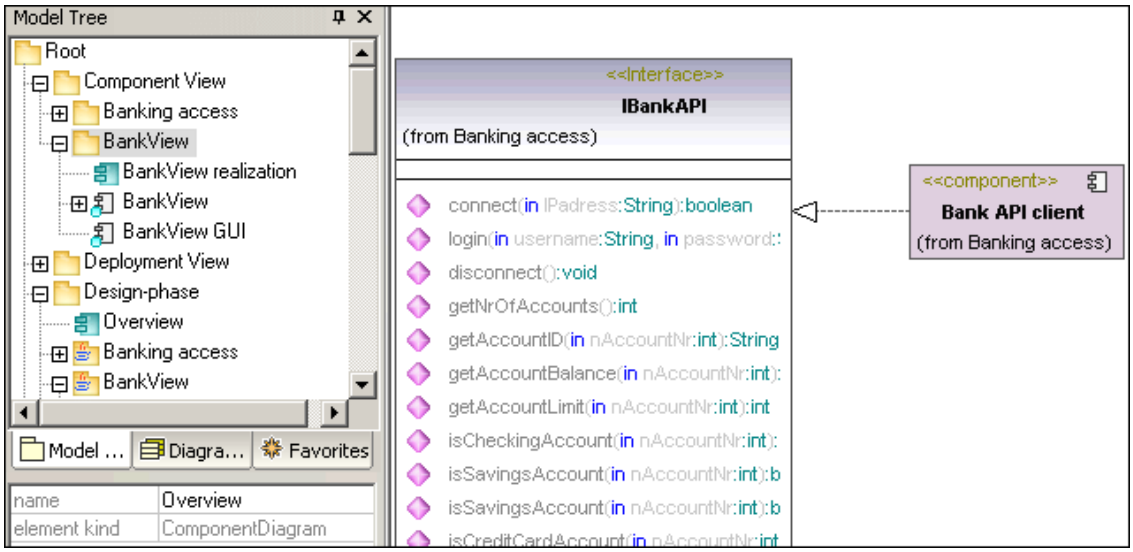
1. Click the realization line between **SavingsAccount** and BankView.
2. Click the line type icon **Direct line**  in the Layout toolbar.



The line properties are immediately altered. Lines have small icons along them called **waypoints**. Waypoints can be clicked and moved to alter line characteristics. Change the line properties to suit your needs.

Inserting components and creating usage dependencies:

1. Double click the **Overview** diagram icon directly under the **Design-phase** package in the Model Tree.
   The Overview component diagram is opened and displays the currently defined system dependencies between components and interfaces.

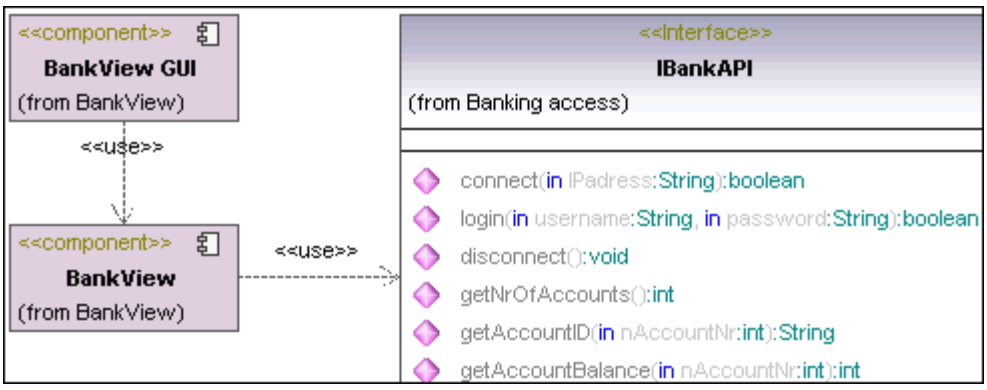


2. Click the **BankView GUI** component under the **Component View | BankView** package in the Model Tree, and drag it into the **Overview** diagram tab.
   The package containing the inserted component is displayed in the name compartment, "from BankView".
3. Use the same method to insert the **BankView** component under the same package.

The BankView component is the component produced by the "forward-engineering" process described in this tutorial.

To create a usage dependency between interfaces and components:

1. Click the Usage icon  in the icon bar.
2. Drag from the **BankView GUI** component to the **BankView** component.
3. Click the Usage icon again, and drag from the **BankView** component to the **IBankAPI** interface.



The usage dependency (<<use>>) connects a **client** element to a **supplier** element. In this case the IBankInterfaceAPI interface uses the services of components BankView and BankView GUI.

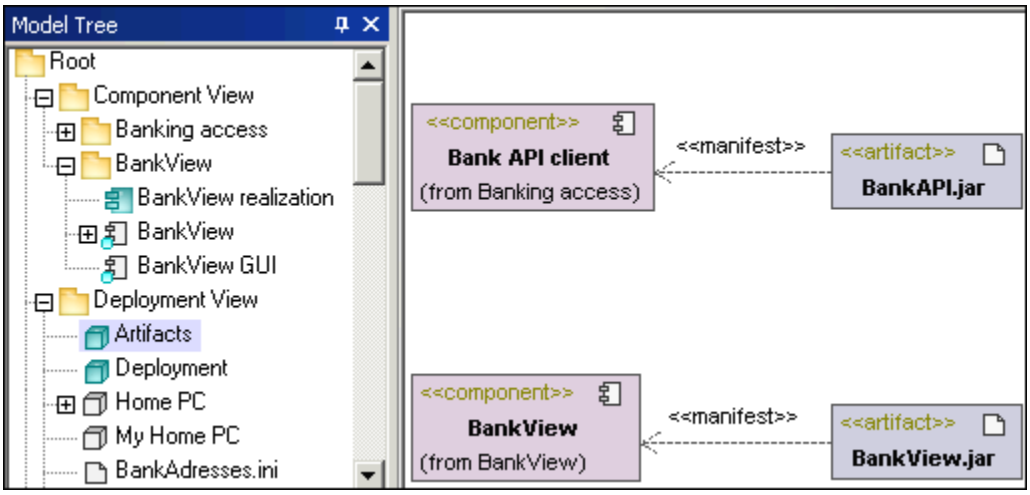
3.6 Deployment Diagrams

The aim of this tutorial section is to:

- Show the artifact manifestation of components
- Add a new node and dependency to a Deployment diagram
- Add artifacts to a node and create relationships between them

To open the Deployment (Artifacts) diagram:

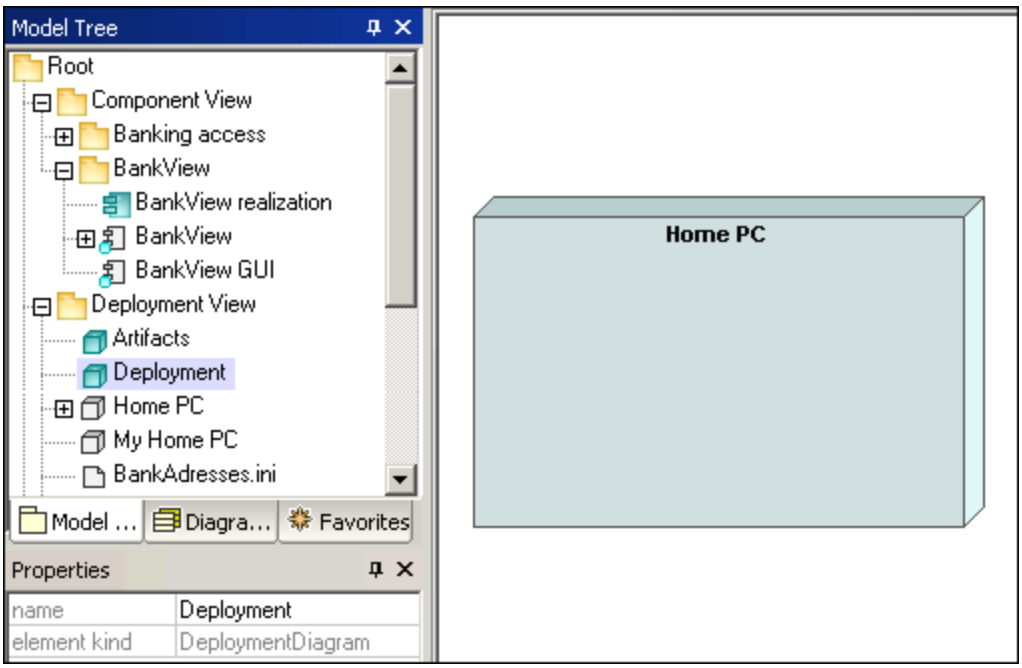
1. Click the Model Tree tab, expand the **Deployment View** diagram package, then double click the **Artifacts** icon.



This diagram shows the manifestation of the **Bank API client** and the **BankView** components, to their respective compiled Java **.jar** files.

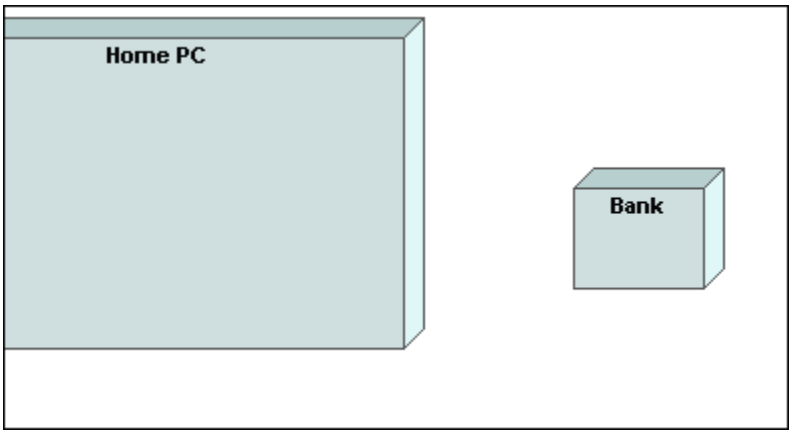
To open the Deployment diagram:

1. Double click the **Deployment** icon under the Deployment View package.
   The Deployment diagram is opened and displays the physical architecture of the system, which currently only comprises of the Home PC node.



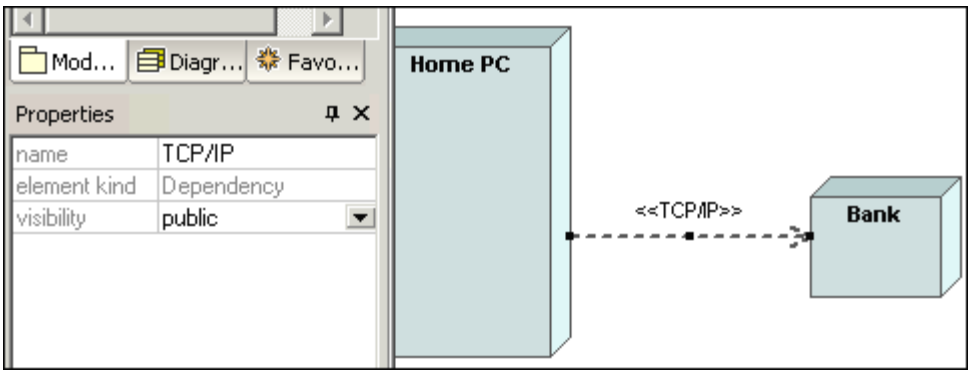
To add a Node to a Deployment diagram:

1. Click the Node icon  in the icon bar, and click right of the Home PC node to insert it.
2. Rename the node to Bank, and drag on one of its edges to enlarge it.



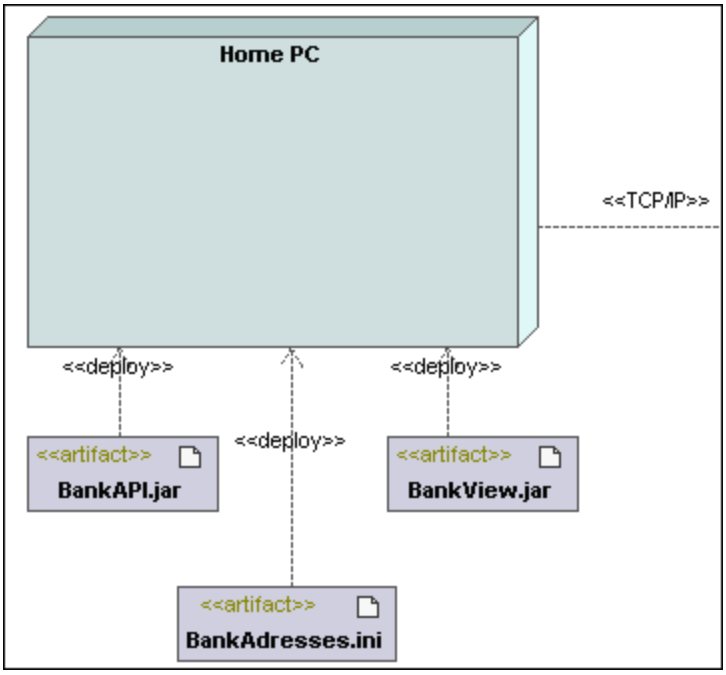
To create a dependency between two nodes:

1. Click the dependency icon  , then drag from the **Home PC** node to the **Bank** node. This creates a dependency between the two nodes.
2. Click into the **name** field of the Properties tab, change it to **TCP/IP**, and press Enter to confirm.
   The dependency name appears above the dependency line.

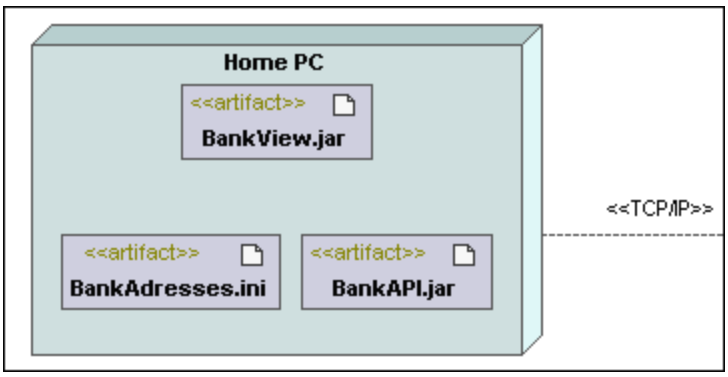


Adding artifacts to a node and creating dependencies between them:
Expand the Deployment View package, in the Model Tree, to see its contents:

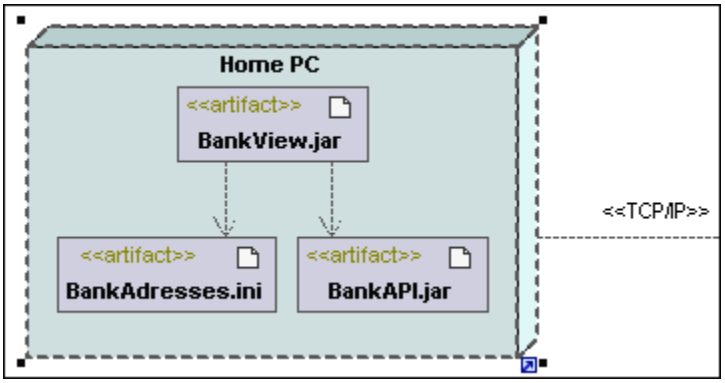
1. Click each of the **BankAddresses.ini**, **BankAPI.jar** and **BankView.jar** artifacts individually, and place them on the diagram background (Deployment dependencies are displayed for each artifact).



2. Click the **BankView.jar** artifact and drag it onto the **Home PC** node.
   The node is highlighted when the drop action will be successful.
3. Use the same method to drag the other artifacts onto the Home PC node.
   The artifacts are now part of the node and move with it when it is repositioned.



4. Click the Dependency icon  in the icon bar, and hold down the **CTRL** key.
5. Drag from the **BankView.jar** artifact to the **BankAddresses.ini** artifact; still holding down the CTRL key.
6. Drag from the **BankView.jar** artifact to the **BankAPI.jar** artifact.



Please note:
Dragging an artifact out of a node onto the diagram background, automatically creates a Deployment dependency.

To delete an artifact from a node and the project:

- Click the artifact you want to delete and press the **Del** keyboard key.

  The artifact and any dependencies are deleted from the **node** as well as the **project**.

To remove an artifact from a node and its diagram:

1. Use drag and drop to place the artifact onto the diagram background.
2. Hold down the **CTRL** key and press **Del**.

   The artifact and any dependencies are deleted from the current **diagram** and not from the project.

3.7 Round-trip engineering (model - code - model)

The aim of this tutorial section is to:

- Perform a project syntax check
- Generate project code
- Add a new method external code i.e. to the SavingsAccount class
- Synchronize the UModel model new code with the model

Packages and Code / model synchronization:
Code can be merged/synchronized at different levels:

- Project, Root package level (menu item)
- Package level (multiple package selection / generation is possible)
- Class level (multiple class selection / generation is possible)

The BankView realization diagram, depicts how the BankView component is realized by its six constituent classes. This is the component that is produced when the forward-engineering section of the tutorial is complete.

To be able to produce code:

- The component must be **realized** by one or more classes.
- The component must have a **physical location**, i.e. directory, assigned to it. The generated code is then placed in this directory.
- Components must be individually set to be **included** in the code engineering process.
- The Java, and C#, namespace root package must be defined.

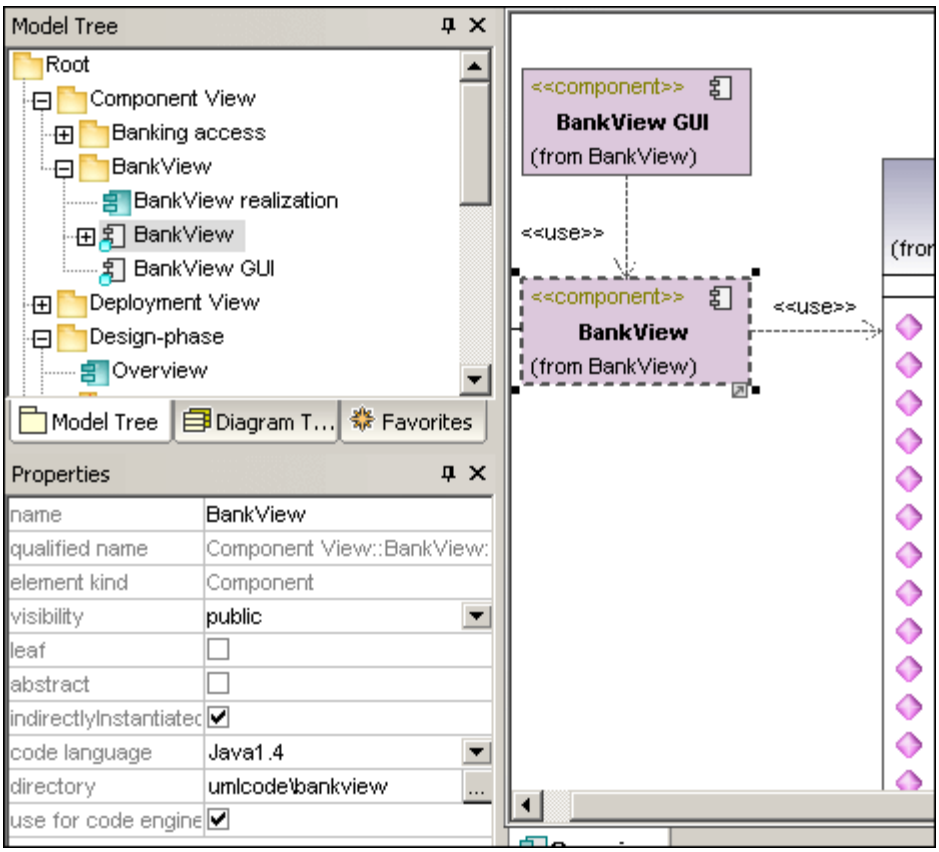
Please note:

The Java namespace root has been set on the **Design-phase | BankView | com** package in the Model Tree.

Java and C# code can be combined in one project and are automatically handled during the round-trip engineering process. The Bank_MultiLanguage.ump file in the ...\ **UModelExamples** folder is an example of a project for both types of code.

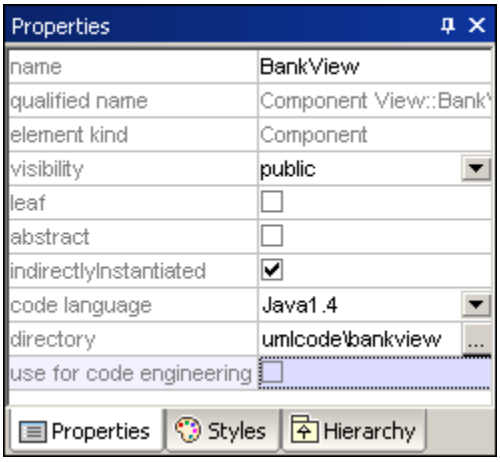
To define a code generation target directory:

1. Double click the **Overview** icon under the **Design-phase** package to switch into the component overview.
2. Click the **BankView** component, in the diagram, and note the current settings in the Properties tab.
3. Click the browse button  , to the right of the directory field.
4. Enter/select the target directory in the dialog box (the supplied example is defined as **InstallationDir\UModelExamples\Tutorial\umlcode\bankview**), or click the "Make New Folder" button to create a new folder.
   The path now appears in the directory field.



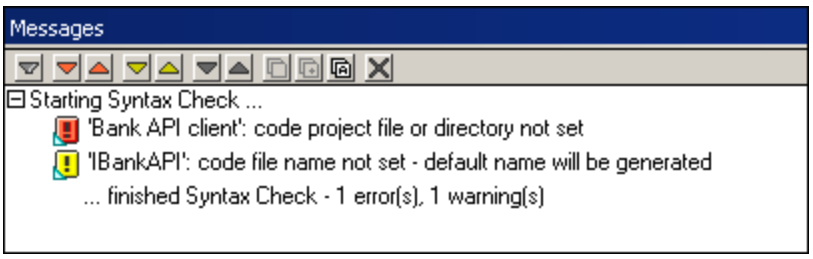
To include/exclude components from code generation:

1. Click the **BankView GUI** component.
2. **Uncheck** the "use for code engineering" check box (if not already unchecked).

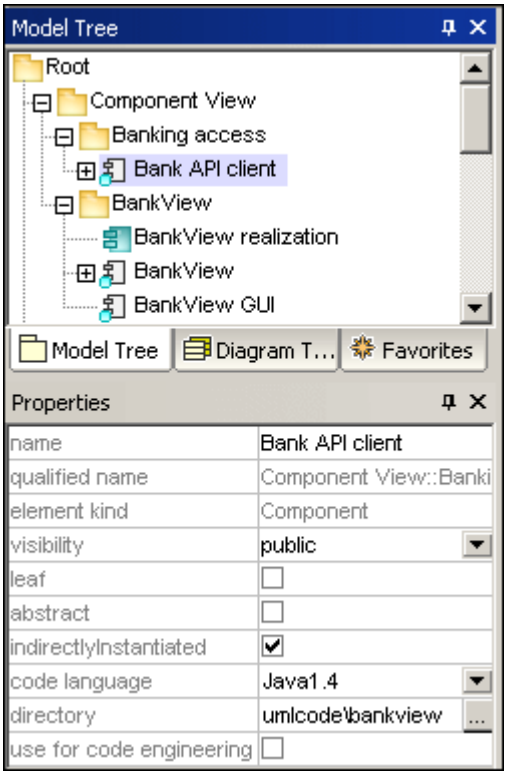


Checking project syntax prior to code generation:

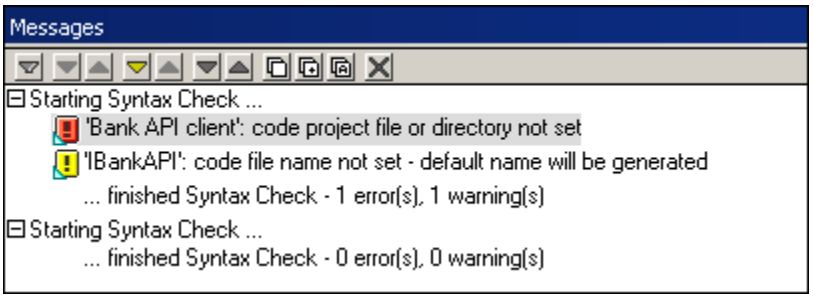
1. Select the menu option **Project | Check project syntax**.
2. A syntax check is performed, and messages appear in the Messages window, "Bank API-client: **code project file or directory not set**" - "IBankAPI: code file name not set".



3. Click the first message in the messages window.
4. The Bank API client package is highlighted in the Model Tree view, with its properties visible in the Properties tab.
5. **Uncheck** the "use for code engineering" check box for the Bank API client component.



6. Check the project syntax again using **Project | Check project syntax**.

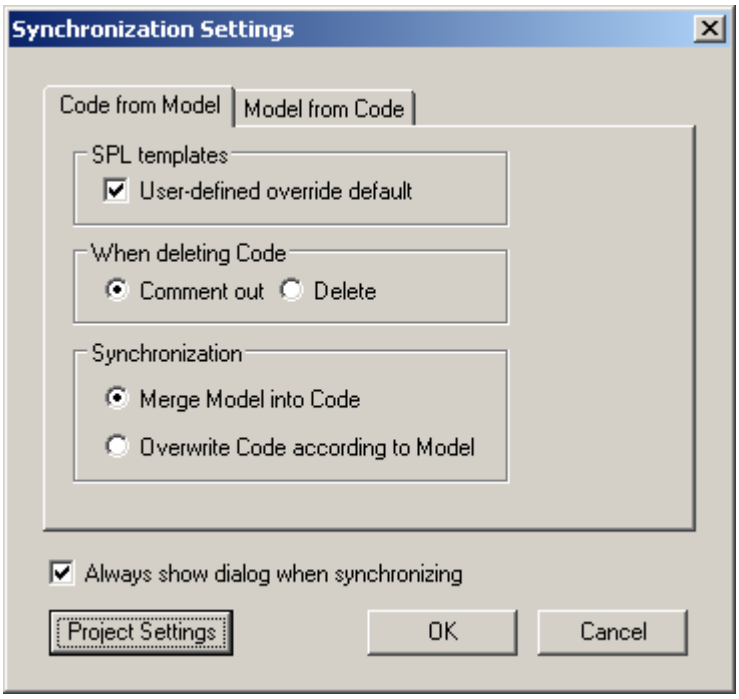


No errors are reported this time around. We can now generate program code for this project. Please see Check Project syntax for more information.

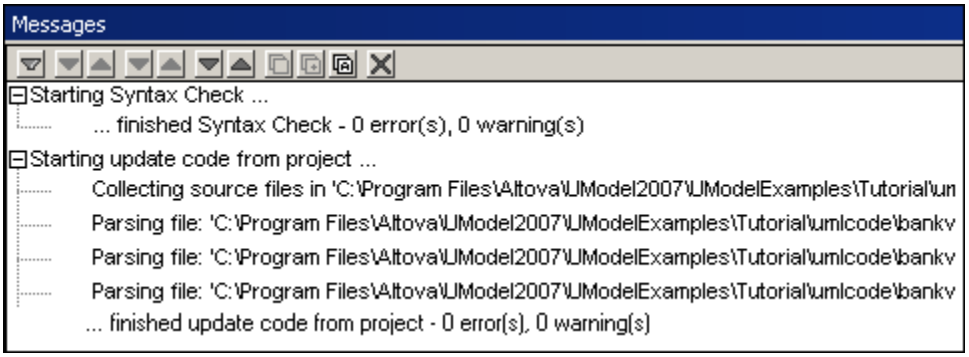
To generate project code:

1. Click the **BankView** package to select it.

2. Select the menu option **Project | Merge Program Code from UModel project**.
3. Select your synchronization options from the dialog box, and press OK to proceed (no changes needed for the tutorial; see "Merge Program Code from UModel project" for more information).



The message pane displays the outcome of the code generation process.



4. Navigate to the target directory.
   Six **.Java** files have been created for the project.

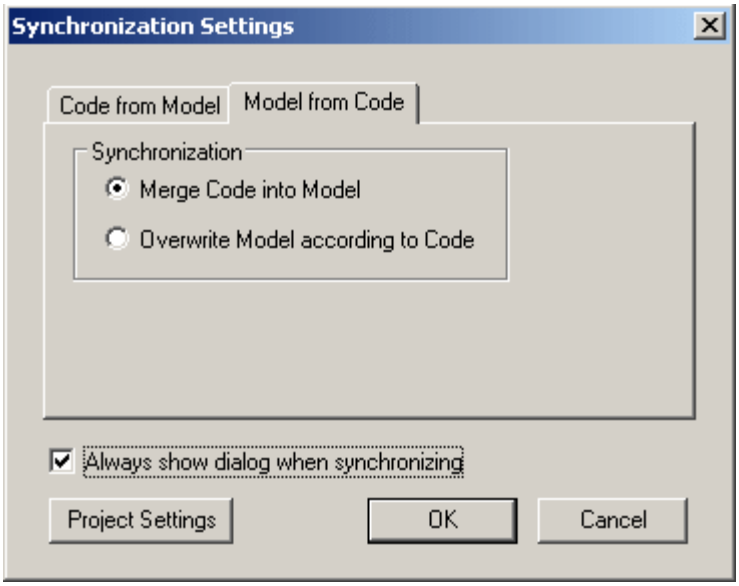
Synchronizing the UModel model having updated Java code externally:

1. Open the **SavingsAccount.java** file in the text editor of your choice, XMLSpy for example.
2. Add the new **method** to the generated code "**public float getInterestRate() {}**", and save the file.

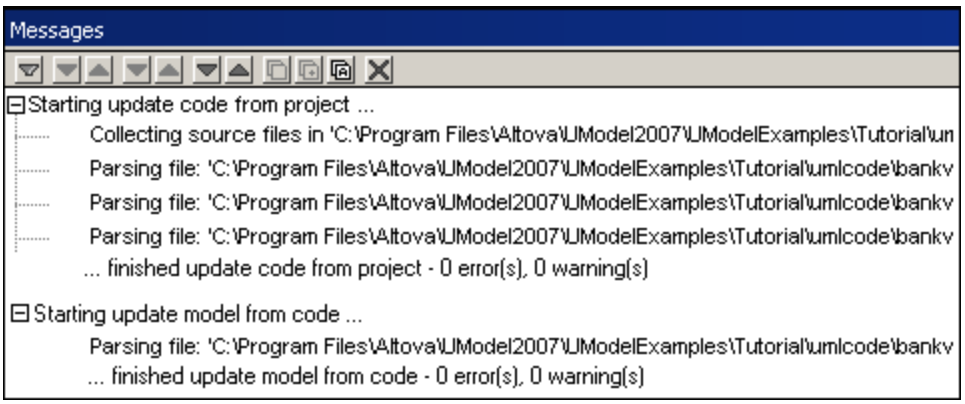
```
1
2     public class SavingsAccount extends Account
3     {
4
5         protected float interestRate;
6
7         public SavingsAccount()
8         {
9         }
10
11        public float getMinimumBalance()
12        {
13        }
14
15        public float getInterestRate()
16        {
17        }
18
19        public boolean collectAccountInfo(IBankAPI bankAPI)
20        {
21        }
22    }
23
```

3. Switch to UModel and right click the **SavingsAccount** class  under the BankView package.
4. Select the option **Code Engineering | Merge UModel Class from Program Code**.

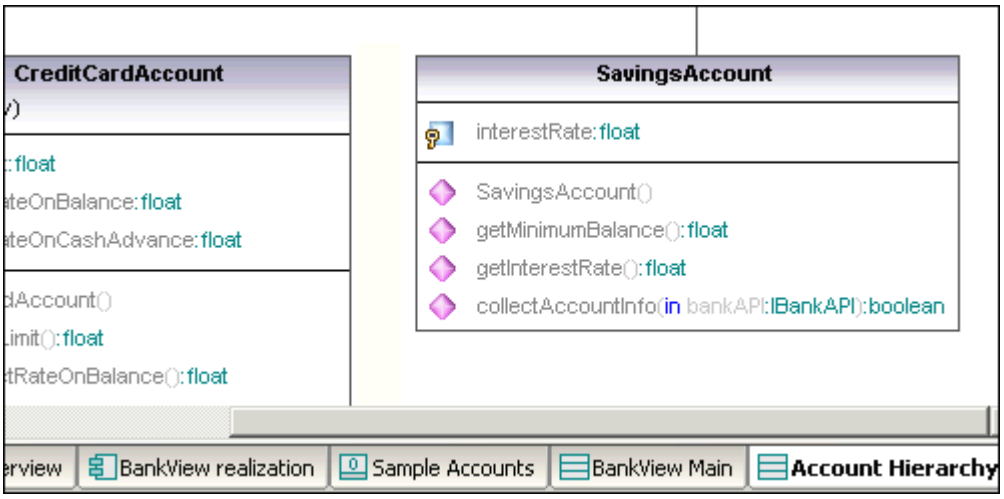
   This opens the Synchronization Settings dialog box with the "Model from Code" tab being active. No changes are needed for the tutorial; see "Merge UModel project from code" for more information)



5. Click OK to merge the model from the code.



6. Click the **Account Hierarchy** tab to see the outcome of the merge process.



The new method added to the code, (getInterestRate...) generates a new **operation** in the SavingsAccount **class** of UModel.

3.8 Round-trip engineering (code - model - code)

The aim of this tutorial section is to:

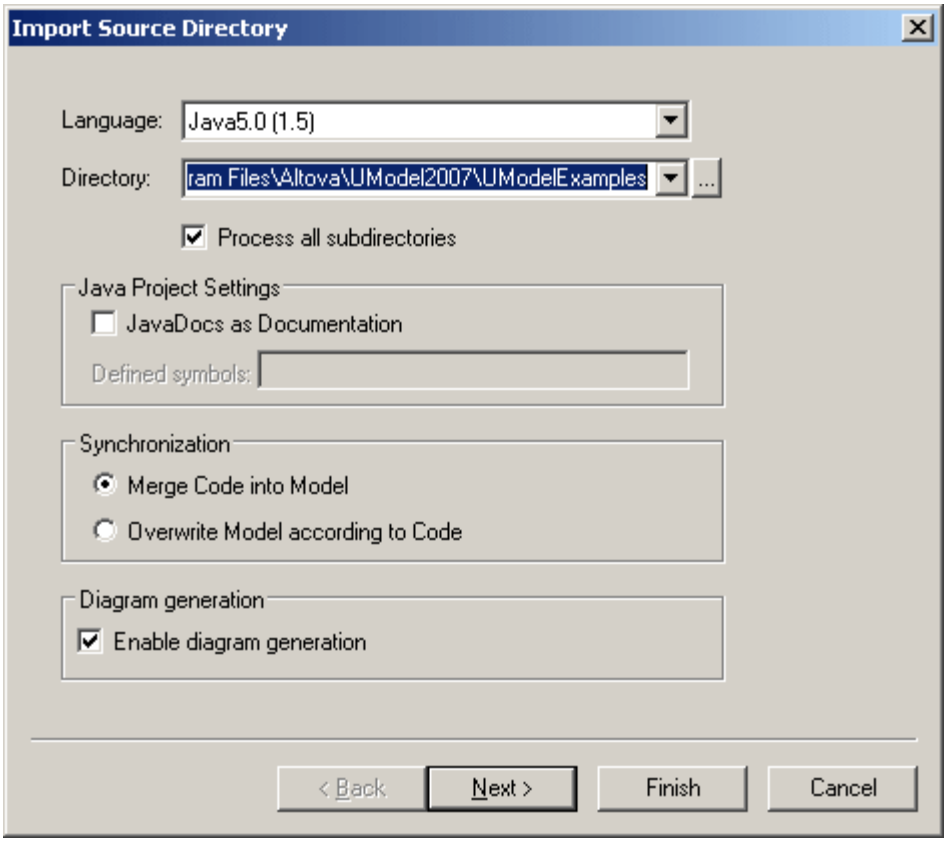
- Import a directory containing Java code generated by XMLSpy
- Add a new class to the project in UModel
- Merge to the program code from a UModel package

The files used in this example are available as the **OrgChart.zip** file under **...\UModelExamples** folder of your installation. Please unzip the OrgChart.zip file into the ...\UModelExamples folder before you start this section.

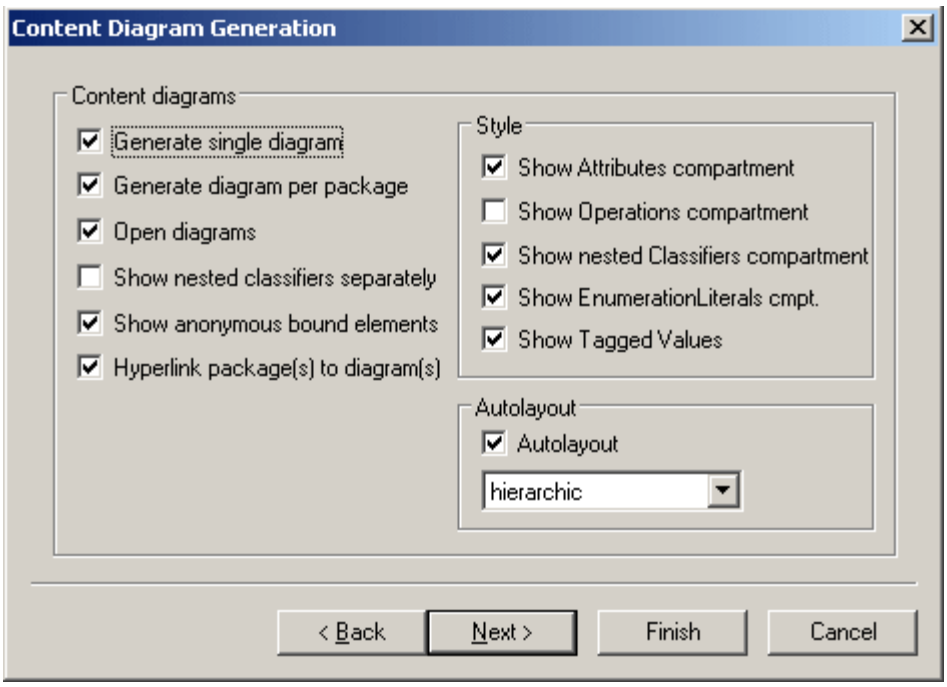
This creates the **OrgChart** directory which will then be used to import the existing code.

To Reverse engineer/import existing code from a directory:

1. Select **File | New** to create a new project.
2. Select **Project | Import source directory**.
3. Select the C#, or Java version (1.4, or 5.0.) that the source code conforms to.
4. Click the Browse button [...] and select the **OrgChart** directory supplied in the **...\UModelExamples** folder.

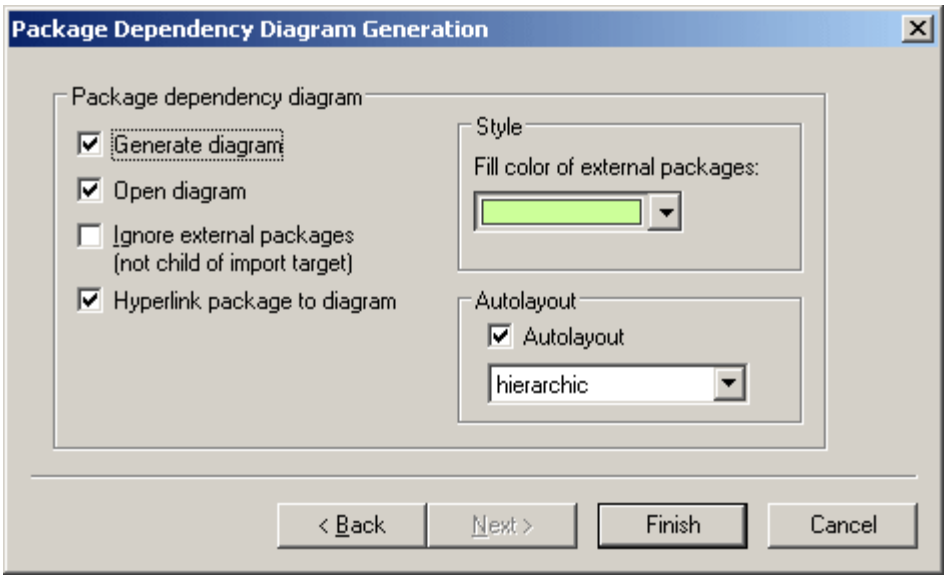


5. Making sure that the "**Enable diagram generation**" check box is active, select any specific import settings you need, and click Next.



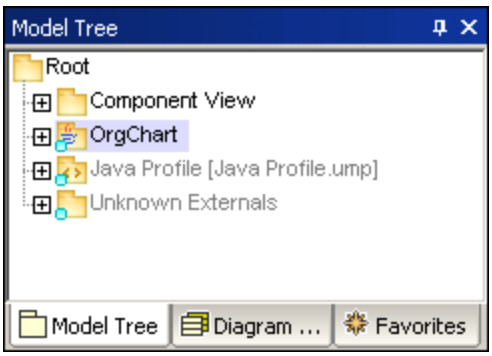
Note that UModel can generate a single overview diagram and/or a diagram for each package. The settings show above are the default settings.

6. Click Next to continue.

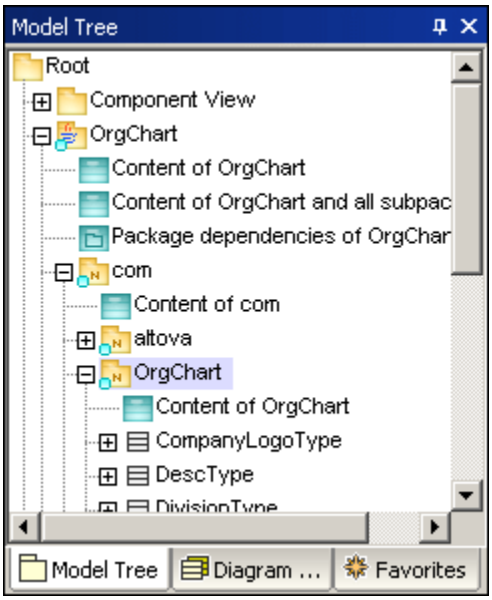


This dialog box allows you to define the package dependency generation settings.

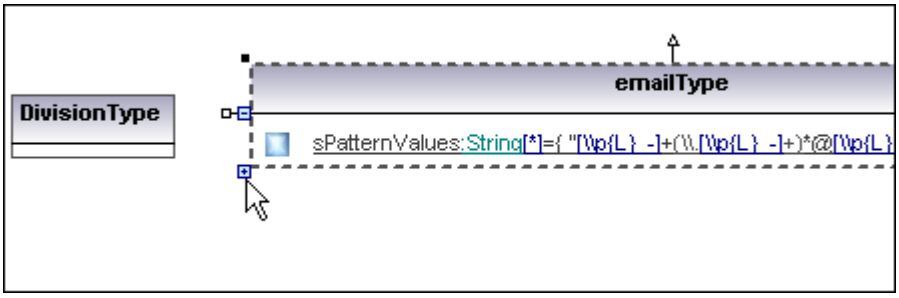
7. Click Finish to use the default settings.
   The data is parsed while being input, and a new package called "**OrgChart**" is created.



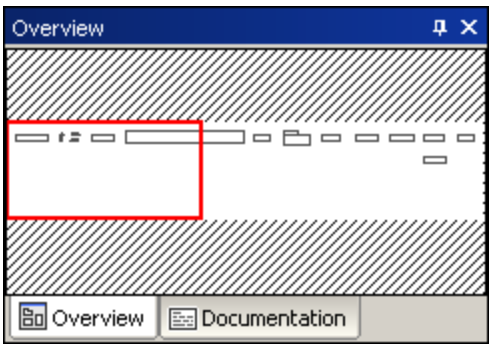
8. Expand the new package and keep expanding the sub packages until you get to the **OrgChart** package (**com | OrgChart**).



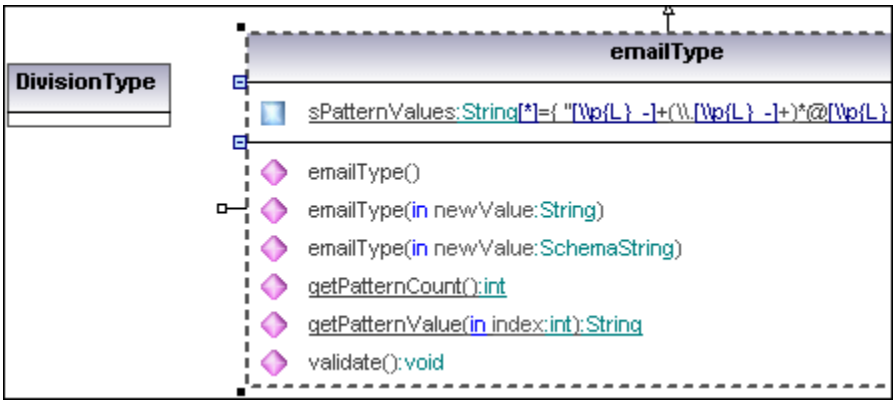
9. Double click the **"Content of OrgChart"** diagram icon .
The collapsed classes that make up OrgChart are displayed in the main tab.



The current window/view is shown by the red box in the Overview window, which occupies an empty area of the diagram.



10. Click the expand icon of the operation compartment, e.g. emailType, to see the constituent operations.



Please note:
You could also select the **Project | Import source project** option and select the Borland JBuilder **OrgChart.jpx** project file to import the project created by XMLSpy.

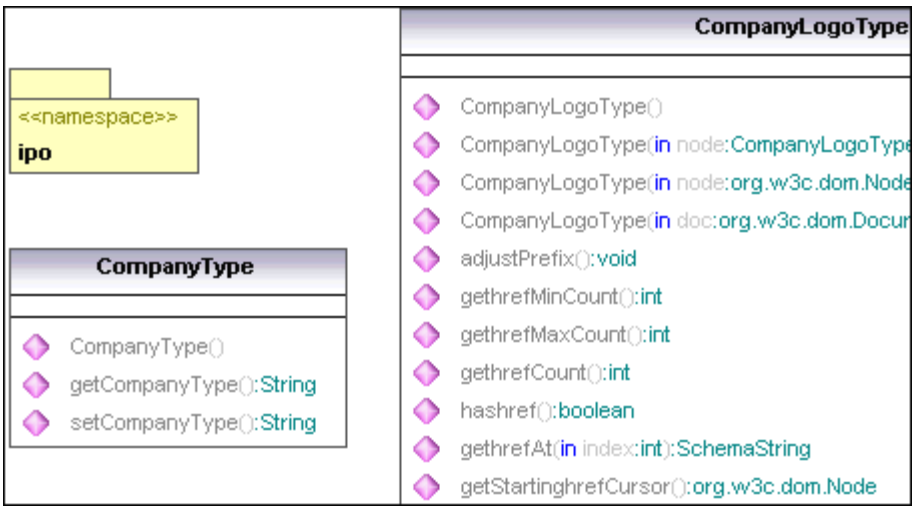
Round-trip engineering and relationships between modeling elements:

When updating model from code, associations between modeling elements are automatically displayed, if the option **Editing | Automatically create Associations** has been activated in the **Tools | Options** 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.

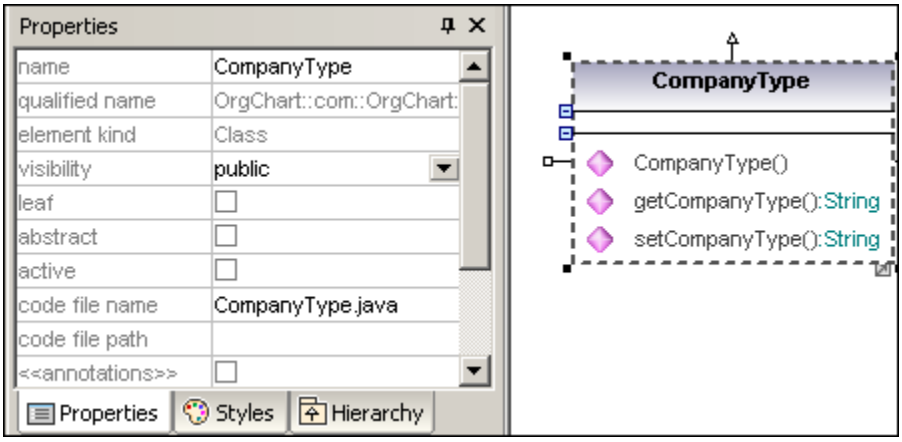
Adding a new class to the OrgChart diagram:

1. Click the Class icon  in the icon bar and click to insert a new class.
2. Add a new Class called **CompanyType**.
3. Add new operations to the class using the F8 shortcut key:
   e.g. **CompanyType()**, **getCompanyType():String**, **setCompanyType():String.**

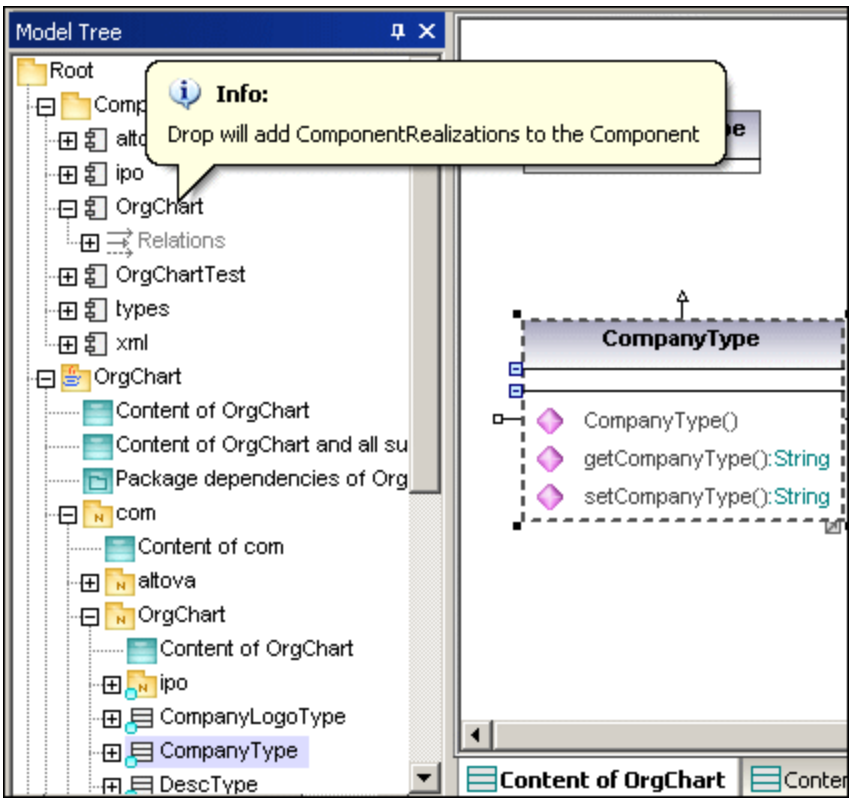


Making the new class available for code generation:
While the CompanyType class is active,

1. Click into the "code file name" field and enter the Java file name of the new class **CompanyType.java**.



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

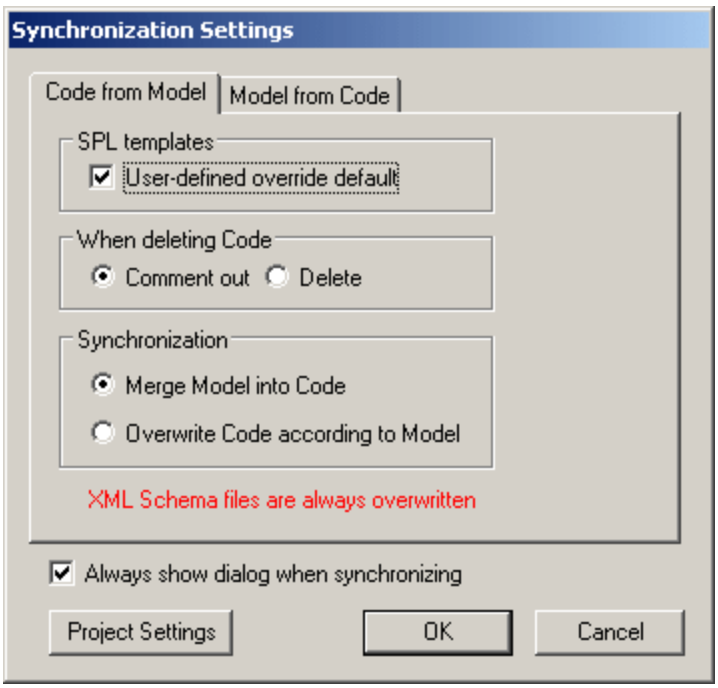


Please note:
This method creates a Realization between a class and a component, without having to use component or deployment diagrams.

3. Expand the **Relations** item below the Orgchart component, to see the newly created realization.

Merging program code from a package:

1. Right click the **OrgChart** package, select **Code Engineering | Merge Program code from UModel Package**, and press Enter to confirm.



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

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

Please note:

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.

That's it!

You have learned how to create a modeling project using the forward engineering process, and also completed a full round-trip code engineering cycle with UModel. The rest of this document describes how best to achieve modeling results with UModel.

Chapter 4

UModel User Interface

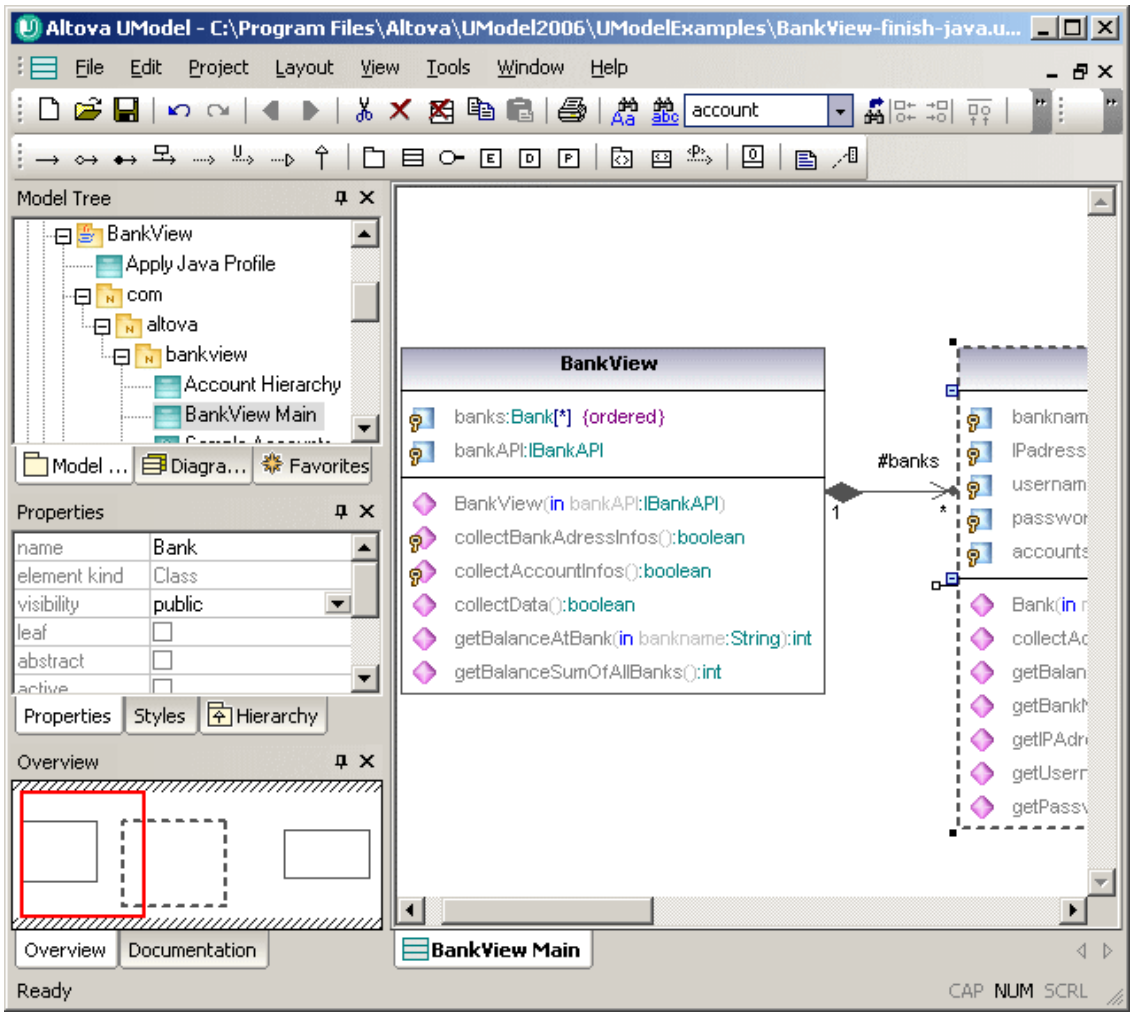
4 UModel User Interface

UModel consists of series of panes on the left and a larger diagram tab at right. The panes at left allow you to view and navigate your UModel project from differing viewpoints, and edit data directly.

The panes are Model Tree, Properties, and Overview. The working/viewing area at right is the UModel Diagram tab which currently shows the Class Diagram of the BankView Main package.

Please note:

All panes, as well as diagram tabs, can be searched using the **Find** combo box in the Main toolbar, which contains the text "**account**" in the screenshot below, or by pressing CTRL+F.

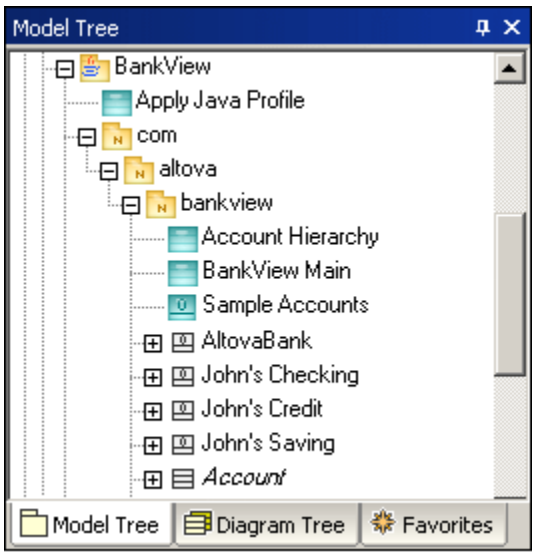


4.1 Model Tree pane

Model Tree tab
The Model Tree tab allows you to manipulate model items directly in the Model Tree, as well as navigate/view specific items in the Design tab. Right clicking an item opens the context menu, from which specific commands can be selected. The contents of the context menu depend on the item that you select.

Model elements in the Model Tree pane can be directly manipulated:

- Added / inserted
- Copied or moved
- Deleted
- Renamed
- Sorted according to several criteria
- Constrained



In the Model Tree tab, each folder symbol is a UML package!

Adding a new package (or any other modeling element):

1. Right click the **folder** that you want the new package/element to appear under.
2. Select **New | Package** (or respective model Element).

Copying or moving model elements:

1. Use the standard windows Cut, Copy or Paste commands or,
2. Drag model elements to different packages. Dragging an elements moves it. Holding down CTRL a and dragging an element creates a copy.

   When dragging elements a message might appear stating that select "No sort" needs to be activated to allow you to complete the action. Please see "Cut, copy and paste in UModel Diagrams" for more information.

Sorting elements in the Model Tree (activating no sort):

1. Right click the empty background of the Model Tree tab.
2. Select **Sort | No sort**.
   Elements can now be positioned anywhere in the Model Tree.

Please note:
The Sort popup menu also allows you to individually define the sort properties of

Properties and **Operations**.

Renaming an element:

1. Double click the element **name** and edit it.
   The Root and Component View packages are the only two elements that cannot be renamed.

Deleting an element:

1. Click the element you want to delete (use CTRL+click to mark multiple elements).
2. Press the **Del.** keyboard key.

   The modeling element is deleted from the Model Tree. This means that it is also deleted from the Diagram tab, if present there, as well as from the project. Elements can be deleted from a diagram without deleting them from the project, using CTRL+Del. Please see deleting elements.

To open a diagram in the Diagram tab:

1. Double click the diagram icon of the diagram you want to view in the diagram tab.

Modeling element icon representation in the Model Tree

Package types:

UML Package
Java namespace root package
C# namespace root package
XML Schema root package
Java, C#, code package (package declarations are created when code is generated)

Diagram types:

Activity diagram
Class diagram
Communication diagram
Component diagram
Composite Structure diagram
Deployment diagram
Interaction Overview diagram
Object diagram
Package diagram
Sequence diagram
State Machine diagram
Timing diagram
Use Case diagram
XML Schema diagram

Element types:

An element that is currently visible in the active diagram is displayed with a blue dot at its base. In this case a class element.

Class Instance/Object
Class instance slot

Class
Property
Operation
Parameter

Actor (visible in active use case diagram)
Use Case
Component

Node
Artifact
Interface

Relations (/package)
Constraints

Opening / expanding packages in the Model Tree view:
There are two methods available to open packages in the tree view; one opens all packages and sub packages, the other opens the current package.

Click the package you want to open and:

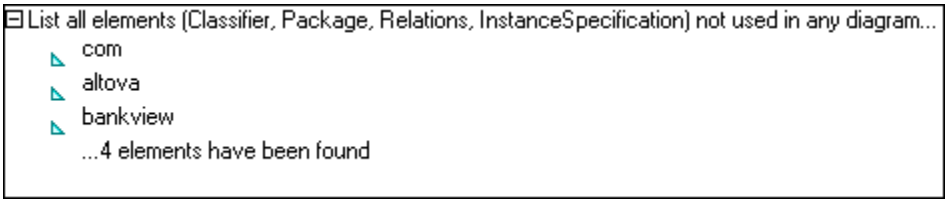
- Press the * key to open the current package and all sub packages
- Press the + key to open the current package.

To **collapse** the packages, press the - keyboard key.
Note that you can use the standard keyboard keys, or the numeric keypad keys to achieve this.

To find modeling elements in Diagram tab(s):
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. There are two methods to find elements:

1. Right click the element you want to see in the Model Tree tab, and select:
   - Show element in active diagram - to find it in the same type of diagram tab
   - Show element in all diagrams - if currently active diagram differs from selected model element.

To generate a list of elements not used in any diagram:
1. Right click the package you would like to inspect.
2. Select the menu option "List elements not used in any diagram.
   A list of unused element appears in the Messages pane. The list in parenthesis, displays the specific elements which have been selected to appear in the unused list, please see the **View** tab in Reference section under, **Tools | Options** for more information.



To **locate** the missing elements in the Model Tree:
- Click the element name in the Messages pane.

Please note:
The unused elements are displayed for the **current** package and its sub packages.

Packages in the Model Tree tab:
Only the Root and Component packages are visible on startup, i.e. when no project is currently loaded.

- Packages can be **created**, or deleted at any position in the Model Tree
- Packages are the containers for all other UML modeling elements, use case diagrams etc.

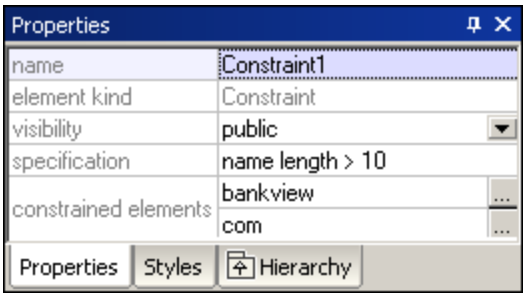
- Packages/contents can be **moved**/copied to other packages in the Model Tree (as well as into valid model diagrams in the diagram tab)
- Packages and their contents can be **sorted** according to several criteria
- Packages can be placed within other packages
- Packages can be used as the **source**, or **target** elements, when generating or synchronizing code

Generating/merging code:
UModel allows you to generate, or merge program code directly from the Model Tree, please see: Synchronizing Model and source code for more information.

Constraining UML elements:
Constraints can be defined for most model elements in UModel. Please note that they are not checked by the syntax checker, as constraints are not part of the Java code generation process.

To constrain an element (Model Tree):

1. Right click the element you want to constrain, and select **New | Constraint**.
2. Enter the name of constraint and press Enter.
3. Click in the "specification" field of the Properties tab, and enter the constraint e.g. name length > 10.



To constrain an element in UML diagrams:

1. Double click the specific element to be able to edit it.
2. Add the constraint between curly braces e.g. interestRate:float **#{interestRate >=0}**.

To assign constraints to multiple modeling elements:

1. Right click the "constrained elements" field in the Properties tab.
2. Select "Add element to constrained elements".
   This opens the "Select Elements to be Constrained" dialog box.
3. Select the specific element you want to assign the current constraint to.

   The "constrained element" field contains the names of the modeling elements it has been assigned to. The image above, shows that Constraint1 has been assigned to the **bankview** and **com** packages.

4.1.1 Diagram Tree tab

Diagram Tree tab
This tab displays the currently available UModel diagrams in two ways:

- Grouped by diagram type, sorted alphabetically
- As an alphabetical list of all project diagrams

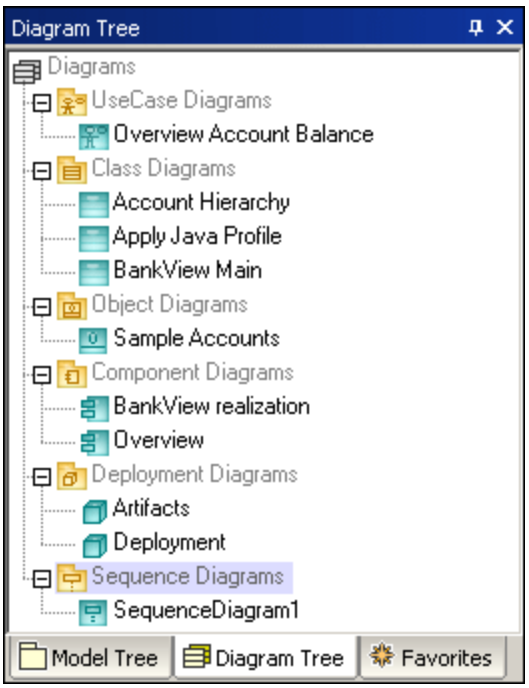
Please note:
Diagrams can be added to, or deleted from, the Diagram Tree tab by right clicking and selecting the requisite command.

To open a diagram in the Diagram tab:

- Double click the diagram you want to view in the diagram tab.

To view all Diagrams within their respective model groups:

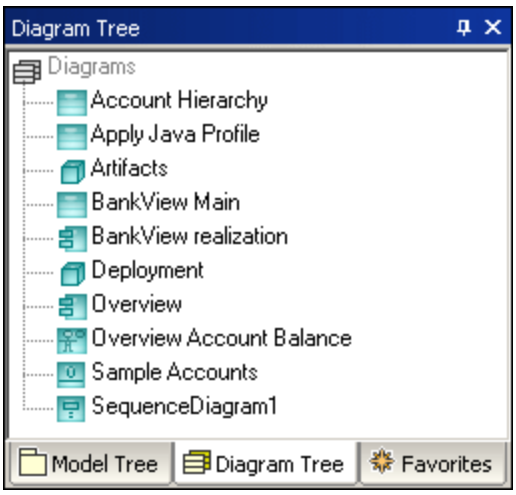
- Right click in the pane, and activate the "Group diagram by diagram type" option.



Diagrams are grouped alphabetically within their group.

To view all Diagram types in list form (alphabetically):

- Right click in the pane, and deactivate the "Group diagram by diagram type" option.



All Diagrams are shown in an alphabetically sorted list.

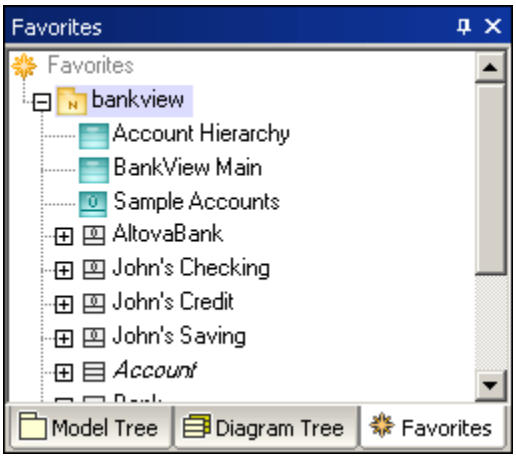
4.1.2 Favorites tab

Favorites tab
Use this tab as a user-defined repository, or library, for all types of **named** UML elements i.e. classes, objects, associations etc. but not ProfileApplication or Generalization dependencies. This allows you to create your personal pick-list of modeling elements for quick access.

The contents of the Favorites tab are automatically saved with each project file. Select the menu option **Tools | Options, File** tab and click the "Load and save with project file" check box to change this setting.

To add an existing modeling element to the Favorites tab:

1. Right click an element in the Model Tree tab, or in the diagram working area.
2. Select the menu item "Add to Favorites".
3. Click the Favorites tab to see the element.



The element appears in the Favorites tab is a view of an existing element, i.e. it is not a copy or clone!

To add a NEW element to the Favorites tab:

1. Right click a previously added package, to which you want to add the element.
2. Select **New | "modeling element"** from the context menu, where "modeling element" is a class, component, or any other modeling element available in the context menu. New elements are added to the same element/package in the project, and are therefore also visible in the Model Tree tab.

To REMOVE an element from the Favorites tab:

1. Right click the same element/package that you added to Favorites.
2. Select **Remove from Favorites**.

Please note:
You can add and remove elements added to the Favorites tab, from the Favorites tab, as well as the Model Tree tab.

Deleting elements from the Favorites tab:

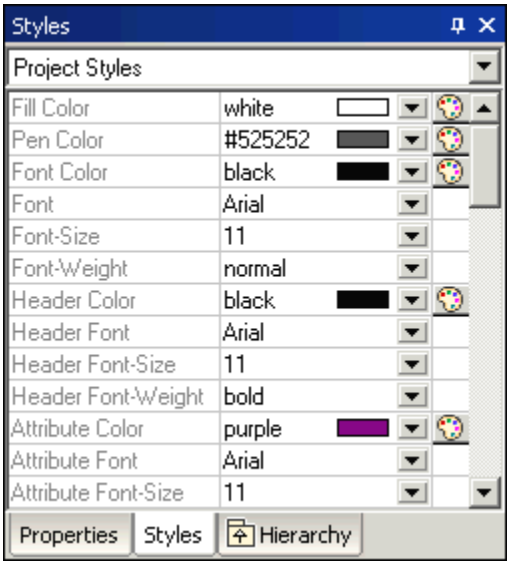
1. Right click the element you want to delete, and press the Del key.
   A message box appears, informing you that the element will be deleted from the project.
2. Click OK if you want to delete it from the project.
3. Click Cancel to retain it, and use the **Remove method** described above, to delete it from the Favorites tab.

4.2 Properties pane

Properties tab
The Properties tab displays the UML properties of the currently active element.

- Clicking **any** model element in any of the supplied views, or tabs, displays its properties.
- Once visible, model properties can be changed, or completed, by entering data, or selecting various options in the tab.
- Selected properties can also be located in the diagram tabs by selecting Show in Active Diagram from the context menu.

Styles tab
The Styles tab is used to view, or change attributes of diagrams, or elements that are displayed in the diagram view.

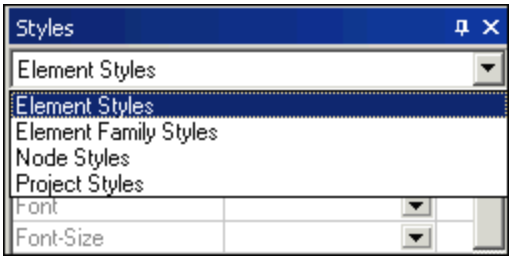


These style attributes fall into two general groups:

- Formatting settings; i.e. font size, weight, color etc.

- Display settings/options; show background color, grid, visibility settings etc.

The Styles tab is subdivided into several different categories/sections which can be selected by clicking the "Styles" combo box. The combo box contents depends on the currently selected model element.



Clicking an element in a diagram tab automatically selects the Element Style context, while clicking and element in the Model Tree tab selects the Project Style context.

Style **precedence** is bottom-up, i.e. changes made at the more specific level override the more general settings. E.g changes (to an object) made at the Element Style level override the current Element Family and Project Styles settings. However, selecting a different object and changing the Element Family Styles setting, updates all other objects except for the one just changed at the Element Style level.

Please note:
Style changes made to model elements can all be undone!

Element Styles:
Applies to the currently selected element in the currently active diagram. Multiple selections are possible.

Element Family Styles:
Applies to all elements of the same type i.e. of the selected Element Family. E.g. you want to have all Component elements colored in aqua. All components in the Component and Deployment diagrams are now in aqua.

Node / Line Styles:
"Node" applies to all rectangular objects.
"Lines" applies to all connectors: association, dependency, realization lines etc. for the whole project.

Project Styles:
Project Styles apply to the current UModel Project in its entirety (e.g. you want to change the default Arial font to Times New Roman for all text in all diagrams of the project).

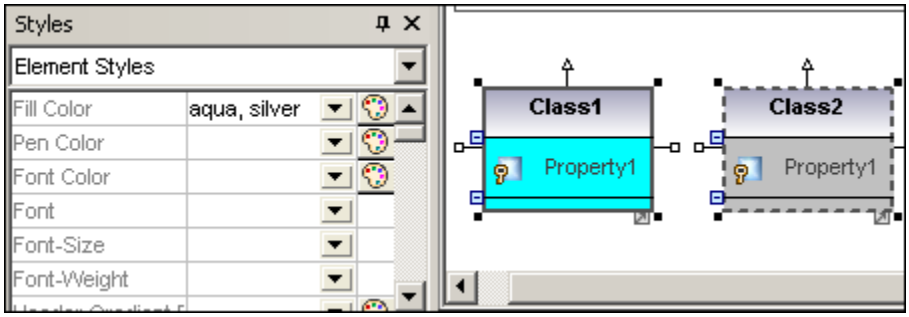
Diagram Styles:
These styles only becomes available when you click/select a diagram background. Changing settings here, only affects the single UML diagram for which the settings are defined in the project.

To change settings for all diagrams of a project:

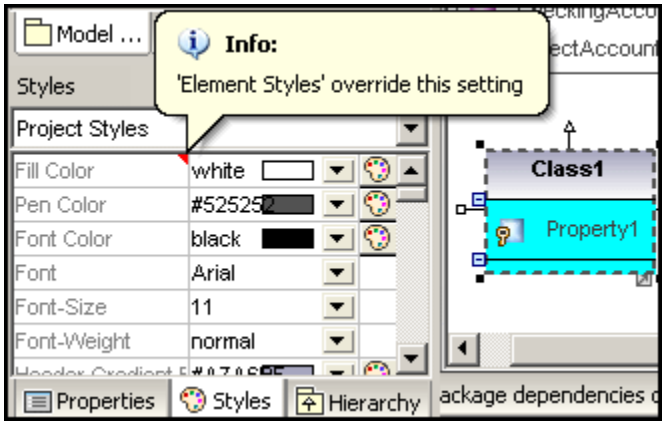
1. Click in the respective diagram,
2. Select the **Project Styles** entry in the combo box, and scroll to the bottom of the tab.
3. Select one of the **Diag.yyy** options e.g. Diag. Background color.
   This then changes the background color of all diagrams in the current project.

Styles display when multiple elements are selected:

If multiple elements are selected in the diagram pane, then all different style values are displayed in the respective field. In the screenshot below, Class1 and Class2 have been selected.
The fill Color field displays the values for each of the elements, i.e. aqua and silver.



Displaying cascading styles:
If a style is overridden at a more specific level, a small red triangle appears in the respective field in the styles tab.
Placing the mouse pointer over the field displays a popup which indicates the style precedence.

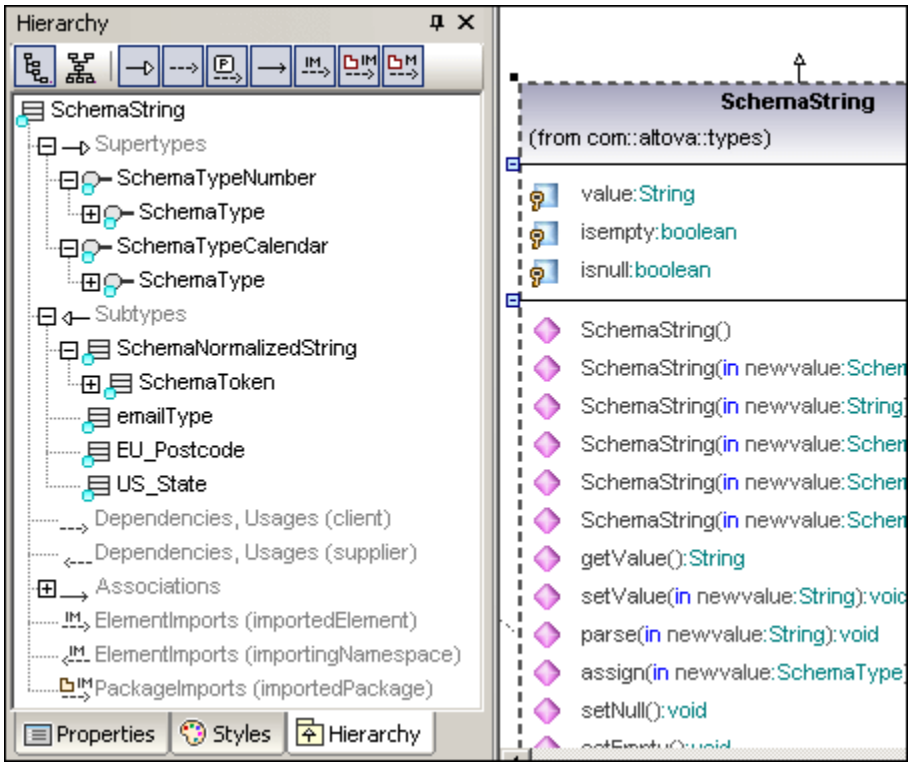


E.g.
The Enumeration, Package and Profile elements all have default fill color settings defined in the Element Family Styles settings. To change the fill colors at the project level, clear the value in the Element Family Styles i.e. select the empty entry in the drop-down list box, select Project styles from the Styles combo box, and change the fill color there.

4.3 Hierarchy tab

Hierarchy tab
The hierarchy tab 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 tab.

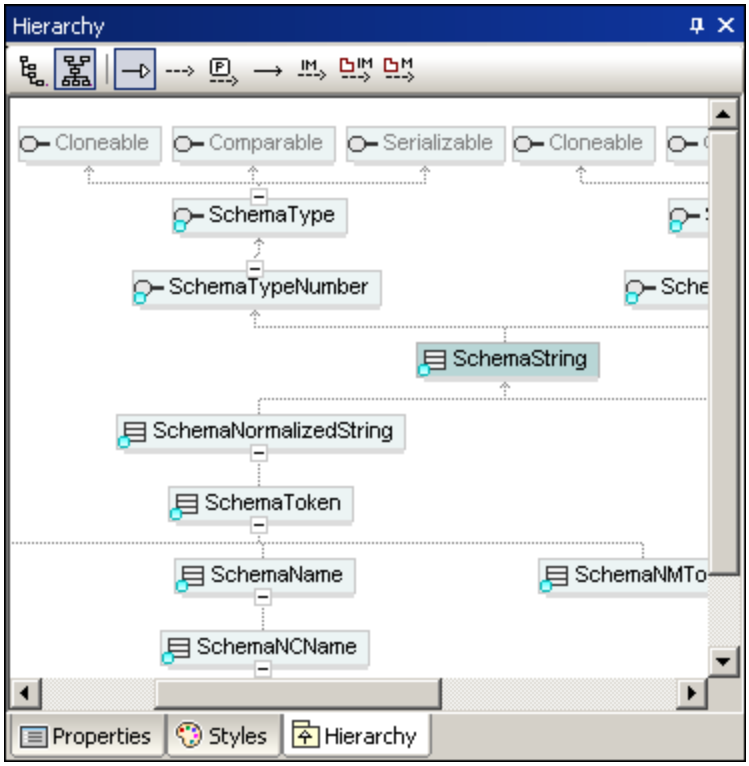
 Show Tree view



This view shows **multiple** relations of the currently selected element e.g. SchemaString. Clicking the various icons in the icon bar, allows you to show all types of relations, or narrow them down by clicking/activating the various icons. In the screenshot above, all icons are active and thus all relations are shown in a tree view.

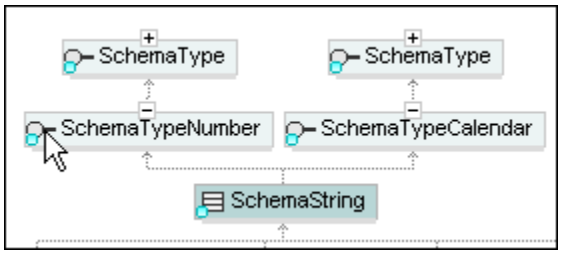
Double clicking one of the element **icons**, in the tab, displays the relations of that element.

 Show graph view

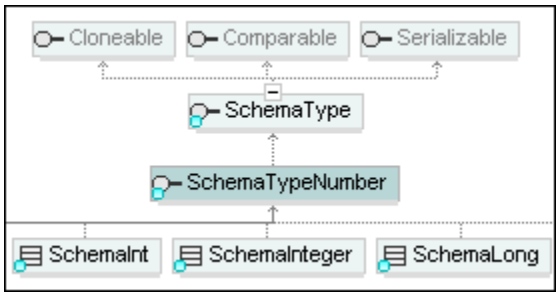


This view shows a **single** set of relations in an hierarchical overview. Only one of the relation icons can be active at any one time. The Show Generalizations icon is currently active.

Double clicking one of the element **icons** in the tab, e.g. SchemaTypeNumber, displays the relations of that element.

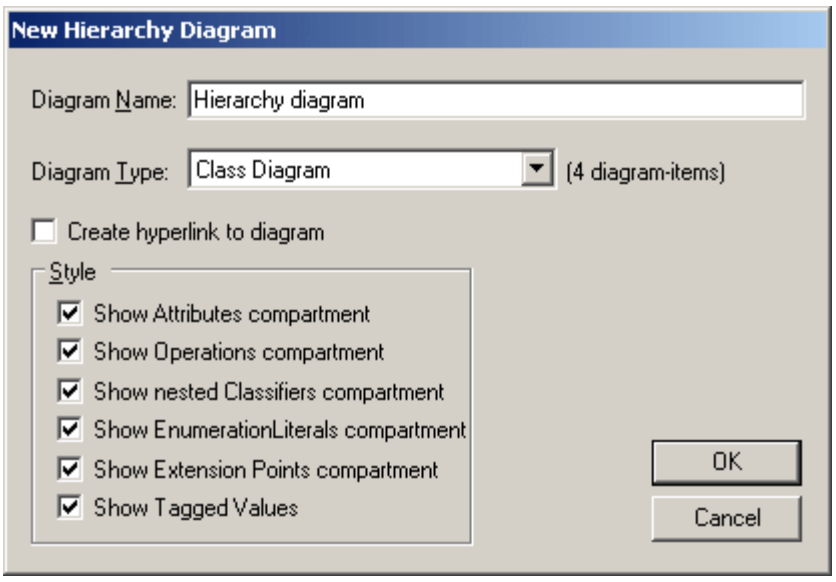


The currently selected element is now SchemaTypeNumber.



Creating a new diagram from the contents of the window:
The current contents of the graph view pane can be displayed in a new diagram.

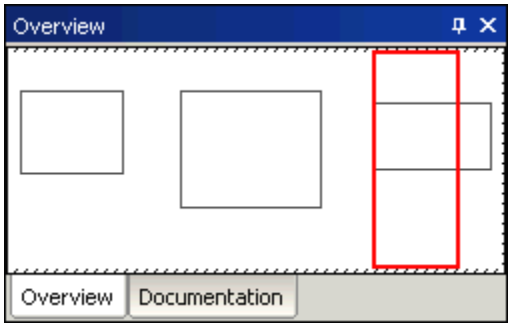
1. Right click in the graph view pane and select **Create diagram as this graph**.



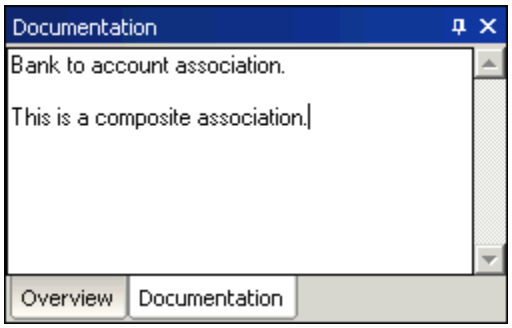
2. Edit the diagram name if necessary, select the style options and click OK.
   A new diagram is created.

4.4 Overview pane

Overview tab
The Overview tab displays an outline view of the currently active diagram. Clicking and dragging the red rectangle, scrolls the diagram view in the diagram tab.



Documentation tab
Allows you to document any of the UML elements available in the Model Tree tab. Click the element you want to document and enter the text in the Documentation tab. The standard editing shortcuts are supported i.e. cut, copy and paste.



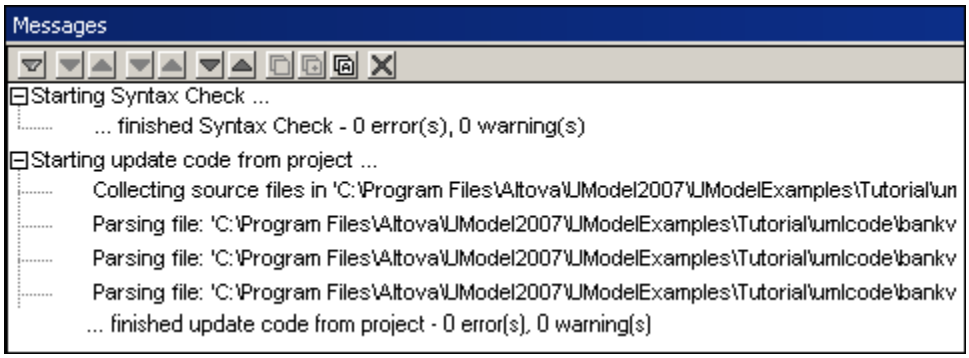
Documentation and code engineering:
During code engineering, only class and interface documentation is input/output. This includes documentation defined for class/interface properties and operations.

1. Select **Project | Synchronization** settings.
2. Activate the **"Write Documentation as JavaDocs"** check box to enable documentation output.

Please note:
When importing XML schemas, only the first annotation of a complex- or simpleType is displayed in the Documentation window.

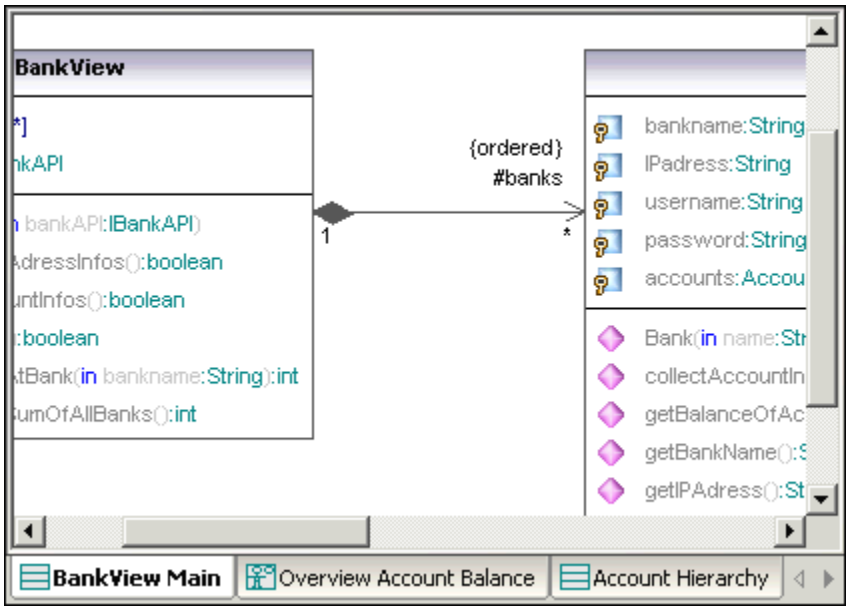
4.5 Messages window

The Messages window displays warnings, hints and error messages when merging code, or checking the project syntax.



4.6 Diagram pane

The diagram pane displays all the currently opened UModel diagrams as individual tabs.



To create a new diagram:

1. Click a package in the Model Tree tab.
2. Select **New | YYY Diagram**.

To create a new diagram containing contents of an existing package:

1. Right click a package and select **Show in new Diagram | Content**.

To open / access a diagram:

- Double click the diagram **icon** in any of the Model Tree pane tabs (to open).
- Clicking any of the tabs in the Diagrams pane (to access).

To close all but the active diagram:

- Right click the diagram tab that is to remain open, select the option **Close All but active**.

Deleting a diagram:

- Click the diagram icon in the Model Tree and press Del. key.

Moving diagrams in a project:

- Drag the diagram icon to any other package in the Model Tree Tab.
  You might have to enable the "no sort" option to move it.

Deleting elements from a diagram:

Delete element from the **diagram** and **project**!

- Select the element you want to delete and press the Del. keyboard key.

Delete element **from diagram only** - not from the project!

1. Select the element you want to "delete"
2. Hold down the **CTRL** key and press **Del**.

An auto-layout function allows you to define how you would like your diagram to be visually structured. Right click the diagram background and select either:

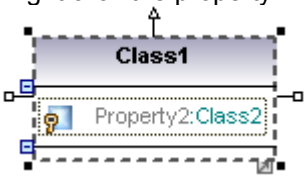
- Autolayout All | Force directed, or
- Autolayout All | Hierarchic

Showing relationships between modeling elements:

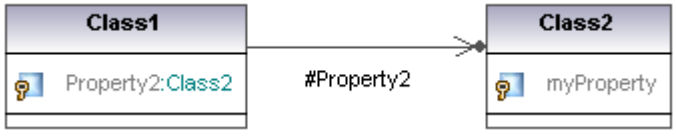
1. Right click the specific element and select **Show**.
   The popup menu shown below is context specific, meaning that only those options are available that are relevant to the specific element.

To show a class attribute/property as an association:

1. Right click the property in the class.



2. Select the menu option **Show | "PropertyXX" as Association**.
   This inserts/opens the referenced class and shows the relevant association.



Configuring diagram properties

Click on the diagram background and then select one of the styles from the Styles combo box. Please see Styles pane for more information.

To enlarge the Diagram size:
The size of the diagram tab is defined by the elements and their placement.

- Drag an element to one of the diagram tab edges to automatically scroll the diagram tab and enlarge it.

Positioning modeling elements - the grid

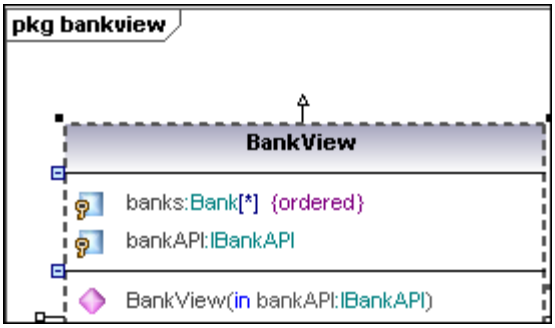
Modeling elements can be positioned manually, or made to snap to a visible/invisible grid in a diagram.

 toggles between showing / hiding the grid

 toggles between snapping elements to the visible / invisible grid

Displaying the UML diagram heading

 toggles between displaying the UML diagram heading, i.e. the frame around a diagram with its name tag in the top left corner.



4.6.1 Cut, copy and paste in UModel Diagrams

Cut, Copy and Paste of diagram elements within the Diagram pane
All UModel diagram elements can be cut, copied and pasted within, across the same type, and even into other types of diagram tab. Mouse or keyboard shortcuts can be used to achieve this in two different ways:

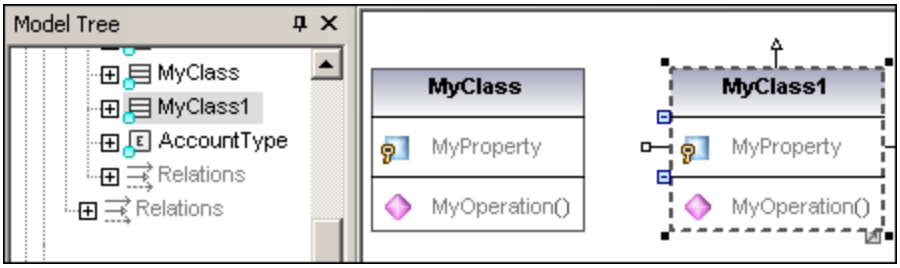
Having copied an element:

- "**Paste**", using the keyboard shortcut CTRL+V, or "Paste" from the context menu, as well as Paste from the Edit menu, always adds a **new** modeling element to the diagram and to the Model Tree.
- "**Paste in diagram only**", using the context menu, i.e. right clicking on the diagram background, only adds a "link/view" of the existing element, to the current diagram and not to the Model Tree.

Using the Class diagram as an example:

Paste (CTRL+V) of a copied class:

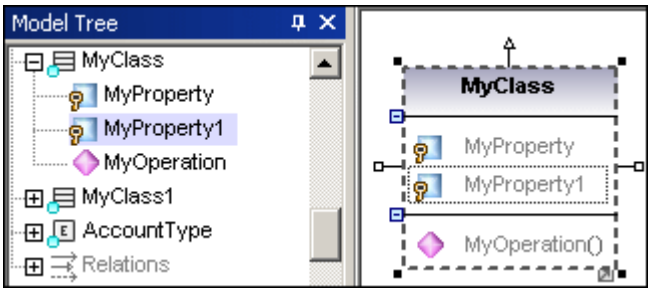
- **Pasting** a copied class in the same diagram (or package), inserts a **new class** with the source class name plus a sequential number. E.g source class name is **myClass**, pasted class name is **myClass1.** All operations and properties are also copied to the new class.



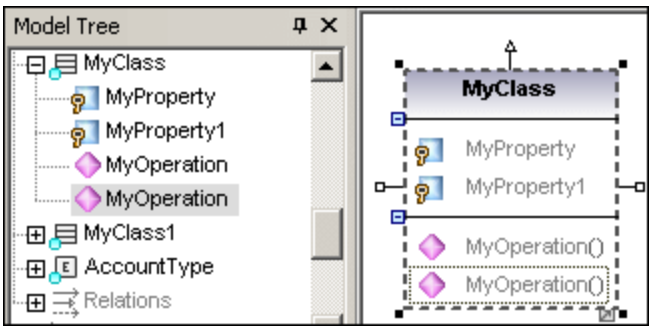
- **Pasting** a copied class into a different package, also inserts a **new** class, but keeps the original class name.
- In both cases the new class is also added to the Model Tree as well.

Paste (CTRL+V) of copied Properties or Operations:

- **Pasting** a Property in the same class, inserts a **new** property with the source property name plus a sequential number e.g. MyProperty1.



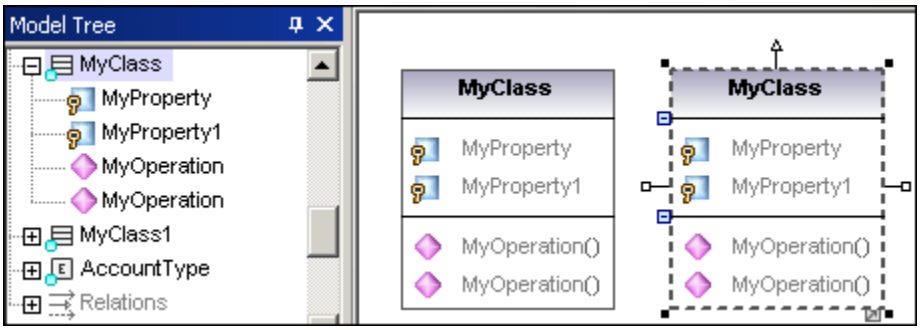
- **Pasting** an Operation in the same class, inserts a **new** operation of the same name as the source operation.



- In both cases a new property/operation is added to the Model Tree.

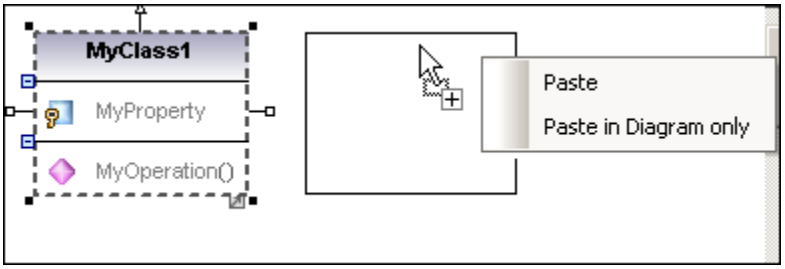
"Paste in Diagram only":
Whenever you use the context menu and select this option, a "link", or "view" to the element is created in the **diagram** you paste it into. Using the Class diagram as an example:

- "Paste in diagram only", creates a "view" to the original class
- The class is inserted into the diagram and displayed exactly as the source class
- A **new** class has **not** been added to the Model Tree!
- No class name or other Operation/Property changes are made
- Changing element properties in one of the "views", changes it in the other one automatically



Copy and pasting of elements using the mouse:

1. Click on the modeling element you want to copy.
2. Move the mouse pointer to the position you want to place the new element.
3. Hold down the CTRL key. A small plus appears below the mouse pointer to signify that this is a copy procedure.
4. Release the mouse button.



A popup menu appears at this point allowing you to select between Paste, and Paste in Diagram only.

5. Select the option that you would like to perform.

Please note:

Using the mouse and CTRL key allows you to copy, or move properties and operations directly within a class.

4.7 Adding/Inserting model elements

Model elements can be created and inserted into diagrams using several methods:

- By adding the elements to specific packages, in the Model Tree view
- By dragging existing elements from the Model Tree tab into the diagram tab
- By clicking a specific UML element icon, and inserting it into the diagram
- By using the context menu to add elements to the diagram (and automatically to the Model Tree view).

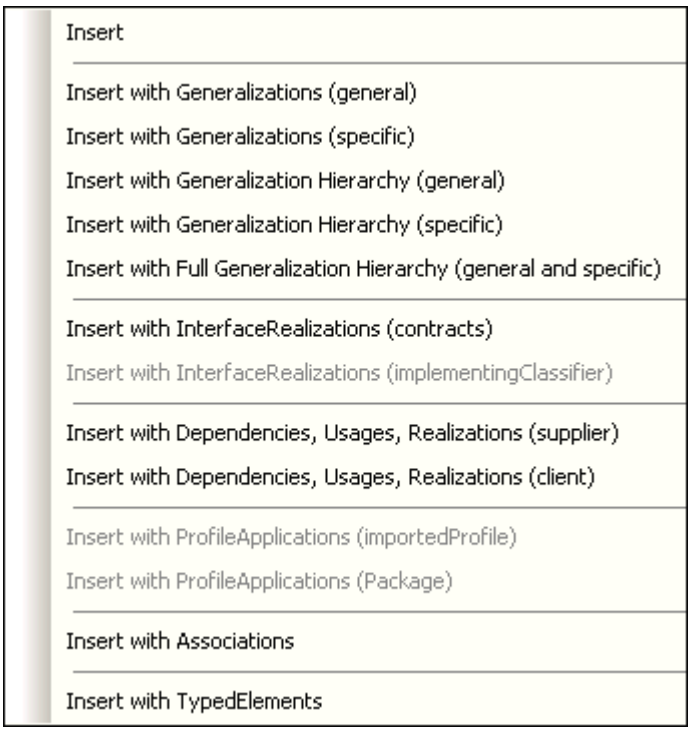
  Please note that multiple elements can be selected in the Model Tree using either SHIFT+click, or CTRL+click.

Adding elements in the Model Tree/Favorites tab:

- Right click a package, select New, and then select the specific element from the submenu.
  This adds the new element to the Model Tree tab in the current project.

Inserting elements from the Model Tree view into a diagram:
Model elements can be inserted individually, or as a group. To mark multiple elements use the CTRL key and click each item. There are two different methods of inserting the elements into the diagram: drag left, and drag right.

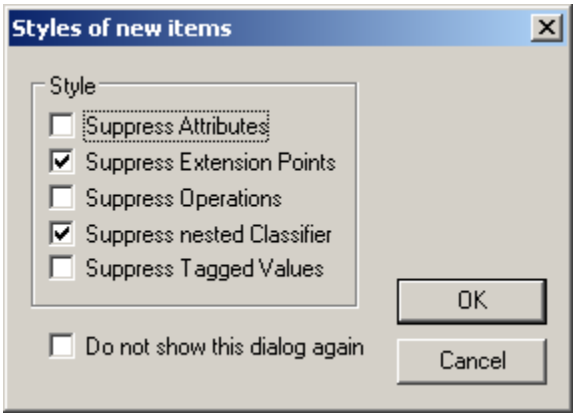
- **Drag left** (normal drag and drop) inserts elements immediately at the cursor position (any associations, dependencies etc. that exist between the currently inserted elements and the new one, are automatically displayed).
- **Drag right** (holding down the right mouse button and releasing it in the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



Example:

You want to replicate the Account Hierarchy diagram in a new class diagram.

1. Right click the **bankview** package and select **New | Class Diagram**.
2. Locate the abstract *Account* class in the model tree, and use **drag right** to place it in the new diagram.
   The context menu shown above, is opened.
3. Select the **Insert with Generalization Hierarchy (specific)** item.



4. Deselect the check boxes for specific items you want to appear in the elements (Properties and Operations in this case).
5. Click OK.
   The Account class **and its three subclasses**, are all inserted into the diagram tab. The Generalization arrows are automatically displayed.

Adding elements to a diagram using the icons in the icon bar:

1. Select the specific element you want to insert by clicking the associated icon in the icon bar.
2. Click in the diagram tab to insert the element.

Please note:
Holding down the CTRL key before clicking in the diagram tab, allows you to insert multiple elements of the same type with each individual click in the diagram.

Adding elements to a diagram using the context menu:

- Right click the diagram background and select **New | element name**.

Please note:
Adding new elements directly to the diagram tab, automatically adds the same element to the Model Tree tab. The element is added to the package containing the UML diagram in the Model Tree view.

- Right click an element and select **Show** | **xx**
  E.g. Right clicking the Account class and selecting **Show | Generalization hierarchy**. This then inserts the derived classes into the diagram as well.

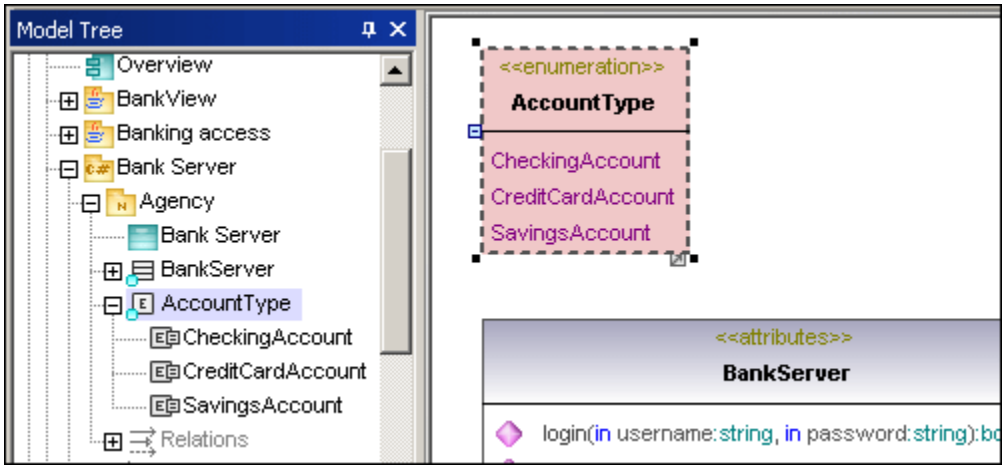
4.8 Hyperlinking modeling elements

UModel now supports automatic and manual hyperlinking of modeling elements. Automatic hyperlinking occurs when selecting the specific setting when importing source code, or binary files, into a model.

Manual hyperlinks are created between any modeling elements (except for lines) and:

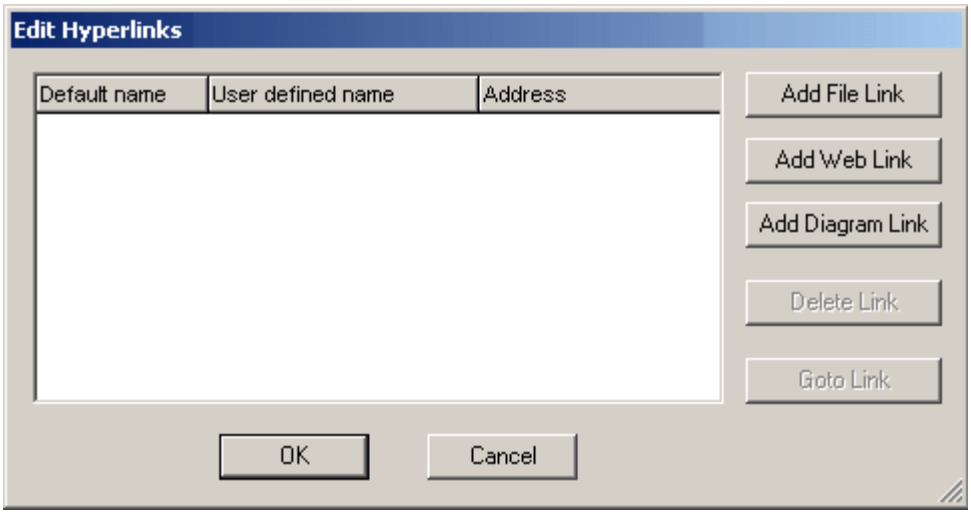
- any diagram in the current ump project
- any diagram on a different ump project
- any element on a diagram
- external documents, e.g. PDF, Excel or Word documents
- web pages

Opening the Bank Server diagram under the Bank Server package displays the IBankAPI interface as well as the BankServer class. An enumeration element containing the names of the EnumerationLiterals is also visible. What we want to do is create a hyperlink from the Enumeration to the Account Hierarchy class diagram.



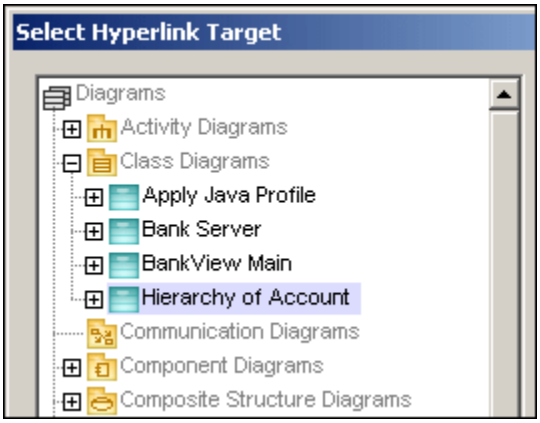
To create a diagram hyperlink:

1. Right click the element and select **Hyperlinks | Insert/Edit hyperlinks**.

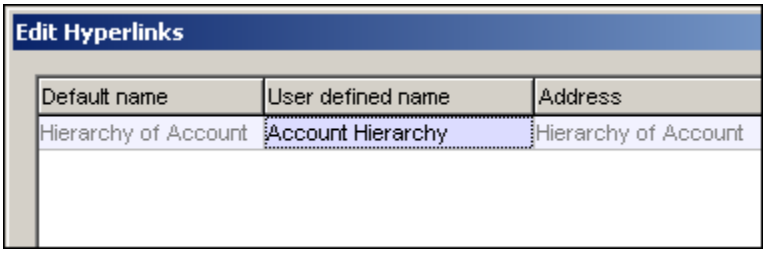


This opens the Edit Hyperlinks dialog box in which you manage the hyperlinks.

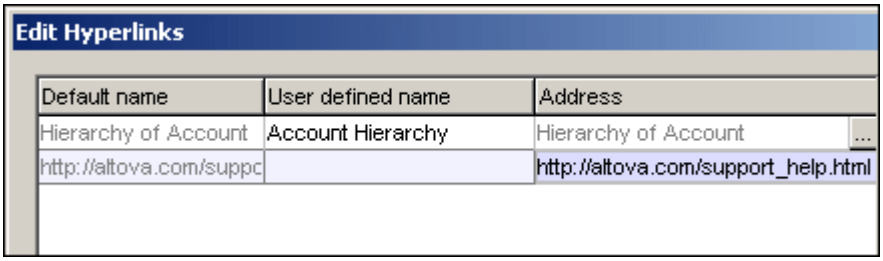
2. Click the **Add Diagram Link** button to define a link to an existing diagram.



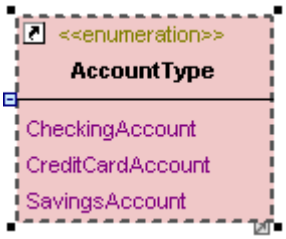
3. Select the hyperlink target that you want to be able to navigate to, e.g. Hierarchy of Account diagram, and click OK.



Double clicking in the User defined name column allows you to define your own link name.
Note that you can add multiple, as well as different kinds of links from a single modeling element e.g. a web link to http://altova.com/support_help.html using the **Add Web Link** button.

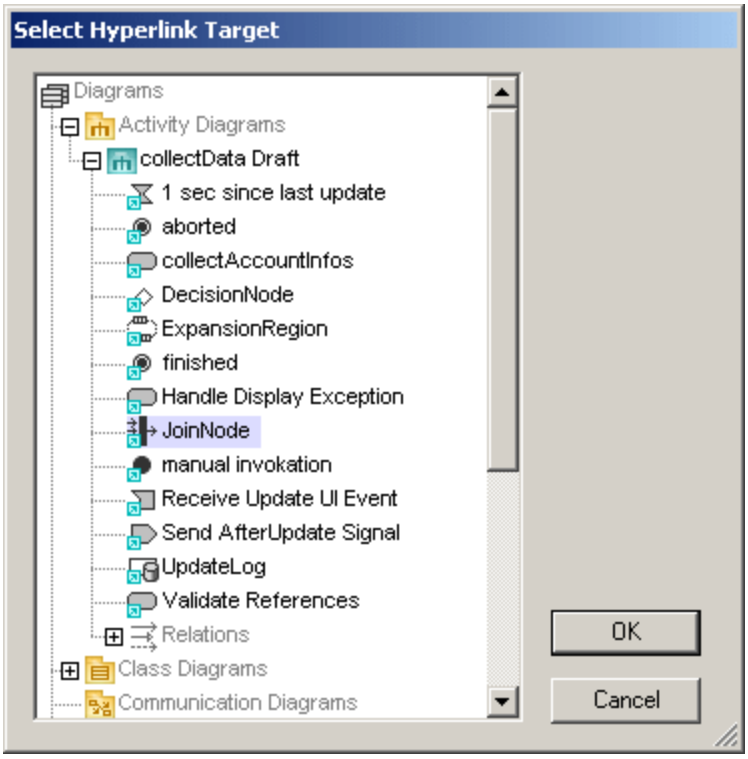


4. Click OK when you have finished defining your hyperlinks.
A link icon has now been added to the top left of the enumeration element.



To create a link to a specific diagram element:

1. Create the hyperlink as before but click the + sign to expand the diagram contents.



2. Select the specific modeling element you want to link to and click OK to confirm.

   Clicking the link icon opens the designated diagram with the element visible and selected.

To create a link to a document:

1. Click the **Add File Link** button in the Edit Hyperlinks dialog box.
2. Select the document that you want to link e.g. *.DOC, *.XLS, *.PDF etc.

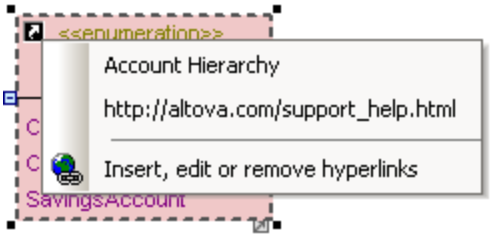
To create a hyperlink from a note:

1. Select the text in the note by dragging or double clicking a word.
2. Right click the selected text and select the menu object **Insert/Edit Hyperlinks**.
3. Use the Edit Hyperlinks dialog box to create a link to a diagram.

To navigate to a hyperlink target:

1. Click the hyperlink icon in the modeling element.
   If only one target is defined then the target diagram, website etc., will appear immediately.

   If multiple targets were defined, a popup dialog appears allowing you to select one of the available targets.



Clicking the first item opens the Hierarchy of Account diagram.

Navigating hyperlinks:

- Click the Previous  and Next  icons, in the main icon bar, to navigate the source and destination links.

To edit/change a hyperlink target:

1. Right click the link icon and select **Insert, edit or remove hyperlinks** item.
2. Use the Edit Hyperlinks dialog box in to manage your hyperlinks.

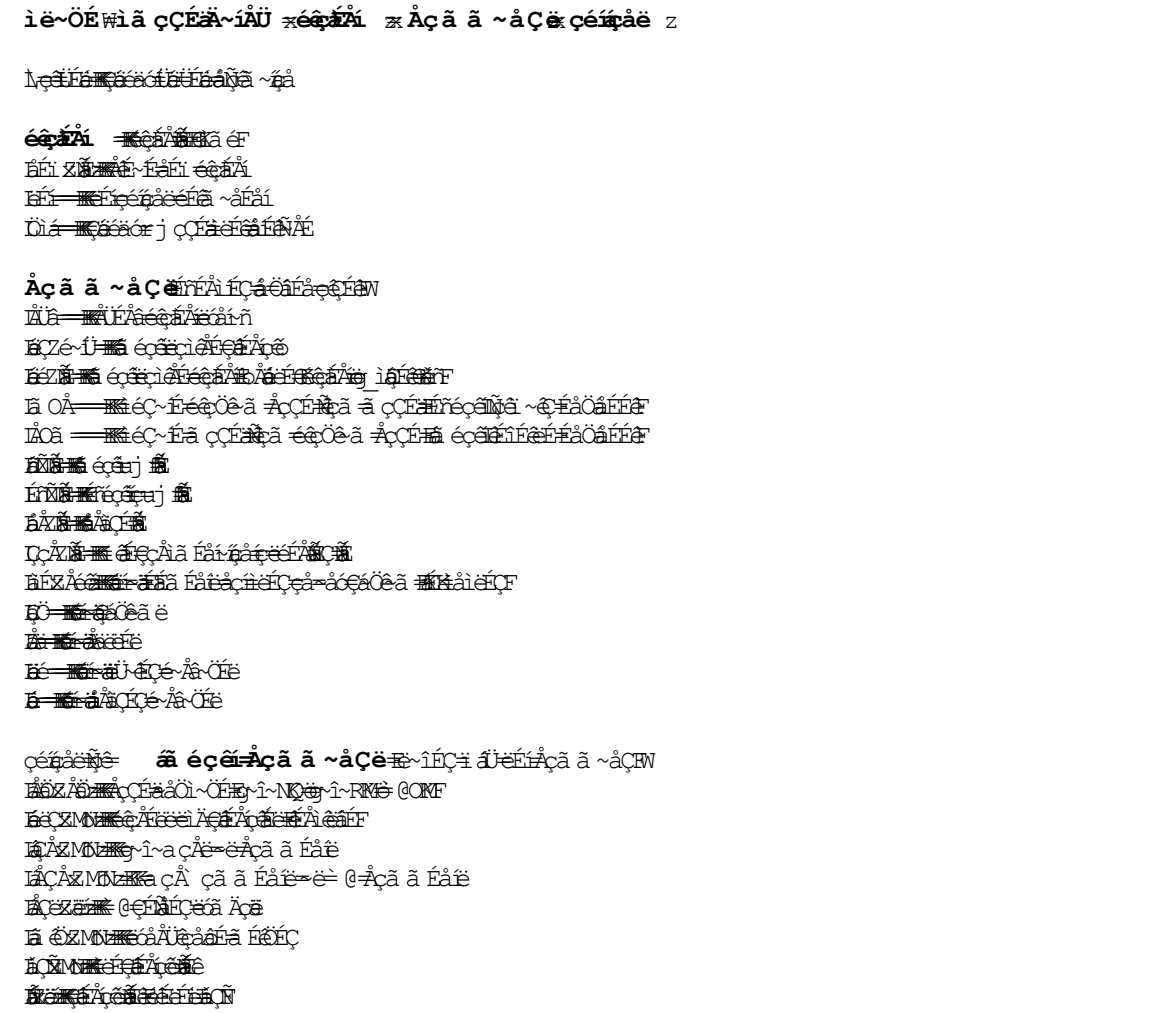
4.9 UModel Command line interface

UModel now supports batch-processing. A **UModelBatch.exe** file is available in the ...**UModel 2007** folder.

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

Please note:

If the path, or file name contains a space, please use quotes around the path/file name i.e. "c:\Program Files\...\File name"



```
çéíáçåëÑê=    Éñéçêí=Åçã ã ~åÇë=Ñê~îÉÇ=ï áÜ=ëÉíÅçã ã ~åÇEW
ÉàÇÅxZM®NzKKÅçã ã Éåí=ëç~î~açÅë
ÉÅÇÅxZM®NzKK=@=Åçã ã Éåí=ë~açÅ çã ã Éåíë
ÉëéxZM®NzKKëÉìëÉÇÉÑåÉÇ=pmi=Éã éäÉë
ÉÅçÇxZM®NzKKÅçã ã Éåíçìíç ÉäÉÇ
Éã êxZM®NzKKéåÅÜêçåáÉã ÉêÉÇ
ÉÖÑxZM®NzKKÖÉåÉê Éã ëéåÖÑá~ã Éë
ÉìëÅxZM®NzKKëÉéåí~ñÅÜÉÅâ

çéíáçåëÑê=    uj f=Éñéçêí
ÉñçxZM®NzKKÑéçêí=r fä ë
ÉñÉñxZM®NzKKÑéçêí=j çÇÉäÉÅÅÅÉñÉåëáçåë
ÉñìîxìÉêzKK=j i=ÉêëáçåE=j iOKMë=j iOKNF

çéíáçåëÑê=    ÇçÅìã Éåí~íáçå =ÖÉåÉê~íáçå
ÇçÇñZÑ ízKKçìíéìíÑçã ~íEqj i=ö qc=ö pt l oaF
```

In the projects section:

The **/new** parameter defines the path and file name of the new project file (*.ump).

The **/set** parameter overwrites current default settings in the registry, with the options/settings defined here.

The **/gui** parameter displays the UModel interface during the batch process.

Example 1:
Import source code and create new project file:

```
? ÿ ÑêçÖê~ã =cáäÉëÿ ~äíçî~ÿ        rj çÇÉäOMMT ÿ rj çÇÉä_~íÅÜKÉñÉ?=åÉï Z? ÿ ÑêçÖê~ã =cáäÉëÿ ~äíçî~ÿ
rj çÇÉäOMMT ÿ rj çÇÉä_~íÅÜl ìíÿ céÉÇKìã é?=áëÇZ?u WÿqÉëí ~ëÉëÿ rj çÇÉäÿ céÉÇ?=ëÉí=ÖìáÌ
áÅäÖZg~î~RKM=áéëÇZN=éÑÇZN=áÖÇÅZN=ÇÖÉåZN=ÇçéåZN=Çã ~ñZR=ÅÜâ
```

/**new**:	Specifies that the newly-created project file should be called "Fred.ump" in C:\Program Files\Altova\rj çÇÉäOMMT \UModelBatchOut\
/**isd**=	Specifies that the root directory to import into 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 GUI during batch processing
/**iclg**:	UModel will import the code as Java5.0
/**ipsd**=1:	recursively process all subdirectories of the root directory specified in the /**isd** parameter
/**pfd**=1:	creates packages in the UModel project for each imported directory
/**ijdc**=1:	created JavaDoc from comments where appropriate
/**dgen**=1:	generates diagrams
/**dopn**=1:	opens generated diagrams
/**dmax**=5:	will open a maximum of 5 diagrams
/**chk**:	performs a syntax check

Example 2:
Imports source code from X:\TestCases\UModel, and saves the resulting project file in "C:\Program...".

```
? WmêçÖêã=cáäéëy^äíçî~yrjçÇÉä        OMMT yrjçÇÉä_~íÅÜKÉñÉ?=áëÇ? Wymêç Öêã=cáäéëy^äíçî~y
rjçÇÉäOMMT yrjçÇÉä_~íÅÜl ìí  yä~åâäéëKã é?=éÇZ?uWqÉëí ~ëÉëyrjçÇÉäy
?=ÅäöZg~î~RKM=áéëÇZN=áçÅZN=çÖÉåZN=çéåZN=ã ~ñZR=ëí~íZN=ëåÅZN=ÅÜâ
```

/**dsat**=1: suppresses attributes in the generated diagrams
/**dsnc**=1: suppresses nested classifiers in the generated diagrams

Example 3:
Synchronize code using existing project file (e.g. one of the ones created above).

```
? WmêçÖêã=cáäéëy^äíçî~yrjçÇÉä        OMMT yrjçÇÉä_~íÅÜKÉñÉ?? WmêçÖêã=cáäéëy^äíçî~y
rjçÇÉäOMMT yrjçÇÉä_~íÅÜl ìíycêÉÇKìã é?=ã OÅ=ÉàÇÅZN=ÉÅçÇZN=Éã êÖZN=ÉÖÑåZN=ÉìëÅZN
```

"C:\Program Files\Altova\UModelOMMT \UModelBatchOut\Fred.ump": the project file we want to use.

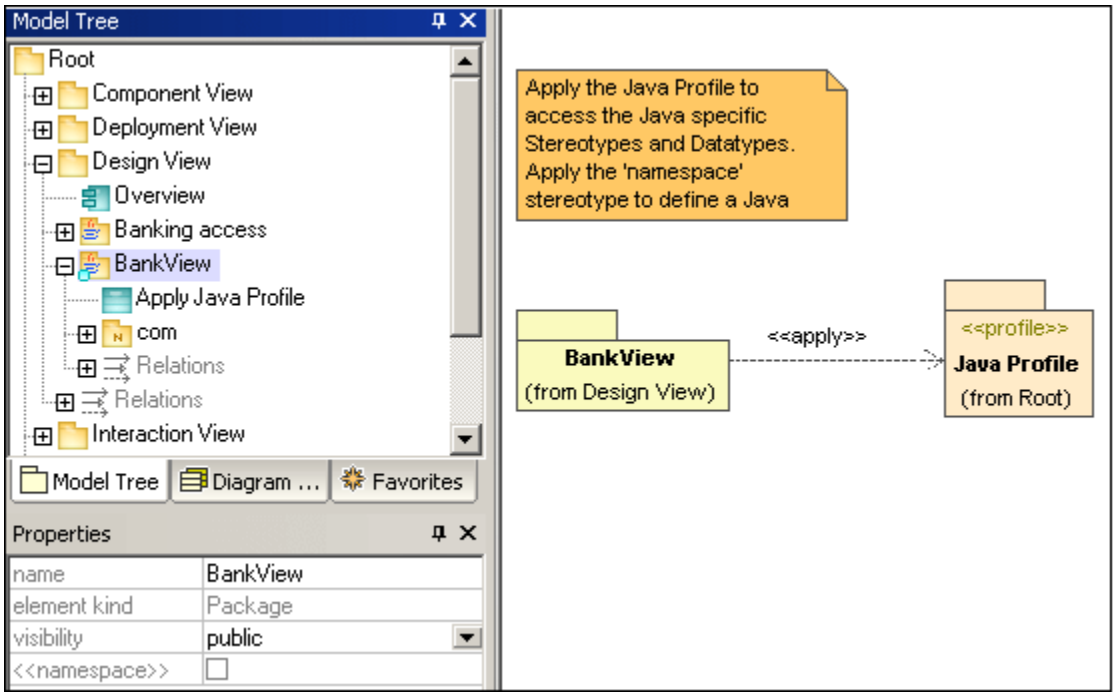
/**m2c**	update the code from the model
/**ejdc**:	comments in the project model should be generated as JavaDoc
/**ecod**=1:	comment out any deleted code
/**emrg**=1	synchronize the merged code
/**egfn**=1:	generate any missing filenames in the project
/**eusc**=1	use the syntax check

4.10 Bank samples

The ...**UModelExamples** folder contains sample files which show different aspects of UML modeling in UModel. They are designed to show language specific models for Java, C# and a combination of both languages in one modeling project.

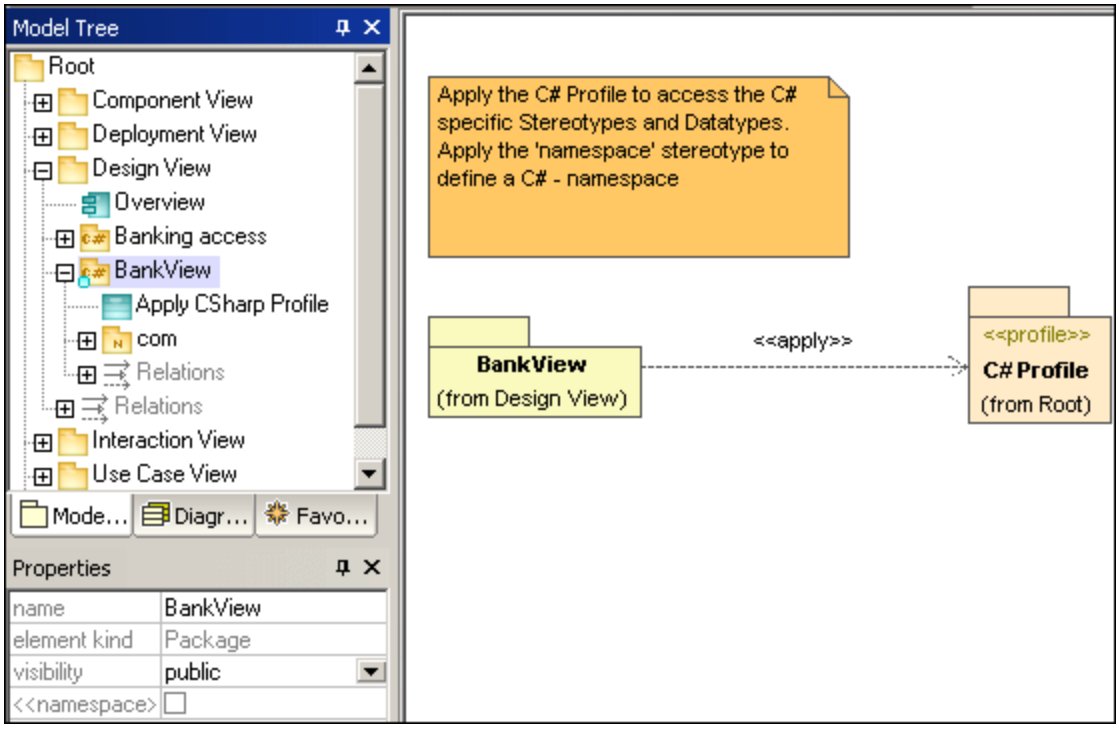
The **Bank_Java.ump** sample file is shown below:

- the Java profile has been assigned to the Bankview package
- the Java namespace root has been assigned to the Banking access and BankView packages.
- the Interaction View package contains two interaction elements which each contain a sequence diagram.



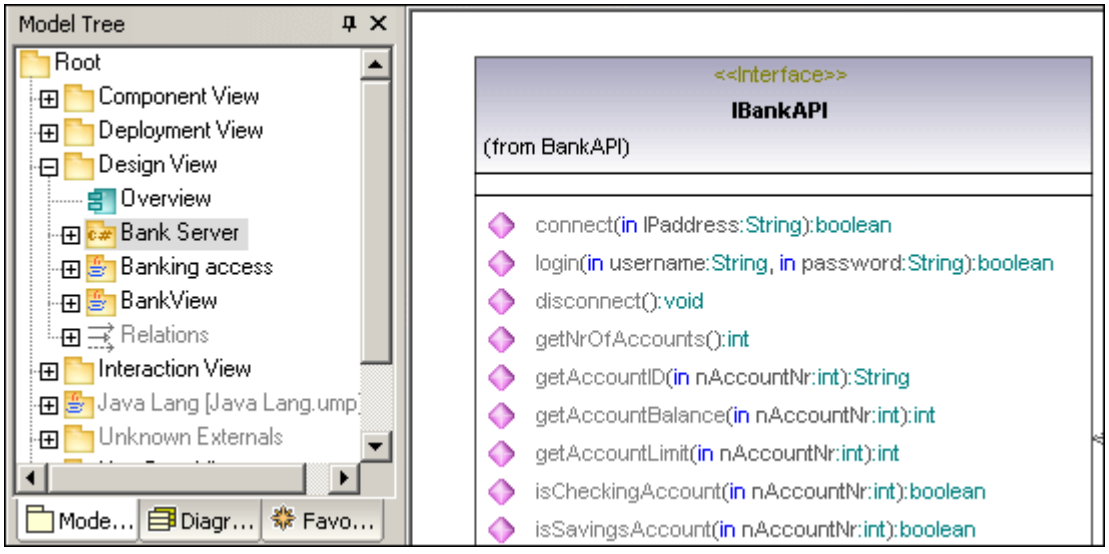
The **Bank_CSharp.ump** sample file is shown below:

- the C# profile has been assigned to the BankView package
- the C# namespace root has been assigned to the Banking access and BankView packages.
- the Interaction View package contains two interaction elements which each contain a sequence diagram.



The **Bank_MultiLanguage.ump** sample file is shown below:

- the Java profile has been assigned to the BankView package
- the C# namespace root has been assigned to the Bank Server package
- the Java namespace root has been assigned to the BankView package.
- the Interaction View package contains two interaction elements which each contain a sequence diagram.



Chapter 5

Projects and code engineering

5 Projects and code engineering

UModel now supports all Java specific constructs, among them:

- Java annotations
- Attributes, operations and nested qualifiers for EnumerationLiterals
- Enumerations can realize interfaces
- Netbeans project files

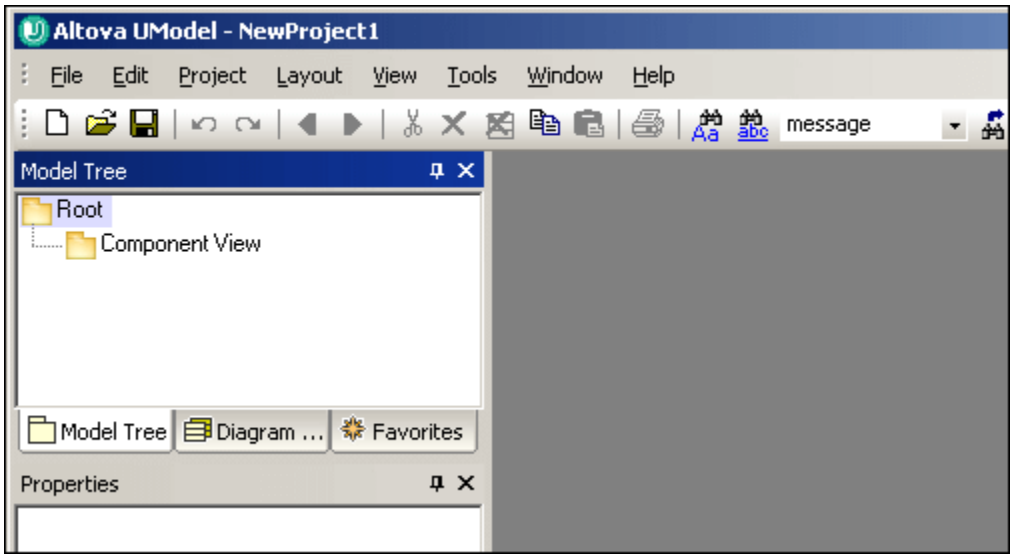
Reverse engineering now supports:

- The ability to generate a single diagram for all reverse engineered elements
- Possibility to show/hide anonymous bound elements on diagrams
- Ability to automatically create hyperlinks from packages to their corresponding package content diagrams during the import process.

To create a new project:

1. Click the **New** icon in the icon bar, (or select the menu item **File | New**).

The Root and Component packages are automatically inserted when a new project is created, and are visible in the Model Tree tab. A new project with the default name NewProject1 is created. Note that starting UModel opens a new project automatically.



A newly created UModel project consists of the following packages:

- Root package, and
- Component View package
  These two packages are the only ones that cannot be renamed, or deleted.

All project relevant data is stored in the UModel project file, which has an *.**ump** extension. Each folder symbol in the Model Tree tab represents a UML package!

UModel Project workflow:

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) that you want; Note that all model elements can be inserted, renamed, and deleted in the Model Tree tab itself, you are not even forced to create them as part of a diagram.

To insert a new package:

1. Right click the package you want the new package to appear under, either Root, or Component View in a new project.
2. Select **New | Package**.

   A new package is created under an existing one. The name field is automatically highlighted allowing you to enter the package name immediately.

- Packages are the containers for all other UML modeling elements, use case diagrams, classes, instances etc.
- Packages can be **created**, at any position in the Model Tree.
- Packages/contents can be **moved**/copied to other packages in the Model Tree (as well as into valid model diagrams in the diagram tab).
- Packages and their contents can be **sorted** (in the Model Tree tab) according to several criteria.
- Packages can be placed within other packages.
- Packages can be used as the **source**, or **target** elements, when merging, or synchronizing code.

To have elements appear in a UML diagram, you have to:

1. Insert a new UML diagram, by right clicking and selecting **New | (Class) Diagram**.
2. Drag and drop an existing model element from the Model Tree into the newly created Diagram, or
3. Use the context menu within the diagram view, to add new elements directly.

To save a project:

Select the menu option **File | Save as...** (or File | Save).

To open a project:

Select the menu option **File | Open,** or select one of the files in the file list.

Please note:

Changes made externally to the project file, or included file(s), are automatically registered and cause a prompt to appear. You can then choose if you want to reload the project or not.

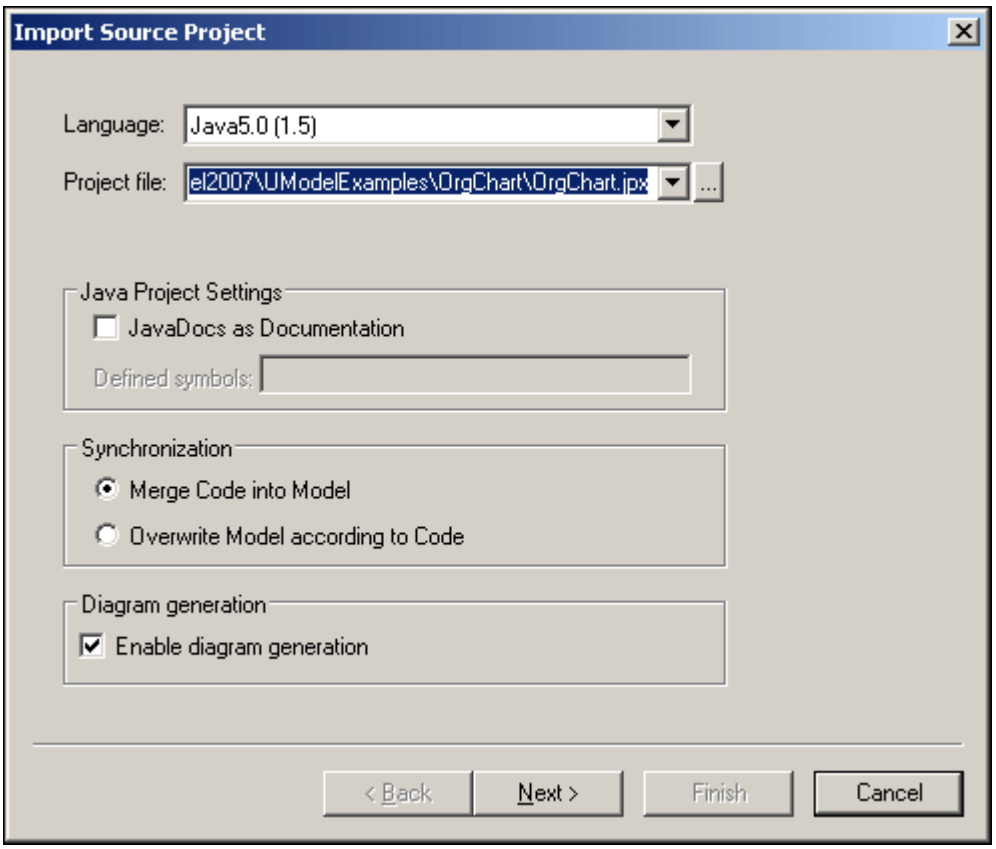
5.1 Importing source code into projects

Source code can be imported as a source project or as a source directory. For an example of importing a **source directory** please see Round-trip engineering (code - model - code) in the tutorial.

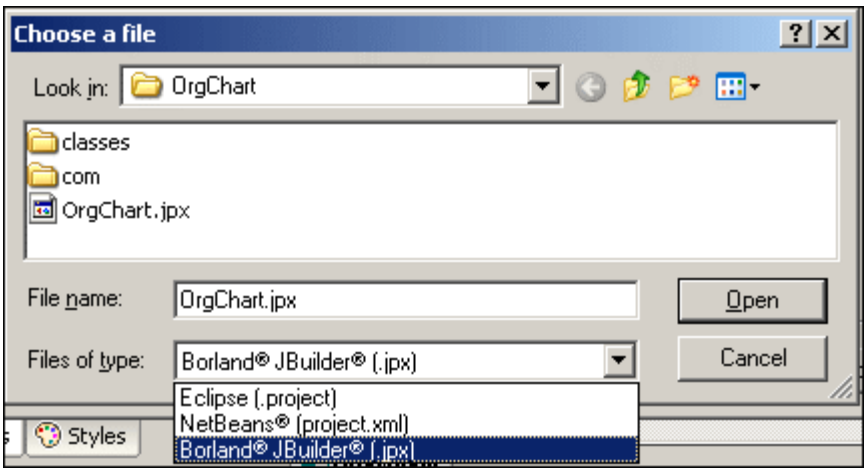
- JBuilder **.jpx,** Eclipse **.project** project files, as well as NetBeans (project.**xml**) are currently supported.
- C# projects:
  - MS Visual studio.Net projects, **csproj**, **csdprj**..., as well as
  - Borland **.bdsproj** project files

To import an existing project into UModel:

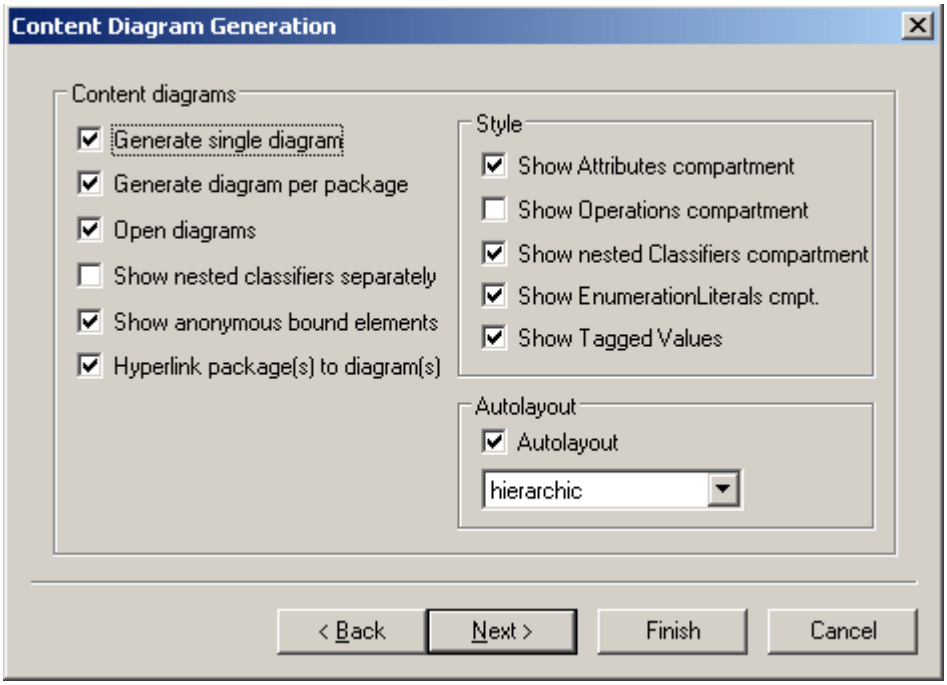
1. Select **Project | Import source project**.
2. Click the browse button [...] in the "Import Source Project" dialog box.



3. Select the project file type e.g. **.jpx** and click Open to confirm. This Jbuilder project file is available in the OrgChart.zip file in the ...**UModelExamples** folder.

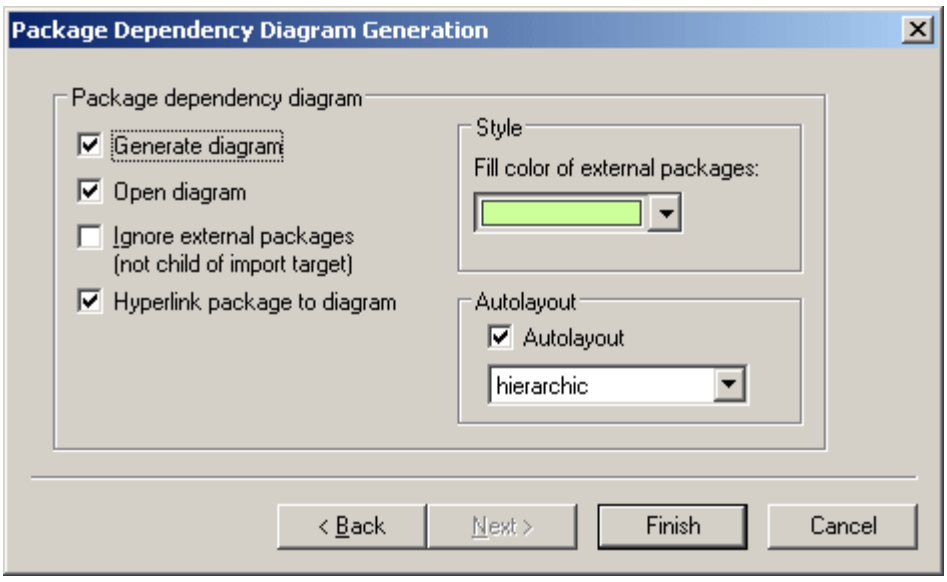


4. Make sure that you have activated the Enable diagram generation check box, and select any other specific import settings you need, and click Next.



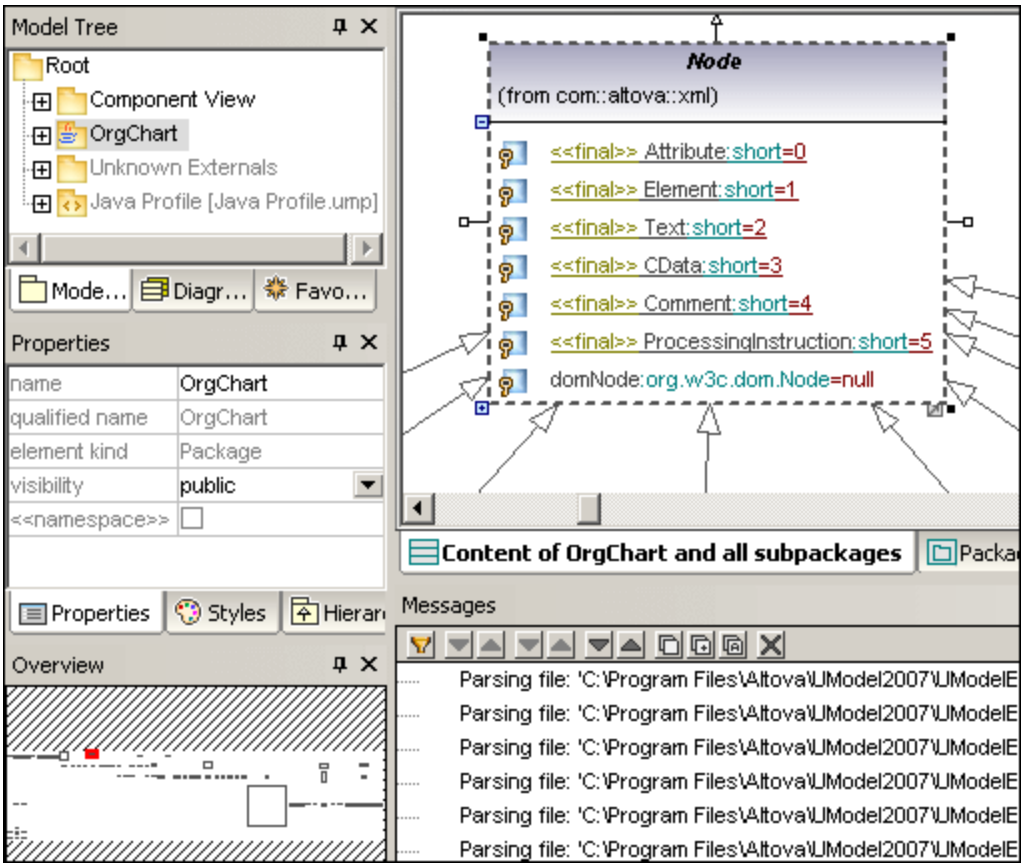
Note that UModel can generate a single overview diagram and/or a diagram for each package. The settings shown above are the default settings.

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



6. Click Finish to use the default settings.

   The project is parsed and the UModel model is generated.

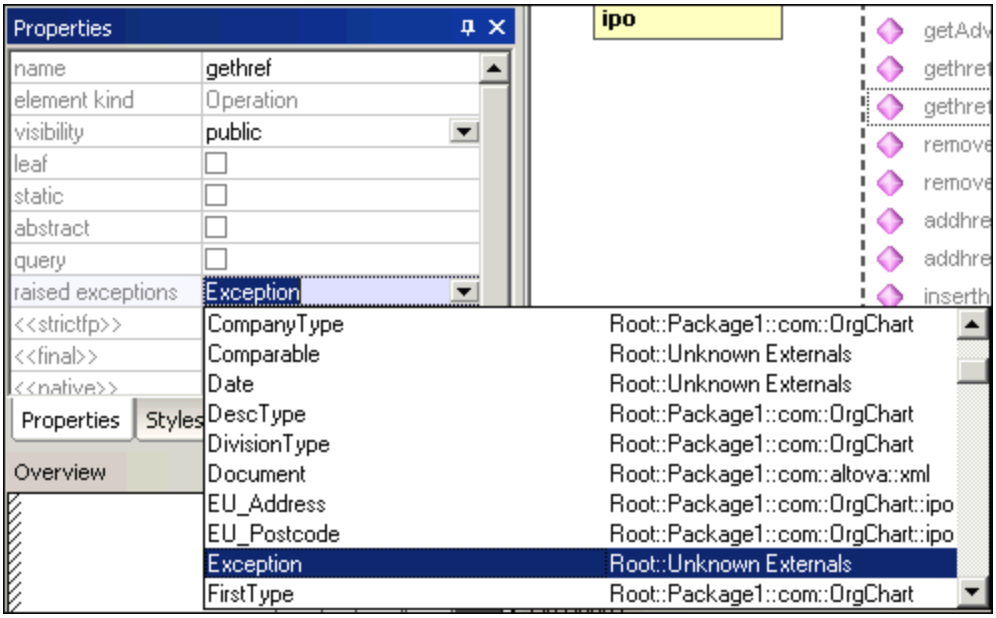


Please note:

If you are importing into an existing project, you will be prompted for the package it should be imported into. If you are using a new project, an OrgChart folder is automatically created.

Raised exceptions

Clicking an operation in one of the classes, then clicking the **Exception** combo box, displays the exception information that an operation can throw.



5.2 Importing C# and Java binaries

UModel now supports the import of C# and Java binaries. This is extremely useful when working with binaries from a third party, or the original source code has become unavailable.

If you intend to import Java and/or C# binary files, the following programs/components must be installed:

Java 1.4/5.0:
Sun Java Runtime Environment (JRE), or Development Kit (JDK) in Versions 1.4, 1.5, 1.6

UModel support:
Type import is supported for all Class Archives targeting these environments, i.e. adhering to the Java Virtual Machine Specification.

C# 2.0:
.NET Framework 2.0, 3.0

UModel support:
Type import is supported for Assemblies targeting:
.NET Framework 1.1, 2.0, 3.0
.NET Compact Framework v1.0, v2.0 (for PocketPC, Smartphone, WindowsCE)

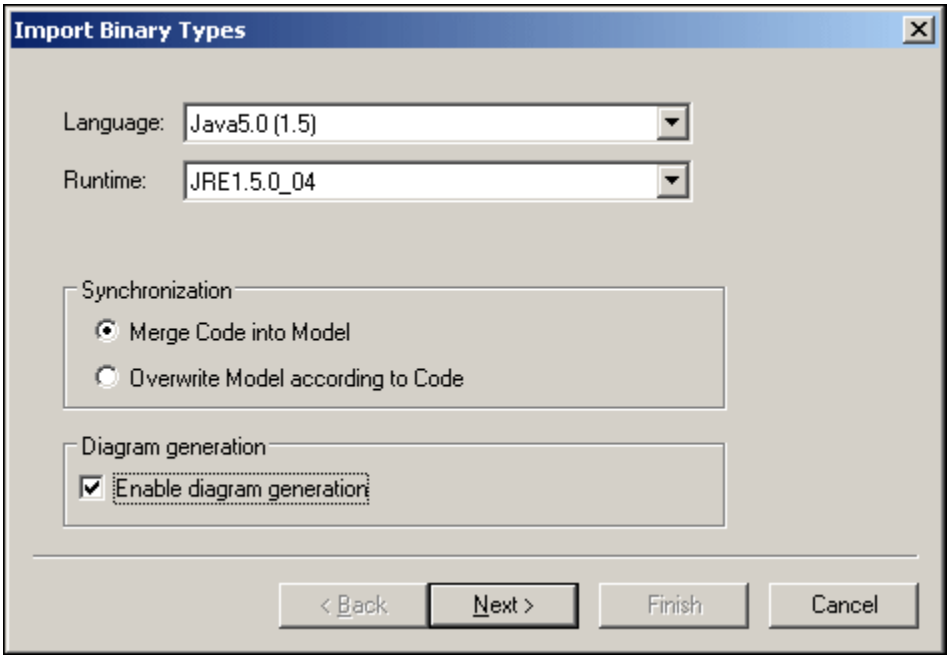
Restrictions:
Assembly mscorlib with the .NET core types can only be imported from the .NET Framework 2.0

These requirements only apply if you intend to import Java or C# binaries; if you do not, there is no need for the Java Runtime Environment, or the MS .NET Framework to be installed.

The import of either Java, or C#, obfuscated binaries is not supported.

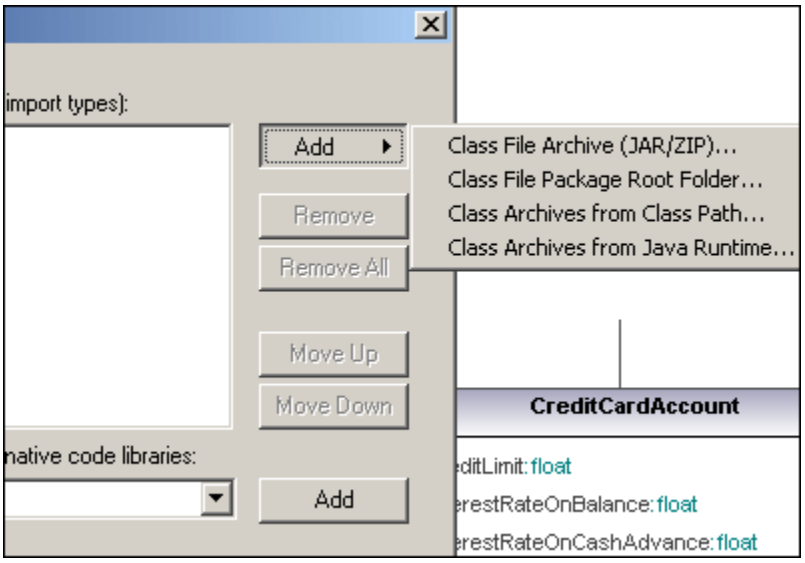
To import binary files:

1. Select the menu option **Project | Import Binary Types**.

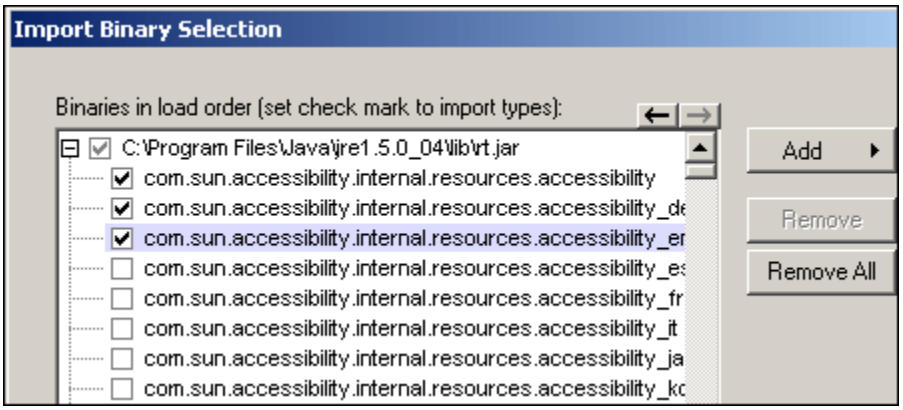


2. Select the language and runtime edition, then click Next.
This opens the Import Binary Selection dialog box.

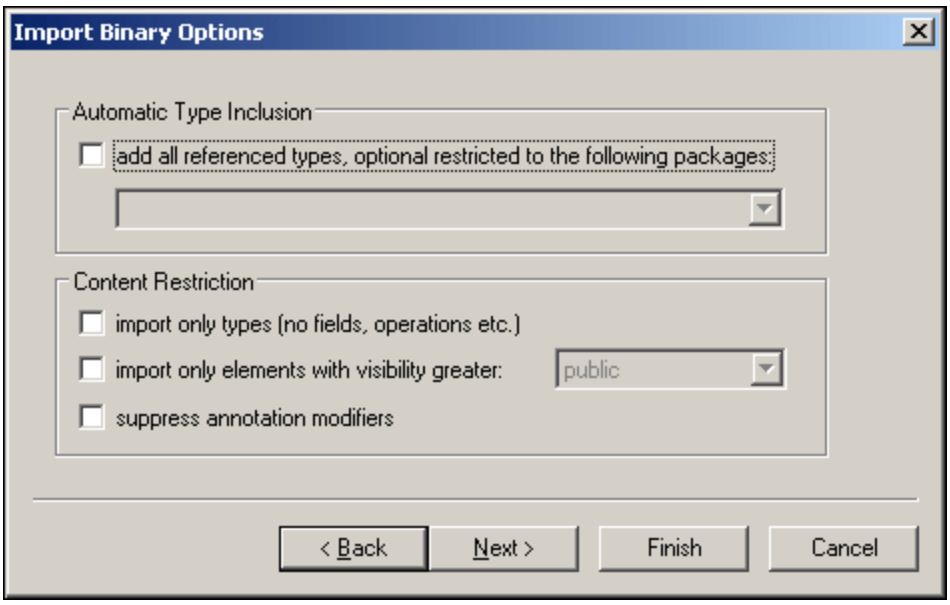
3. Click the Add button and select the Class Archive from the flyout window, e.g. Class Archives from Java Runtime... .



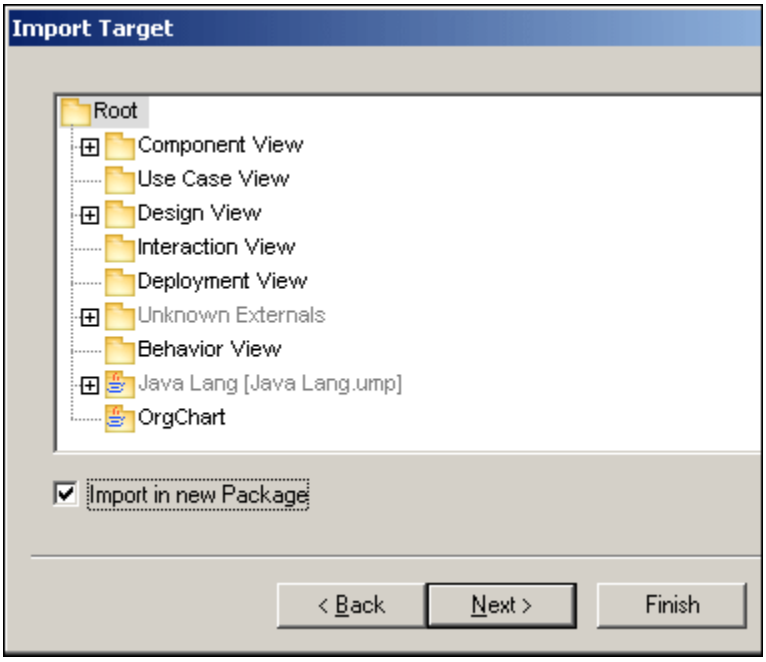
3. Click the "+" expand button to expand the list of binaries, and activate the check box (es) of those that you want to import (the first three in the screen shot below), then click Next.



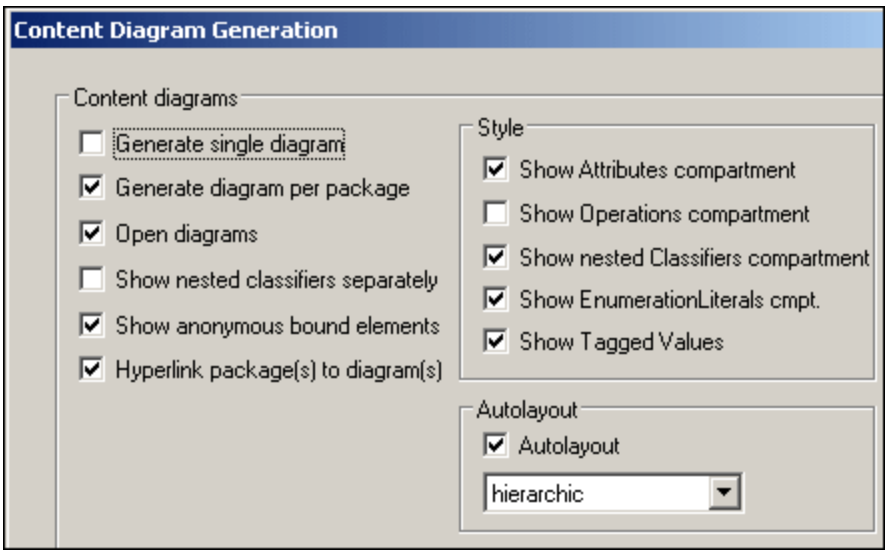
This opens the **Import Binary Options** dialog box.



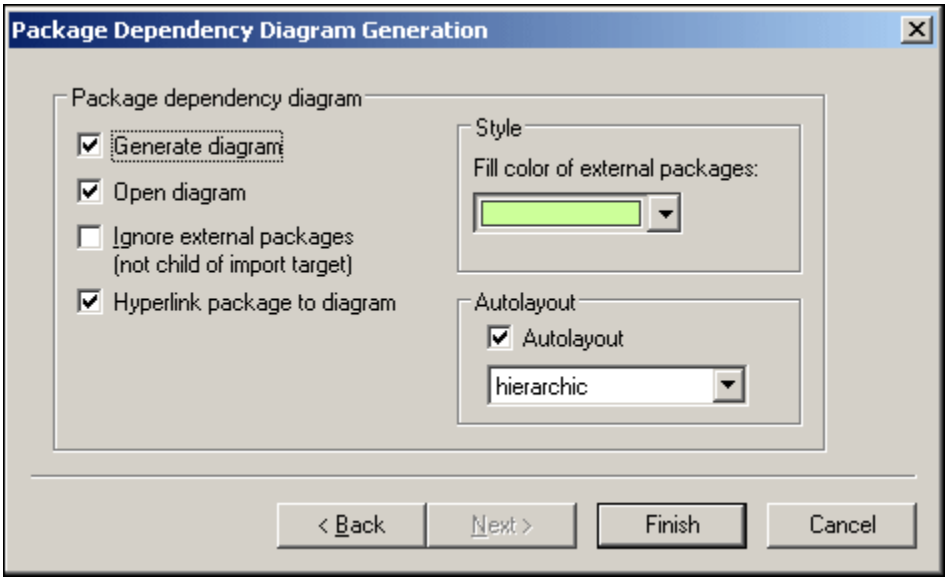
4. Select the specific options you need and click Next to continue.



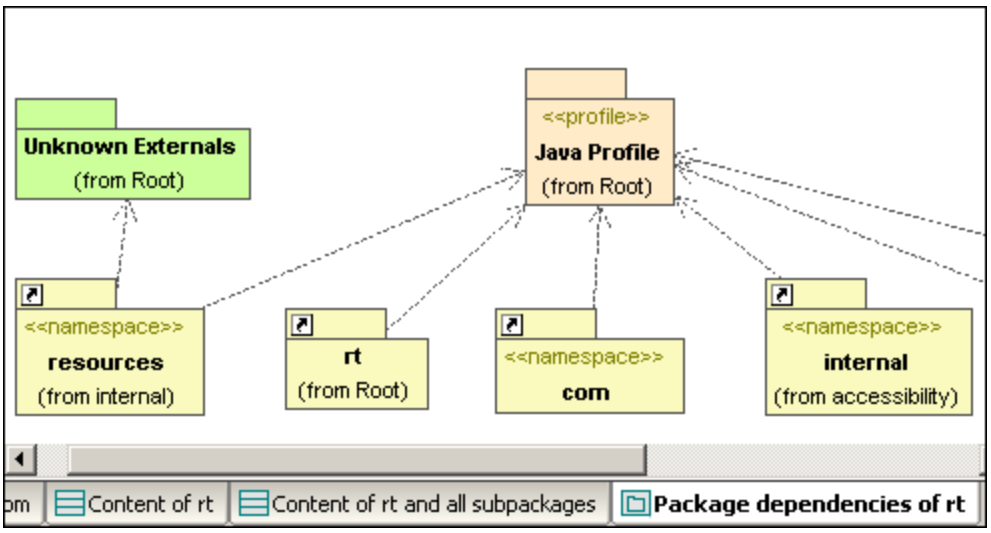
5. Define the Import Target, or click the Import in new Package check box, then click Next.



6. Select the Content Diagram Generation properties from the dialog box and click Next to continue.
   Note that you can generate a single diagram for each package, as well a single overview diagram.

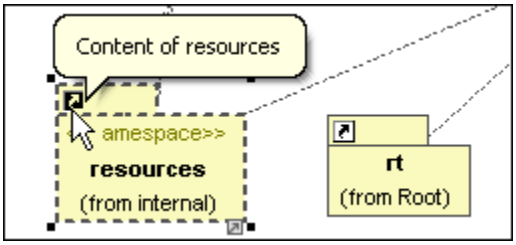


7. Select the Package Dependency options that you would like to include and click Finish to complete the import procedure.
   The screenshot below shows the diagram containing the package dependencies of the Java binaries.



8. Click the other tabs to see the class files etc.

Please note:
Clicking the link icon of a folder, automatically opens the referenced diagram.



5.3 Synchronizing Model and source code

UModel allows you to synchronize model and code from both sides.

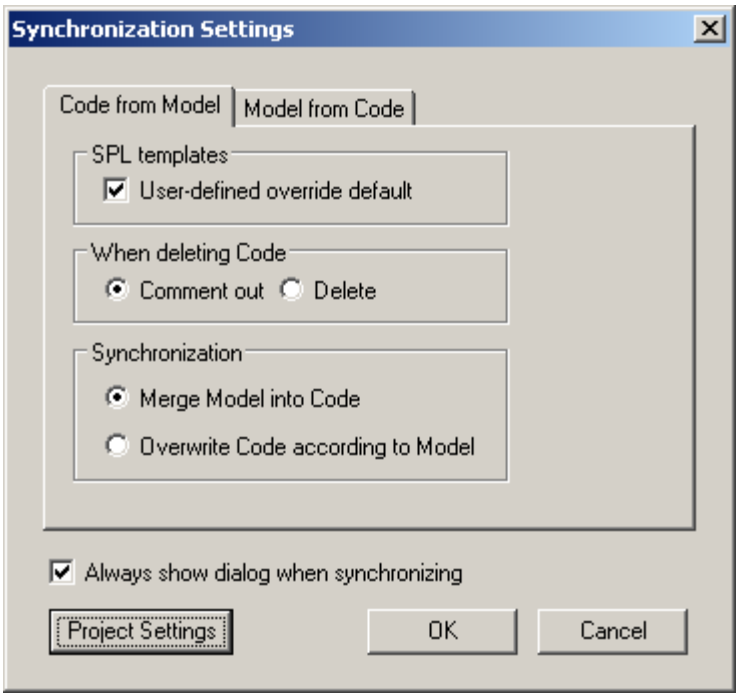
Code / model synchronization:
Code can be merged/synchronized at different levels described below. When using the context menu, e.g. when right clicking a class, the context menu reflects your selection in the menu option. Note that the Project menu only allows you to synchronize at the root/project level.

Project, Root package level:
1. Right click the Root package.
2. Select one of the code merging options: Merge Program..., or Merge UModel project... Alternatively, use the Project menu.

Package level:
1. Use SHIFT, or CTRL + click to select the package(s) you want to merge.
2. Right click the selection, and select one of the code merging options: Merge Program..., or Merge UModel project...

Class level:
1. Use SHIFT, or CTRL + click to select the classes(s) you want to merge.
2. Right click the selection, and select one of the code merging options: Merge Program..., or Merge UModel project...

Define your synchronization options by selecting:
1. **Project | Synchronization options**.
   Each tab allows you to define the specific merge settings.
2. Click the "Project Settings" button to select the specific programming language settings.
3. Define you specific settings and confirm with OK.



Please note:
When synchronizing code, you might be confronted with a dialog box that prompts you to update your UModel project before synchronization.

This only occurs if you are using UModel projects created before the latest release. Please click YES to update your project, and save your project file. This prompt will not occur once this has been done.

SPL Templates:
SPL templates are used during the generation of Java and C# code.

To modify the provided SPL templates:

1. Locate the provided SPL templates in the default directory: ...**UModel2007 \UModelSPL\Java\Default**. (or ...\C#\Default.)
2. Copy the SPL files you want to edit/modify into the **parent** directory, i.e. ...**UModel2007 \UModelSPL\Java**.
3. Make your changes and save them there.

To use the user-defined SPL templates:

1. Select the menu option **Project | Synchronization settings**.
2. Activate the "User-defined override default" checkbox in the SPL templates group.

Then select one of the menu options shown below, to initiate the synchronization process.

- **Project | Merge Program Code from UModel project**, please see Round-trip engineering (model - code - model) for more information, or
- **Project | Merge UModel Project from Project code**, please see Round-trip engineering (code - model - code) for more information.

5.4 Forward engineering prerequisites

Minimum conditions needed to produce code for forward engineering:

- A component must be **realized** by one or more **classes**, or **interfaces**.
- The component must have a **physical location**, i.e. directory, assigned to it. The generated code is then placed in this directory.
- Components must be individually set to be **included** in the code engineering process.
- The Java, or C#, namespace root package must be defined.

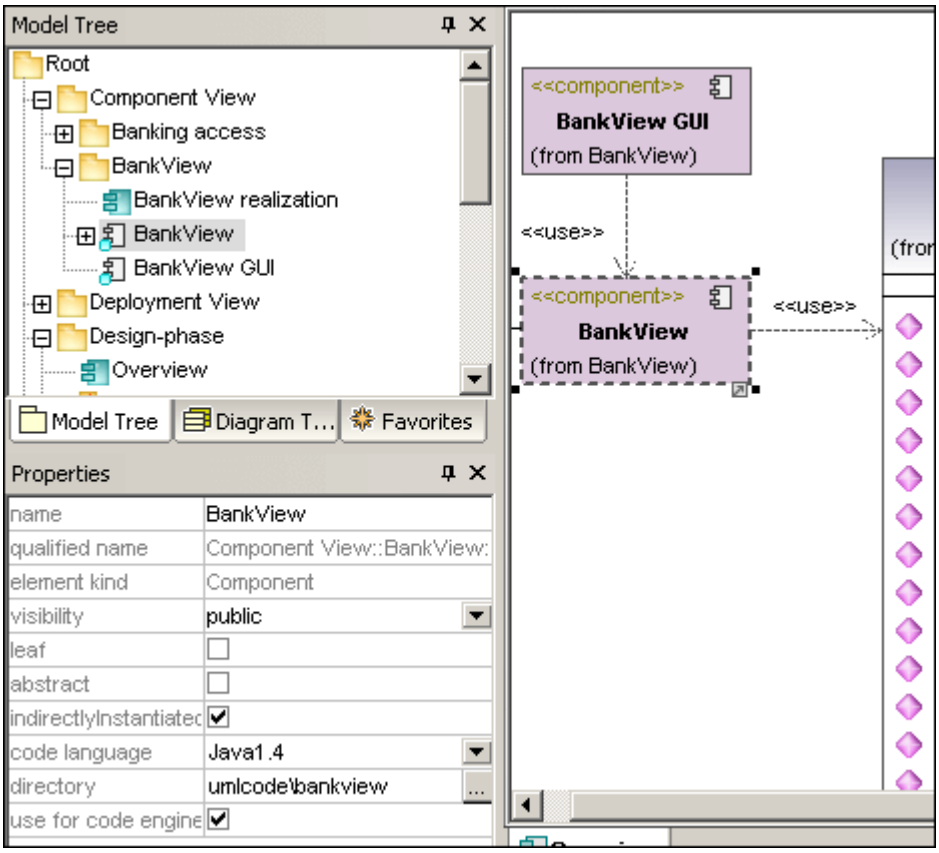
To create a component realization:

1. Drag the class, or interface onto the respective component in the Model Tree view.

   You can also create a realization in a component diagram using the Realization icon.

To assign a physical location:

1. Select the component in the Model Tree, or in the diagram.
2. Click the Browse button ... of the **directory** property and select a directory (or enter it directly).



To include components in the code engineering process:

1. Select the component in the Model Tree, or in the diagram.
2. Activate the "**use for code engineering**" check box.

To define the Java namespace root:

1. Right clicking a package and selecting "Set as Java namespace root" sets the Java namespace root.

   This means that this package and all sub packages, are enabled during the code

engineering process. The Java namespace root is denoted with a  icon in the Model Tree pane.

- Selecting the command again **removes** the Java namespace for this package.

5.5 Java code to/from UModel elements

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								
Java name				**UModel Element name**				
Project	project file			project file				Component
	directory			directory				
Package	name			name				Package <<namespace>>
Class	name			name				Class
	modifiers	package		visibility	package			
		public			public			
		protected			protected			
		private			private			
		abstract		abstract				
		strictfp		<<strictfp>>				
		final		<<final>>				
	file name			code file name				
	associated project file/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)				

Java ◁▷ UModel							
Java name				**UModel Element name**			
Class	Method	name		name		Operation	Class
		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			
		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			

Java ◁▷ UModel							
Java name				**UModel Element name**			
Interface	name			name			Interface
	modifiers	package		visibility	package		
		public			public		
		protected			protected		
		private			private		
		abstract		abstract			
		strictfp		<<strictfp>>			
	file name			code file name			
	associated project file/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			
			type	type			
			type dimensions	multiplicity			
		Type Parameter	name	name		Template Parameter	
			bound	constraining classifier			
	Type Parameter	name		name		Template Parameter	
		bound		constraining classifier			

Java ◁▷ UModel						
Java name			**UModel Element name**			
Enum	name		name			Enumeration
	modifiers	package	visibility	package		
		public		public		
		protected		protected		
		private		private		
	file name		code file name			
	associated project file/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	
		modifiers package	visibility	package		
		public		public		
		protected		protected		
		private		private		
		static	static			
		abstract	abstract			
		final	<<final>>			
		native	<<native>>			
		strictfp	<<strictfp>>			
		synchronized	<<synchronized>>			

Java ◁▷ UModel								
Java name				UModel Element name				
Enum	Method	throws clause		raised exceptions			Operation	Enumeration
		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				
		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				
Parameterized Type				Anonymous Bound Element				

5.6 C# code to/from UModel elements

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# ◁▷ UModel							
C#				UModel			
Project	project file			project file			Component
	directory			directory			
Name space	name			name			Package <<name space>>
Class	name			name			Class
	modifiers	internal		visibility	package		
		protected internal			protected <<internal>>		
		public			public		
		protected			protected		
		private			private		
		sealed		leaf			
		abstract		abstract			
		static		<<static>>			
		unsafe		<<unsafe>>			
		partial		<<partial>>			
		new		<<new>>			
	file name			code file name			
	associated project file/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)			

C# ◁▷ UModel										
C#							UModel			
Class	Constant	name					name		Property <<const>>	Class
		modifiers	internal				visibility	package		
			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>>			
			virtual				<<virtual>>			
			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			
			nullable				<<nullable>>			
		Type Parameter	name				name		Template Parameter	
			constraint				constraining classifier			
			predefined constraint	struct			<<ValueTypeConstraint>>			
				class			<<ReferenceTypeConstraint>>			
				new()			<<ConstructorConstraint>>			
			attribute sections				<<attributes>>			

C# ◁▷ UModel												
C#					UModel							
Class	Constructor	name			name				Operation	Class		
		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						
				params	varArgList							
			type		type							
			type dimensions		multiplicity							
			type pointer		type modifier							
			nullable		<<nullable>>							
	Destructor	name			name				Operation			
		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						
		type dimensions			multiplicity				Parameter			
		nullable			<<nullable>>							
		Get Accessor	modifiers	internal	visibility	package			<<GetAccessor>>			
				protected internal		protected internal						
				protected		protected						
				private		private						

C# ◁▷ UModel										
C#					UModel					
Class	Property	Set Accessor	modifiers	internal	visibility	package	<<SetAccessor>>	Operation <<property>>	Class	
				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					
			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>>					

C# ◁▷ UModel										
C#					UModel					
Class	Indexer	Get Accessor	modifiers	internal	visibility	package	<<Get Accessor>>	Operation <<indexer>>	Class	
				protected internal		protected internal				
				protected		protected				
				private		private				
		Set Accessor	modifiers	internal	visibility	package	<<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					
				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# ◁▷ UModel							
C#				UModel			
Class	name			name			Class
	modifiers	internal		visibility	package		
		protected internal			protected <<internal>>		
		public			public		
		protected			protected		
		private			private		
		unsafe		<<unsafe>>			
		partial		<<partial>>			
		new		<<new>>			
	file name			code file name			
	associated project file/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>>			
		type		type			
		type dimensions		multiplicity			

C# ◁▷ UModel									
C#					UModel				
Class	Constant	type pointer			type modifier			Property <<const >>	Class
		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				
			override		<<override>>				
			virtual		<<virtual>>				
			new		<<new>>				
			unsafe		<<unsafe>>				
		attribute sections			<<attributes>>				
		doc comments			Comment(->Documentation)				
		type			direction	return			
		Parameter	name		name		Parameter		
			modifiers	ref	direction	inout			
				out		out			
				params	varArgList				
			type		type				
			type dimensions		multiplicity				
			type pointer		type modifier				
			nullable		<<nullable>>				

C# ◁▷ UModel									
C#					UModel				
Struct	Method	Type Parameter	name		name	Template Parameter	Operation	Class <<struct>>	
			constraint		constraining classifier				
			predefined constraint	struct	<<ValueTypeConstraint>>				
				class	<<ReferenceTypeConstraint>>				
				new()	<<ConstructorConstraint>>				
			attribute sections		<<attributes>>				
	Constructor	name			name		Operation		
		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			
				params	varArgList				
			type		type				
			type dimensions		multiplicity				
			type pointer		type modifier				
			nullable		<<nullable>>				
	Destructor	name			name		Operation		
		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)				

C# ◁▷ UModel											
C#					UModel						
Struct	Property	type			direction	return	Parameter	Operation <<property>>	Class <<struct>>		
		type dimensions			multiplicity						
		nullable			<<nullable>>						
		Get Accessor	modifiers	internal	visibility	package	<<GetAccessor>>				
				protected internal		protected internal					
				protected		protected					
				private		private					
		Set Accessor	modifiers	internal	visibility	package	<<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						
			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>>						

C# ◁▷ UModel										
C#					UModel					
Struct	Indexer	Get Accessor	modifiers	internal	visibility	package	<<GetAccessor>>	Operation <<indexer>>	Class <<struct>>	
				protected internal		protected internal				
				protected		protected				
				private		private				
		Set Accessor	modifiers	internal	visibility	package	<<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					
			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 Remove Accessor			<<AddRemoveAccessor>>					
	Type Parameter	name			name			Template Parameter		
		constraint			constraining classifier					
		predefined constraint		struct	<<ValueTypeConstraint>>					
				class	<<ReferenceTypeConstraint>>					
				new()	<<ConstructorConstraint>>					
		attribute sections			<<attributes>>					
Interface	name				name				Interface	
	modifiers	internal			visibility	package				
		protected internal				protected <<internal>>				
		public				public				
		protected				protected				
		private				private				
		unsafe			<<unsafe>>					

C# ◁▷ UModel												
C#									UModel			
Interface		partial							<<partial>>			Interface
		new							<<new>>			
	file name								code file name			
	associated project file/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			
			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	package	<<GetAccessor>>	
				protected internal						protected internal		
				protected						protected		
				private						private		
		Set Accessor	modifiers	internal					visibility	package	<<SetAccessor>>	
				protected internal						protected internal		
				protected						protected		
				private						private		

C# ◁▷ UModel										
C#							UModel			
Interface	Indexer	name (="this")					name (="this")			Interface
		modifiers	public				visibility	public		
			new				<<new>>			
			unsafe				<<unsafe>>			
		attribute sections					<<attributes>>			
		doc comments					Comment(->Documentation)			
		type					direction	return	Parameter	Operation <<indexer>>
		Parameter	name				name			
			modifier	params			varArgList			
			type				type			
			type dimensions				multiplicity			
			type pointer				type modifier			
			nullable				<<nullable>>			
		Get Accessor	modifiers	internal			visibility	package	<<GetAccessor>>	
				protected internal				protected internal		
				protected				protected		
				private				private		
		Set Accessor	modifiers	internal			visibility	package	<<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>>			
		Remove Accessor								
	Type Parameter	name					name			Template Parameter
		constraint					constraining classifier			
		predefined constraint	struct				<<ValueTypeConstraint>>			
			class				<<ReferenceTypeConstraint>>			
			new()				<<ConstructorConstraint>>			
		attribute sections					<<attributes>>			
Delegate	name						name			Class <<delegate>>
	modifiers	internal					visibility	package		
		protected internal						protected <<internal>>		
		public						public		
		protected						protected		

C# ◁▷ UModel								
C#				**UModel**				
Delegate		private			private			Class <<delegate>>
		unsafe		<<unsafe>>				
		new		<<new>>				
	file name			code file name				
	associated project file/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				
		nullable		<<nullable>>				
	Type Parameter	name		name			Template Parameter	
		constraint		constraining classifier				
		predefined constraint	struct	<<ValueTypeConstraint>>				
			class	<<ReferenceTypeConstraint>>				
			new()	<<ConstructorConstraint>>				
		attribute sections		<<attributes>>				
Enum	name			name				Enum-eration
	modifiers	internal		visibility	package			
		protected internal			protected <<internal>>			
		public			public			
		protected			protected			
		private			private			
		new		<<new>>				
	file name			code file name				
	associated project file/directory			ComponentRealization				
	base type			type	<<BaseType>>			
	attribute sections			<<attributes>>				
	doc comments			Comment(->Documentation)				
	Enum Constant	name		name			Enumerat-ion Literal	
		default value		default				
		attribute sections		<<attributes>>				
		doc comments		Comment(->Documentation)				
Parameterized Type				Anonymous Bound Element				

5.7 XML Schema to/from UModel elements

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

XSD/UML Element

Stereotype property (=tagged value)

XSD ◁▷ Umodel								
XSD				UModel				
file path				project file				Component
schema	target namespace			name				Package <<name space>>
	attributeFormDefault			attributeFormDefault				
	blockDefault			blockDefault				
	elementFormDefault			elementFormDefault				
	finalDefault			finalDefault				
	version			version				
	xml:lang			xml:lang				
	xmlns			xmlns				
	annotation	source		source				
		appinfo					Comment <<appinfo>>	
		documentation	xml:lang	xml:lang			Comment <<documentation>>	
	attributeGroup	name		name			Class <<attributeGroup>>	Class <<schema>>
		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				

XSD ◁▷ Umodel								
XSD					UModel			
schema	attribute	name			name		Class <<attribute>>	Class << schema >>
		form			form			
		use			use			
		type			type	Property		
		default			default			
		fixed				fixed		
		annotation	appinfo			Comment <<appinfo>>		
			documentation			Comment <<document-ation>>		
		simpleType			name (= name of Class + "_anonymousType[n]")	DataType <<simpleType>>		
	element	name			name		Class <<element>>	
		abstract			abstract			
		block			block			
		final			final			
		nillable			nillable			
		type			type	Property		
		default			default			
		fixed				fixed		
		annotation	appinfo			Comment <<appinfo>>		
			documentation			Comment <<document-ation>>		
		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 <<document-ation>>		

<table>
<tr><th colspan="10">XSD ◁▷ Umodel</th></tr>
<tr><th colspan="6">XSD</th><th colspan="4">UModel</th></tr>
<tr><td rowspan="16">schema</td><td rowspan="16">group</td><td rowspan="8">all</td><td colspan="2" rowspan="2"></td><td colspan="2">name (= "_all")</td><td>Property</td><td rowspan="8">Class <<all>></td><td rowspan="16">Class <<group>></td><td rowspan="16">Class <<schema>></td></tr>
<tr><td colspan="2">name (= "mg"_ + "all")</td></tr>
<tr><td rowspan="2">annotation</td><td>appinfo</td><td></td><td>Comment <<appinfo>></td></tr>
<tr><td>documentation</td><td></td><td>Comment <<documentation>></td></tr>
<tr><td rowspan="3">element</td><td>name</td><td>name</td><td rowspan="3">Property <<element>></td></tr>
<tr><td>ref</td><td rowspan="2">type</td></tr>
<tr><td>type</td></tr>
<tr><td rowspan="8">choice</td><td colspan="2" rowspan="2"></td><td colspan="2">name (= "_choice")</td><td>Property</td><td rowspan="8">Class <<choice>></td></tr>
<tr><td colspan="2">name (= "mg"_ + "choice")</td></tr>
<tr><td rowspan="2">annotation</td><td>appinfo</td><td></td><td>Comment <<appinfo>></td></tr>
<tr><td>documentation</td><td></td><td>Comment <<documentation>></td></tr>
<tr><td rowspan="3">element</td><td>name</td><td>name</td><td rowspan="3">Property <<element>></td></tr>
<tr><td>ref</td><td rowspan="2">type</td></tr>
<tr><td>type</td></tr>
<tr><td>group</td><td></td><td></td><td>Property <<group>></td></tr>
<tr><td>any</td><td></td><td></td><td>Property <<any>></td></tr>
<tr><td rowspan="2">choice</td><td rowspan="2"></td><td rowspan="2"></td><td>Property</td></tr>
<tr><td>Class <<choice>></td></tr>
<tr><td rowspan="2">sequence</td><td rowspan="2"></td><td rowspan="2"></td><td>Property</td></tr>
<tr><td>Class <<sequence>></td></tr>
</table>

<table>
<tr><th colspan="10">XSD ◁▷ Umodel</th></tr>
<tr><th colspan="6">XSD</th><th colspan="4">UModel</th></tr>
<tr><td rowspan="13">schema</td><td rowspan="12">group</td><td rowspan="12">sequence</td><td colspan="2" rowspan="2"></td><td colspan="2">name (= "_sequence")</td><td>Property</td><td rowspan="12">Class <<sequence>></td><td rowspan="12">Class <<group>></td><td rowspan="13">Class <<schema>></td></tr>
<tr><td colspan="2">name (= "mg"_ + "sequence")</td></tr>
<tr><td rowspan="2">annotation</td><td>appinfo</td><td></td><td>Comment <<appinfo>></td></tr>
<tr><td>documentation</td><td></td><td>Comment <<documentation>></td></tr>
<tr><td rowspan="3">element</td><td>name</td><td>name</td><td rowspan="3">Property <<element>></td></tr>
<tr><td>ref</td><td rowspan="2">type</td></tr>
<tr><td>type</td></tr>
<tr><td>group</td><td></td><td></td><td>Property <<group>></td></tr>
<tr><td>any</td><td></td><td></td><td>Property <<any>></td></tr>
<tr><td rowspan="2">choice</td><td rowspan="2"></td><td rowspan="2"></td><td>Property</td></tr>
<tr><td>Class <<choice>></td></tr>
<tr><td rowspan="2">sequence</td><td rowspan="2"></td><td rowspan="2"></td><td>Property</td></tr>
<tr><td>Class <<sequence>></td></tr>
<tr><td rowspan="5">notation</td><td colspan="3">name</td><td colspan="3">name</td><td rowspan="5">DataType <<notation>></td></tr>
<tr><td colspan="3">system</td><td colspan="3">system</td></tr>
<tr><td colspan="3">public</td><td colspan="3">public</td></tr>
<tr><td rowspan="2">annotation</td><td>appinfo</td><td></td><td></td><td>Comment <<appinfo>></td></tr>
<tr><td>documentation</td><td></td><td></td><td>Comment <<documentation>></td></tr>
</table>

<table>
<tr><th colspan="8">XSD ◁▷ Umodel</th></tr>
<tr><th colspan="5">XSD</th><th colspan="3">UModel</th></tr>
<tr><td rowspan="15">schema</td><td rowspan="15">complexType</td><td colspan="2">name</td><td>name</td><td rowspan="15">Class
<<complexType>></td><td rowspan="15">Class
<<schema>></td></tr>
<tr><td colspan="2">abstract</td><td>abstract</td></tr>
<tr><td colspan="2">block</td><td>block</td></tr>
<tr><td colspan="2">final</td><td>final</td></tr>
<tr><td colspan="2">mixed</td><td>mixed</td></tr>
<tr><td rowspan="3">annotation</td><td colspan="2">source</td><td>source</td></tr>
<tr><td>appinfo</td><td></td><td></td><td>Comment
<<appinfo>></td></tr>
<tr><td>documentation</td><td>xml:lang</td><td>xml:lang</td><td>Comment
<<documentation>></td></tr>
<tr><td rowspan="5">group</td><td colspan="2"></td><td>name (= "_ref[n]")</td><td rowspan="4">Property
<<group>></td></tr>
<tr><td>maxOccurs</td><td rowspan="2">multiplicity</td></tr>
<tr><td>minOccurs</td></tr>
<tr><td>ref</td><td>type</td></tr>
<tr><td colspan="2"></td><td>name (= "mg"_ + "all")</td><td>Class <<all>></td></tr>
<tr><td rowspan="3">all</td><td colspan="2"></td><td>name (= "_all")</td><td rowspan="3">Property</td></tr>
<tr><td>maxOccurs</td><td rowspan="2">multiplicity</td></tr>
<tr><td>minOccurs</td></tr>
<tr><td rowspan="3">choice</td><td colspan="2" rowspan="2"></td><td>name (= "mg"_ + "choice[n]")</td><td>Class
<<choice>></td></tr>
<tr><td>name (= "_choice[n]")</td><td rowspan="3">Property</td></tr>
<tr><td>maxOccurs</td><td rowspan="2">multiplicity</td></tr>
<tr><td>minOccurs</td></tr>
<tr><td rowspan="3">sequence</td><td colspan="2" rowspan="2"></td><td>name (= "mg"_ + "sequence[n]")</td><td>Class
<<sequence>></td></tr>
<tr><td>name (= "_sequence[n]")</td><td rowspan="3">Property</td></tr>
<tr><td>maxOccurs</td><td rowspan="2">multiplicity</td></tr>
<tr><td>minOccurs</td></tr>
</table>

<table>
<tr><th colspan="8">XSD ◁|▷ Umodel</th></tr>
<tr><th colspan="5">XSD</th><th colspan="3">UModel</th></tr>
<tr><td rowspan="11">schema</td><td rowspan="6">complexType</td><td rowspan="3">attribute</td><td>name</td><td></td><td>name</td><td rowspan="3">Property
<<attribute>></td><td rowspan="6">Class
<<complexType>></td><td rowspan="11">Class
<<schema>></td></tr>
<tr><td>ref</td><td></td><td rowspan="2">type</td></tr>
<tr><td>type</td><td></td></tr>
<tr><td>attributeGroup</td><td>ref</td><td></td><td>type</td><td>Property
<<attributeGroup>></td></tr>
<tr><td rowspan="2">anyAttribute</td><td>namespace</td><td></td><td>namespace</td><td rowspan="2">Property
<<anyAttribute>></td></tr>
<tr><td>processContents</td><td></td><td>processContents</td></tr>
<tr><td rowspan="2">complexContent</td><td>restriction</td><td rowspan="2">base</td><td rowspan="2">general</td><td>Generalization
<<restriction>></td></tr>
<tr><td>extension</td><td>Generalization
<<extension>></td></tr>
<tr><td rowspan="5">simpleType</td><td colspan="2">name</td><td></td><td>name</td><td colspan="2" rowspan="5">DataType
<<simpleType>>
Enumeration
<<simpleType>></td></tr>
<tr><td colspan="2">final</td><td></td><td>final</td></tr>
<tr><td rowspan="3">annotation</td><td>source</td><td></td><td>source</td></tr>
<tr><td>appinfo</td><td></td><td></td><td>Comment
<<appinfo>></td></tr>
<tr><td>documentation</td><td>xml:lang</td><td>xml:lang</td><td>Comment
<<documentation>></td></tr>
<tr><td rowspan="2">list</td><td colspan="2">itemType</td><td>name (= "_itemType")</td><td>Property
<<itemType>></td><td rowspan="2"><<list>></td></tr>
<tr><td colspan="2">simpleType</td><td colspan="2">DataType
<<simpleType>></td></tr>
<tr><td rowspan="2">union</td><td colspan="2">memberTypes</td><td>name (= "memberType[n]")</td><td>Property
<<memberType>></td><td rowspan="2"><<union>></td></tr>
<tr><td colspan="2">simpleType</td><td colspan="2">DataType
<<simpleType>></td></tr>
<tr><td>minExclusive</td><td colspan="2">value
fixed</td><td colspan="2">value
fixed</td><td><<minExclusive>></td></tr>
<tr><td>minInclusive</td><td colspan="2">value
fixed</td><td colspan="2">value
fixed</td><td><<minInclusive>></td></tr>
</table>

XSD ◁▷ Umodel								
XSD					UModel			
schema	simpleType	maxExclusive	value		value	<<maxExclusive>>	DataType <<simpleType>> Enumeration <<simpleType>>	Class <<schema>>
			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>>		
		minExclusive	value		value	<<minExclusive>>		
			fixed		fixed			

XSD ◁▷ Umodel							
XSD				UModel			
schema	complexType simpleContent	minInclusive	value	value	<<minInclusive>>	DataType <<complexType>> <<simpleContent>>	Class << schema >>
			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>>		
			processContents	processContents			
		simpleType			DataType <<simpleType>>		
		restriction	base	general	Generalization <<restriction>>		
		extension	base	general	Generalization <<extension>>		

XSD ◁▷ Umodel								
XSD					UModel			
schema	import	schemaLocation			schemaLocation		ElementImport <<import>>	Class <<schema>>
		namespace			namespace			
		annotation	appinfo			Comment <<appinfo>>		
			documentation			Comment <<documentation>>		
	include	schemaLocation			schemaLocation		ElementImport <<include>>	
		annotation	appinfo			Comment <<appinfo>>		
			documentation			Comment <<documentation>>		
	redefine	schemaLocation			schemaLocation		ElementImport <<include>>	
		annotation	appinfo			Comment <<appinfo>>		
			documentation			Comment <<documentation>>		
		simpleType			<<redefine>>		DataType <<simpleType>>	
		complexType					Class <<complexType>>	
		attributeGroup					Class <<attributeGroup>>	
		group					Class <<group>>	

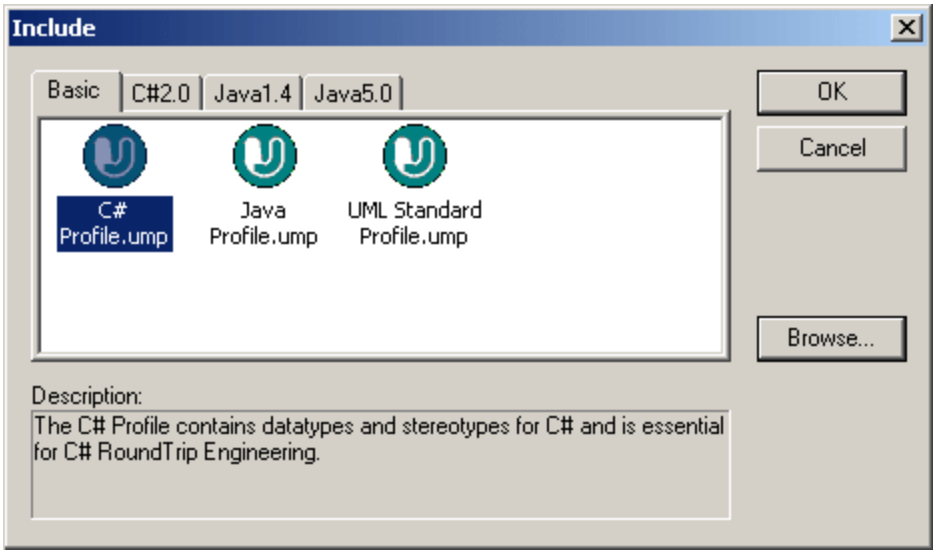
5.8 Including other UModel projects

UModel is supplied with several files that can be included in a UModel project. Clicking one of the Java tabs allows you to include Java lang classes, interfaces and packages in your project, by selecting one of the supplied files.

1. Select **Project | Include Subproject** to open the "Include" dialog box.
2. Click the UModel project file you want to include, and press OK.

UModel projects can be included within other UModel projects. To include projects place the respective *.ump files in:

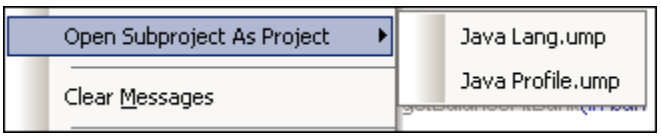
- ...\UModel2007**UModelInclude** to appear in the Basic tab, or
- ...\UModel2007**UModelInclude\Java1.4 / Java5.0** to appear in the Java tab.



Please note:
An include file, which contains all types of the Microsoft .NET Framework 2.0, is available in the C# 2.0 tab.

To view all currently imported projects:

- Select the menu option **Project | Open Subproject as project**.
  The flyout menu displays the currently included subprojects.



To create a user-defined tab/folder:

1. Navigate to the ...**UModel2007\UModelInclude** and create/add your folder below ...\UModelInclude, i.e. ...**UModelInclude\myfolder.**

To create descriptive text for each UModel project file:

1. Create a text file using the same name as the *.ump file and place in the same folder. Eg. 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 Del. key.
2. You are prompted if you want to continue the deletion process.
3. Click OK to delete the included file from the project.

Please note:

- To delete or remove a project from the "Include" dialog box, delete or remove the (MyModel).ump file from the respective folder.

5.9 Sharing Packages and Diagrams

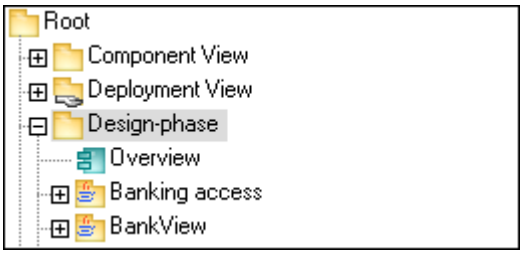
UModel allows you to share packages and UML diagrams they might contain, between different projects. Packages can be included in other UModel projects by reference, or as a copy.

Shared package prerequisites:

- Links to other packages outside of the shared scope are not permissible.

To share a package between projects:

1. Right click a package in the Model Tree tab and select **Subproject | Share package**.

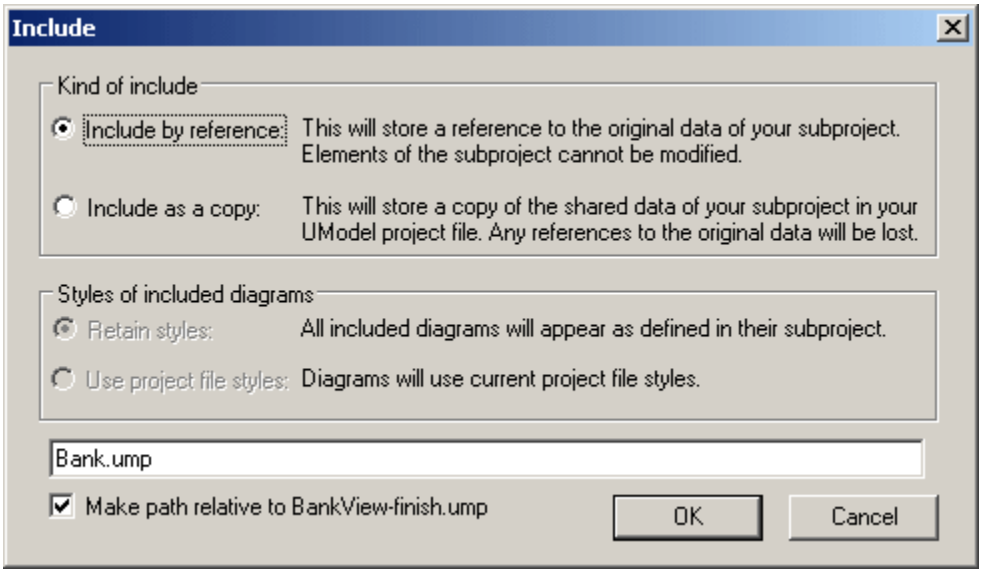


A "shared" icon appears below the shared package in the Model Tree. This package can now be included in any other UModel project.

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).

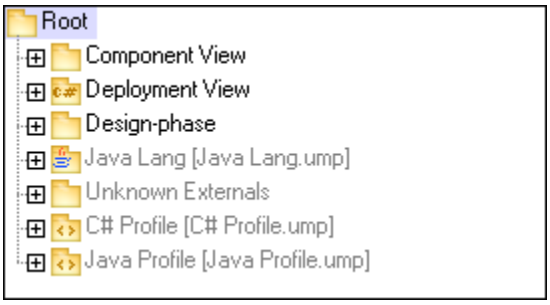
2. Select the menu item **Project | Include Subproject...**
3. Click the Browse button, select the project that contains the shared package and click Open.



The "Include" dialog box allows you to choose between including the package/project

by reference, or as a copy.

4. Select the specific option (Include by reference) and click OK.



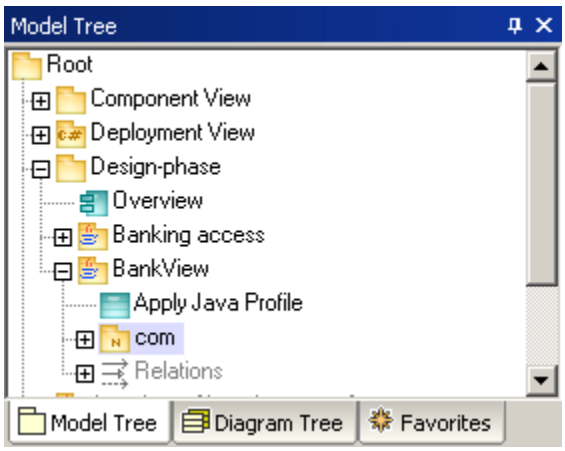
The "Deployment View" package is now visible in the new package. The packages' source project is displayed in parenthesis (BankView-start.ump).

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**.

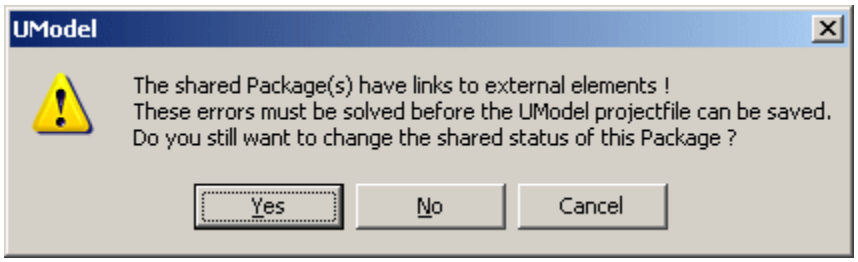
Please note:

All included projects of the source project, have also been included: Java Lang, Unknown Externals and Java Profile.

Shared packages - links to external elements:

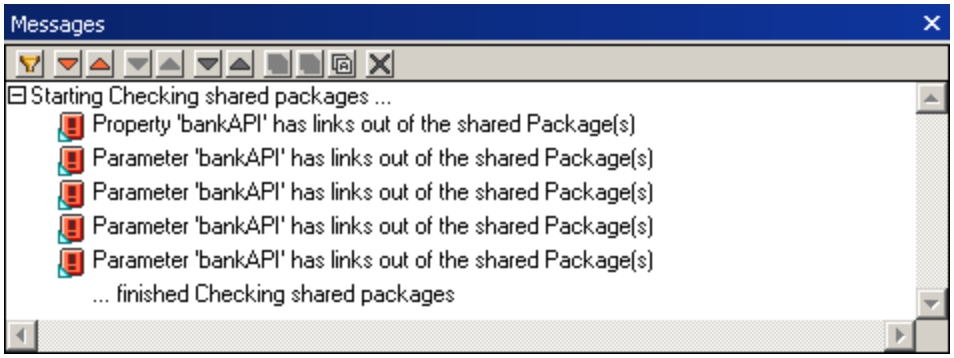


Attempting to share a package which has links to external elements causes a prompt to appear. E.g. trying to share the BankView package.

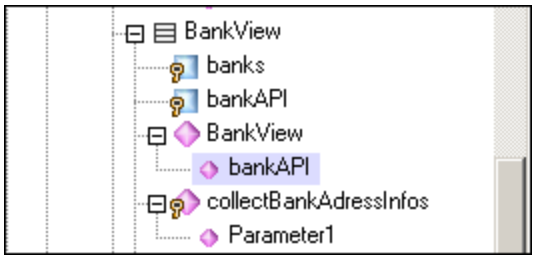


Clicking Yes, forces you to resolve the external links before you can save.

The Messages pane provides information on each of the external links.



Clicking an error entry, in the Messages pane, displays the relevant element in the Model Tree tab.

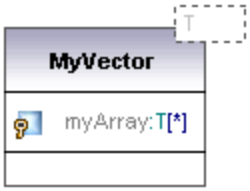


5.10 UML templates

UModel now supports the use of UML templates and their mapping to/from Java 5.0 and C# 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.

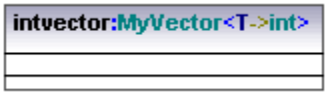
Template **signature** display in UModel:



- 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.

5.10.1 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, datatype), 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.

Extending template parameters

T>Comparable<T->T>

5.10.2 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>

5.10.3 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->?>

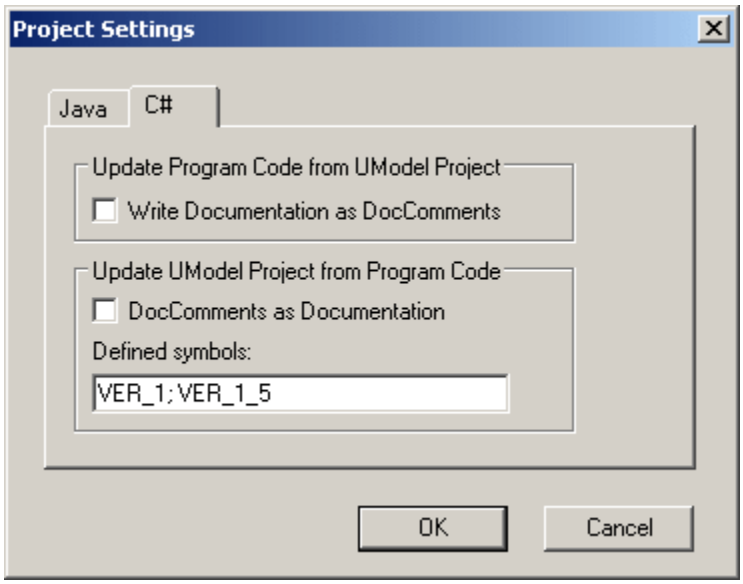
This class contains a generic vector of unspecified type (? is the wildcard).

Typed properties can be displayed as associations:

- Right click a property and select **Show | PropertyX as Association**, or
- Drag a property onto the diagram background.

5.11 Project Settings

This option allows you to define the global project settings.



Select the menu item **Tools | Options** to define your local settings, please see **Tools | Options** in the Reference section for more details on the local settings.

5.12 Enhancing performance

Due to the fact that some modeling projects can become quite large, 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 - **Styles tab | Use Syntax Coloring** = false.
- Disable "gradient" as a background color for diagrams, use a solid color. E.g. **Styles tab | Diagram background color | White.**

Chapter 6

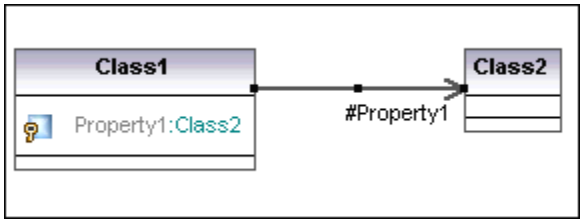
Creating model relationships

6 Creating model relationships

Model relationships can be created and inserted into diagrams using several methods:

- By clicking the aggregation  , or composition  icons in the icon bar.
- By using the connection handles, please see Use cases for an example.
- By clicking the association icon  in the icon bar, and creating a connection between elements using drag and drop

When an association has been created, a new attribute is automatically inserted in the originating (A:name) class, e.g. Property1:Class2, in the example below.



Having created the association it is shown as active, and the Properties tab displays its properties.

Clicking an association line, displays the association properties in the Properties tab. A:Name and B:Name indicate the role of each class in the other.

Depending on the "memberEndKind" - **property** (of A:name "Property1"):
the **attribute** either belongs to:

- the **class** - i.e. **A**:memberEndKind = **memberEnd**, (attribute is visible in class1), or
- the **association** - i.e. **B**:memberEndKind = **ownedEnd** (attribute not visible in class2).

  If both attributes belongs to the **association**, i.e. both ends are defined as "ownedEnd, then this association becomes bi-directional, and the navigability arrow disappears. Both ends of the association are "ownedEnd".

If the memberEndKind of any of the association is set to "**navigableOwnedEnd**, then the

attribute is still part of the association, but the navigability arrow reappears depending on which end (A:name or B:Name) it is set.

To define the type of association (association, aggregate, or composite)

1. Click the association arrow.
2. Scroll down to the **aggregation** item in the Properties tab.
3. Select: none, shared or composite.

None: a standard association
shared: an **aggregate** association
composite: a **composite** association.

Please note:

Associations can be created using the same class as both the source and target. This is a so-called self link. It describes the ability of an object to send a message to itself, for recursive calls.

Click the relationship icon, then drag from the element, dropping somewhere else on the same element. A self-link appears.

Displaying associations in Diagrams automatically

When inserting diagram elements in a diagram, the "Automatically create Associations" option in the **Tools | Options | Editing** tab, allows existing associations between modeling elements to be automatically created/displayed in the current diagram. This occurs if the attributes type is set, and the referenced "type" modeling element is in the current diagram.

Deleting relationships/associations:

1. Click the relationship in the diagram tab, or in the Model Tree.
2. Press the **Del.** keyboard key.
   The dependency is deleted from the diagram and project.

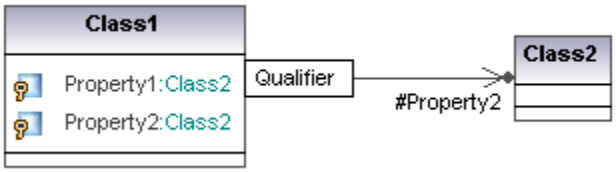
Deleting class associations:

Deleting a **class** association does not delete the **attribute/property** that was automatically generated, from the class!

1. Right click the attribute/property in the class.
2. Select the option "**Delete PropertyX**" from "ClassX" to delete it.

Creating association qualifiers:

1. Having defined an association between two classes
2. Right click the association line and select **New | Qualifier**.



Please note that qualifiers are attributes of an association.

6.1 Associations, realizations and dependencies

Creating relationships using connection handles:

1. Given two classes in the class diagram,
2. Click the first class to make it the active class.
   Connection handles appear on three sides.
3. Move the mouse pointer over the handle on the right border of the class.
   A Tooltip appears, informing you of the type of relationship that this handle creates, Association in this case.
4. Drag to create a connector, and drop it on the second class. The target class is highlighted if this type of association is possible.
   An association has now been created between these two classes.

Elements in the various model diagrams supply you with different connection handles. E.g. a class in a class diagram supplies the following relationship handles (in clockwise fashion):

- InterfaceRealization
- Generalization
- Association

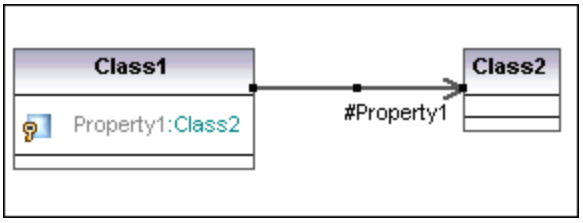
An Artifact in the Deployment view supplies the following handles:

- Manifestation
- Association
- Deployment

Creating relationships using icons in the icon bar:
Given two elements in a modeling diagram,

1. Click the icon that represents the relationship you want to create e.g. association, aggregation, or composition.
2. Drag from the one object to the other, and drop when the target element is highlighted.

   When creating a new association, a new attribute is automatically inserted in the originating (A:name) class, Property1:Class2, in the example below.



   UModel always shows all attributes of a class!

   Please note:
   The screenshots in this manual do not show the Association **Ownership dot**.

   To enable it, set the **Show Assoc. Ownership**, in the Styles tab, to true.

Deleting relationships/associations:

1. Click the relationship in the diagram tab, or in the Model Tree.

2. Press the **Del.** keyboard key.
   The dependency is deleted from the diagram and project.

Deleting class associations:

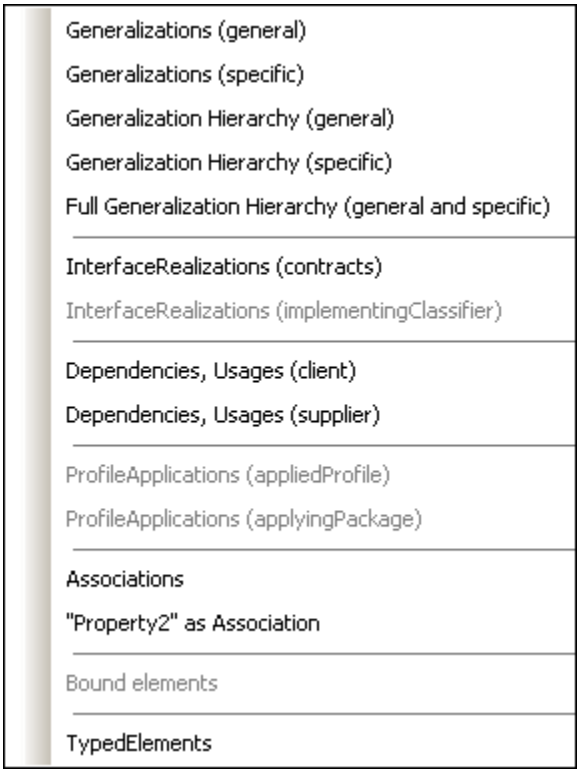
Deleting a **class** association does not delete the **attribute/property** that was automatically generated, from the class!

1. Right click the attribute/property in the class.
2. Select the option "**Delete PropertyX**" from "ClassX" to delete it.

6.2 Showing model relationships

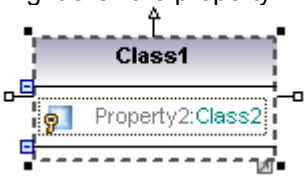
Showing relationships between modeling elements:

1. Right click the specific element and select **Show**.
   The popup menu shown below is context specific, meaning that only those options are available that are relevant to the specific element.

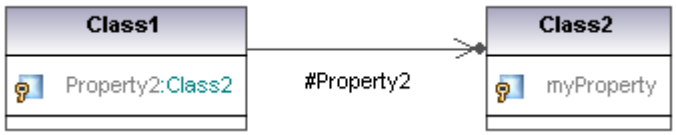


To show a class attribute/property as an association:

1. Right click the property in the class.



2. Select the menu option **Show | "PropertyXX" as Association**.
   This inserts/opens the referenced class and shows the relevant association.



Chapter 7

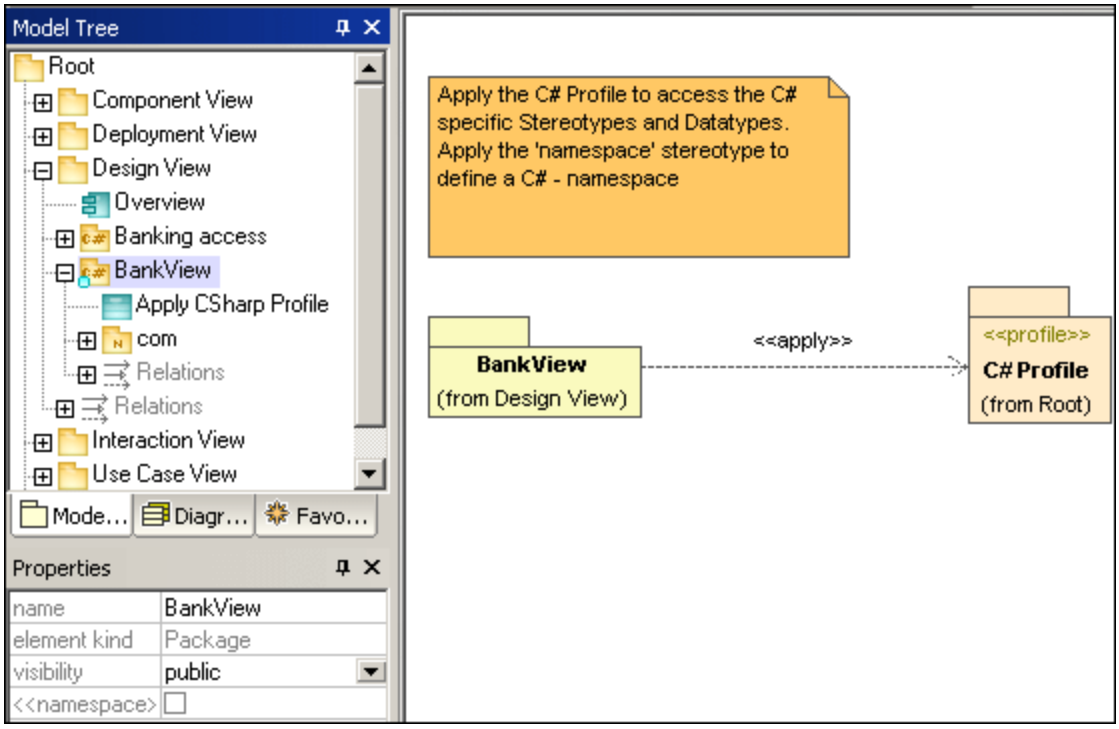
Profiles and stereotypes

7 Profiles and stereotypes

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.

The Java Profile.ump (or C# Profile.ump) file needs to be applied when creating new UModel projects using the menu item **Project | Include Subproject.** This profile supplies the Java datatypes and stereotypes, and is essential when creating code for round-trip engineering.

The **Bank_CSharp.ump** sample file (in the ...**UModelExamples** folder) shows how this is done. The C# profile has been applied to the BankView package.



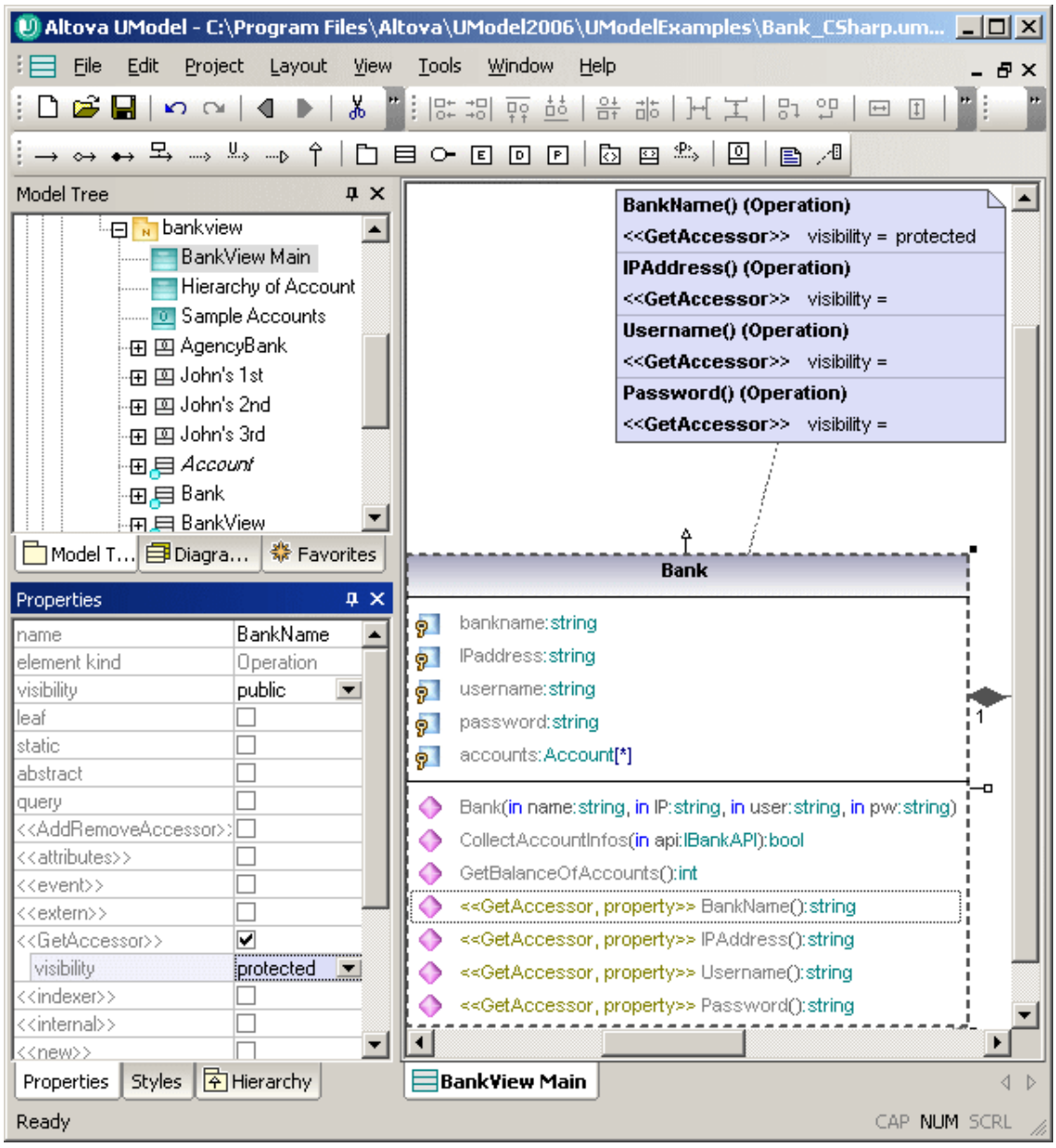
- Profiles are specific types of packages, that are applied to other packages.
- Stereotypes are specific metaclasses, that extend standard classes.
- "Tagged values" are values of stereotype attributes.

A Profile Application shows which profiles have been applied to a package, and 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.

Stereotypes:
A stereotype defines how an existing metaclass may be extended. It is a kind of class that extends Classes through Extensions. Stereotypes can only be created in Profiles. Stereotypes are displayed as classes, in class diagrams, with the addition of the keyword <<stereotype>> added above the name of the class.

- Stereotypes may have **properties**, which are called "**tag definitions**"
- When the stereotype is applied to a model element, the property **values** are called "**tagged values**"
- When stereotypes containing properties are applied, the tagged values are automatically displayed in a comment element (shown below). Please see Tagged values for more info on how to customize the tagged values view
- If the attribute is of type "enumeration", then an popup menu allows you to select from the predefined values. You can also enter/select the specific value in the Properties tab e.g. <<GetAccessor>> visibility = public, protected etc.

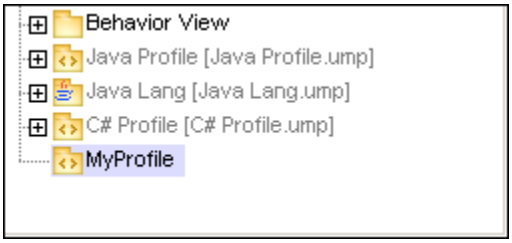


7.1 Adding Stereotypes and defining tagged values

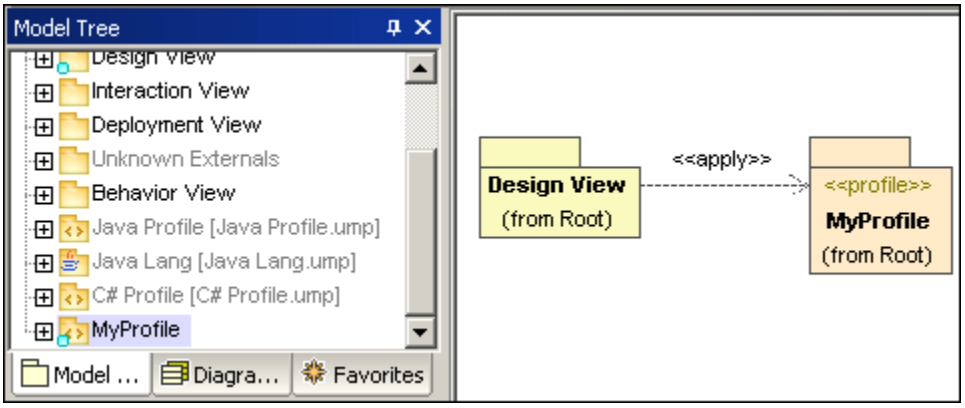
This section uses the Bank_MultiLanguage.ump file available in the ...**UModelExamples** folder.

Creating a stereotype and defining its attributes

1. Create a new profile in the Model Tree view, e.g. right click the Root package and select **New | Profile** and name it "MyProfile".

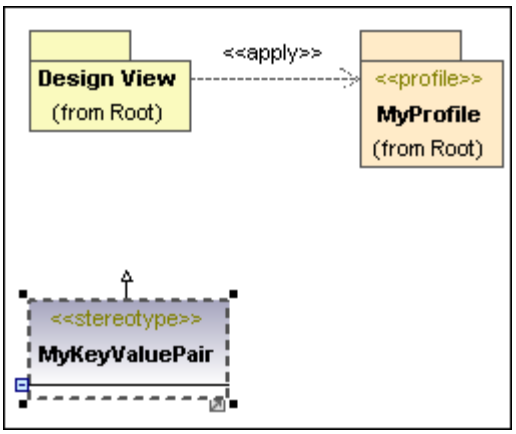


2. Right click MyProfile and select **New Diagram | Class Diagram**.
3. Drag the newly created profile "MyProfile", from the Model Tree into the new class diagram.
4. Drag the **DesignView** package into the new class diagram as well.
5. Click the ProfileApplication icon  in the icon bar, select the DesignView package and drag the connector onto the MyProfile package.

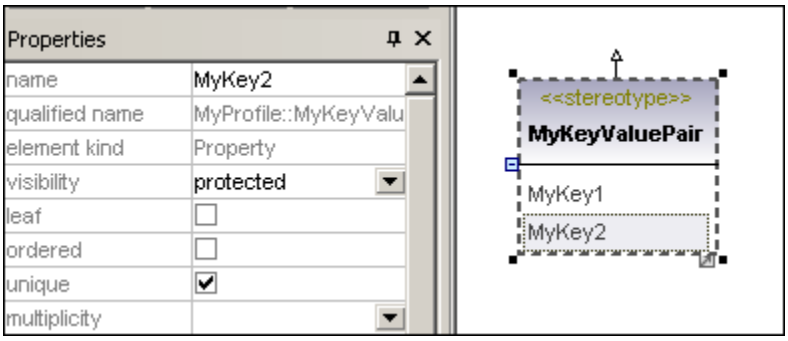


This allows the stereotypes defined in this profile (MyProfile) to be used in the DesignView package, or any of its subpackages.

6. Click the stereotype icon  in the icon bar and insert a stereotype "class".



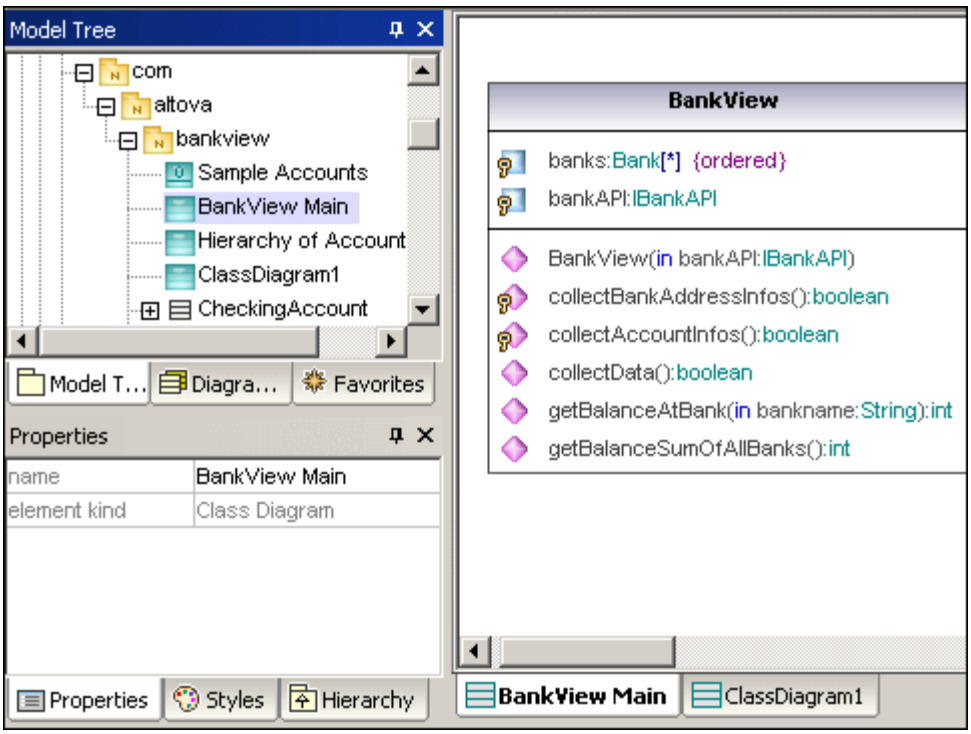
7. Press F7 to add an attribute to the stereotype e.g. **MyKey1**. Do the same thing to add MyKey2.



This concludes the definition of the stereotype for the moment. We can now use/assign the stereotype when adding an attribute to a class which is part of the BankView package.

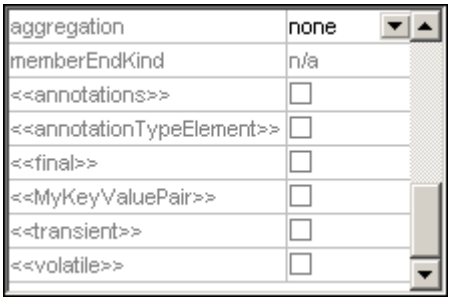
Using / assigning stereotypes

1. Double click the BankView Main class diagram icon in the Model Tree.

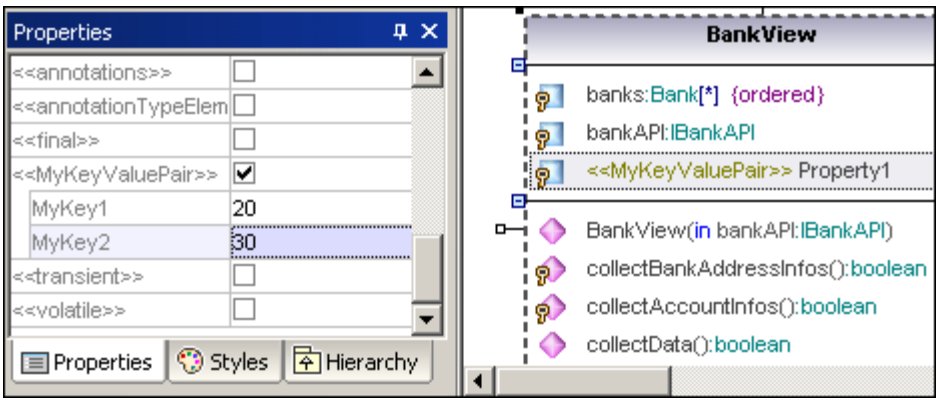


This opens the class diagram and displays the associations between the various classes. We now want to add an attribute to the BankView class, and assign/use the previously defined stereotype.

2. Click the **BankView** class and press F7 to add an attribute.
3. Use the scrollbar of the Properties tab to scroll to the bottom of the list. Notice that the **MyKeyValuePair** stereotype is available in the list box.

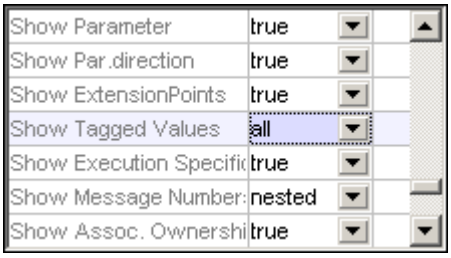


4. Click the **MyKeyValuePair** check box to activate/apply it. The two tagged values MyKey1 and MyKey2, are now shown under the Stereotype entry.
5. Double click in the respective fields and enter some values.

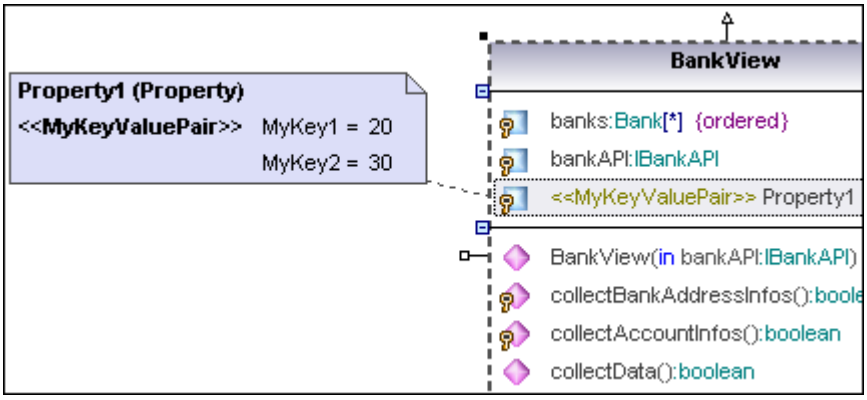


Displaying tagged values in a diagram

1. Click the Styles tab, scroll down to the **Show Tagged Values** entry and select **all**.



The diagram tab now displays the tagged values in the note element. Double clicking a value in the note element allows you to edit it directly.

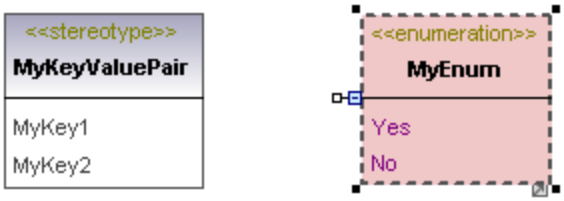


Stereotypes and enumerations

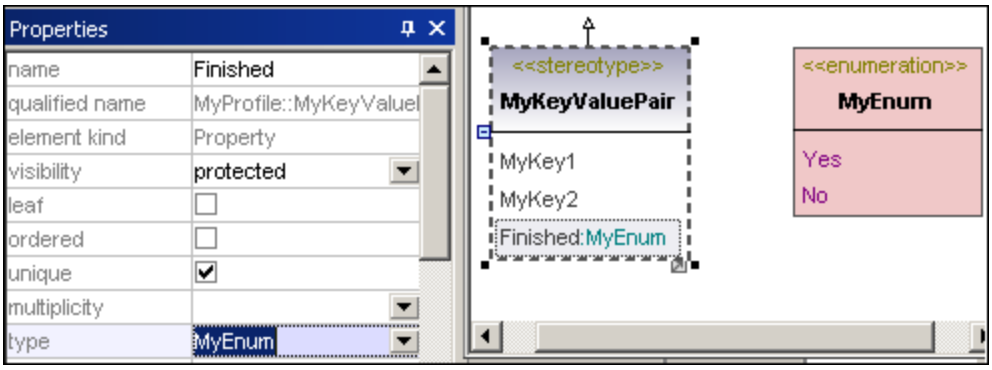
UModel has an efficient method of selecting enumerated values of stereotypes.

Click the diagram tab containing the stereotype definition:

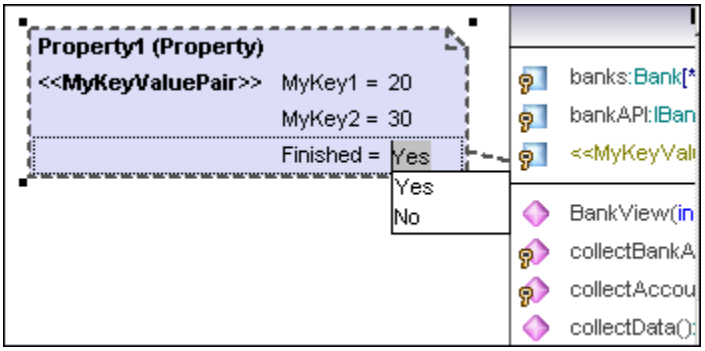
1. Click the Enumeration icon  in the icon bar to insert an enumeration in the class diagram (containing the previously defined stereotype).
2. Add EnumerationLiterals to the enumeration by pressing SHIFT+F7, or use the context menu, e.g. **Yes**, **No**.



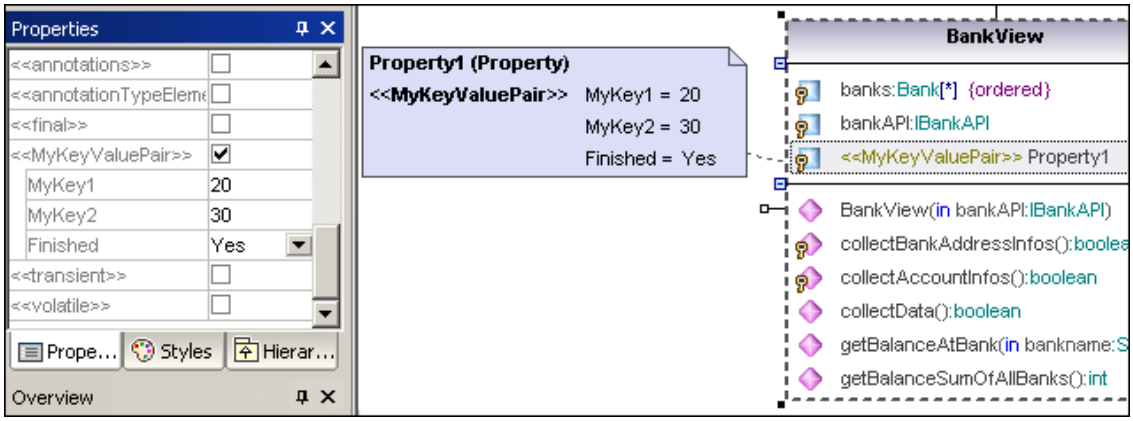
3. Click the stereotype "class" and press F7 to add a new attribute/property, e.g. **Finished**.
4. Select type "**My Enum**" from the Properties tab.



5. Switch back to the **BankView Main** class diagram.
6. Property Finished, is now shown as a tagged value in the note element.



Double clicking the Finished tagged value, presents the predefined enumeration values in a popup. Click one of the enumerations to select it.



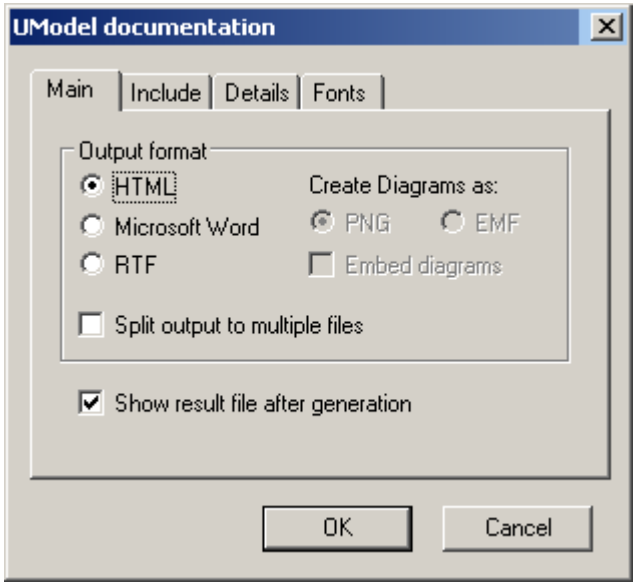
Chapter 8

Generating UML documentation

8 Generating UML documentation

The **Project | Generate Documentation...** command generates detailed documentation about your UML project in HTML, Microsoft Word, or RTF formats. **Note**: In order to generate documentation in MS Word format, you must have MS Word (version 2000 or later) installed.

Note that you can also create **partial documentation** of modeling elements by right clicking an element in the Model Tree and selecting "Generate Documentation". The documentation options are the same in both cases.

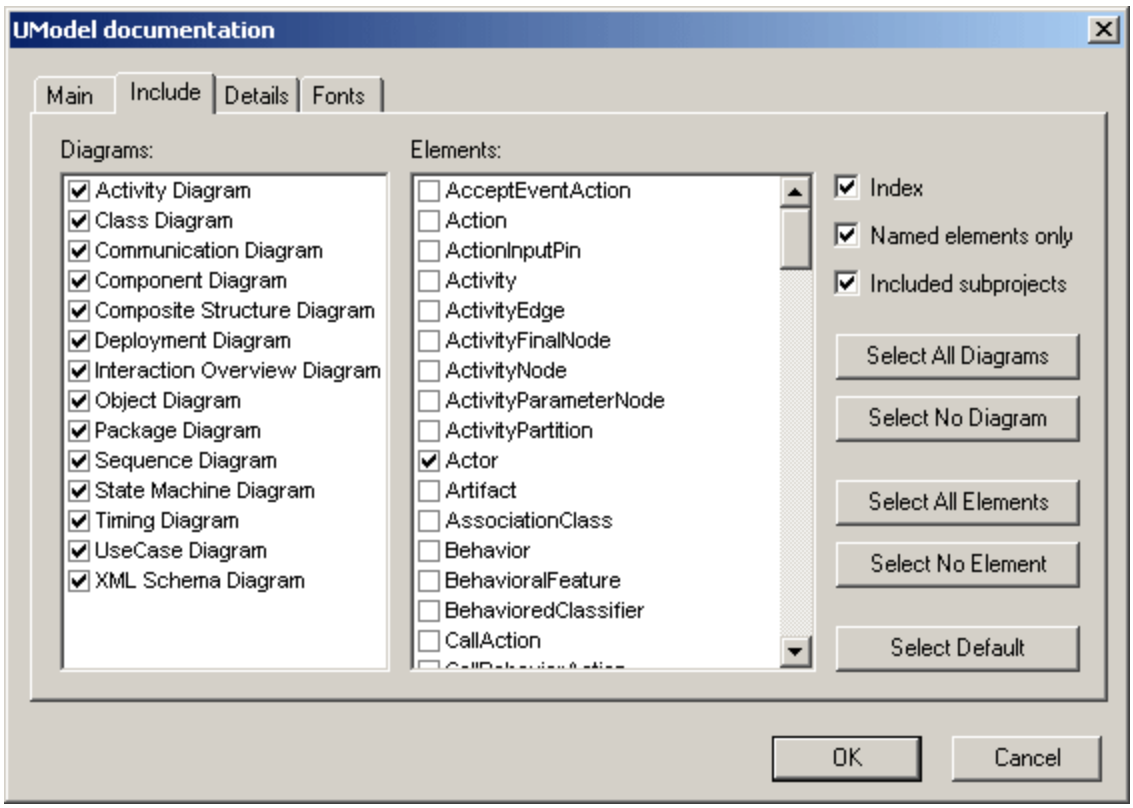


Related elements are hyperlinked in the onscreen output, enabling you to navigate from component to component. Note also that documentation is also generated for included C# and/or Java subprojects (profiles). Note that documenting subprojects can be disabled by deselecting the "Included subprojects" check box.

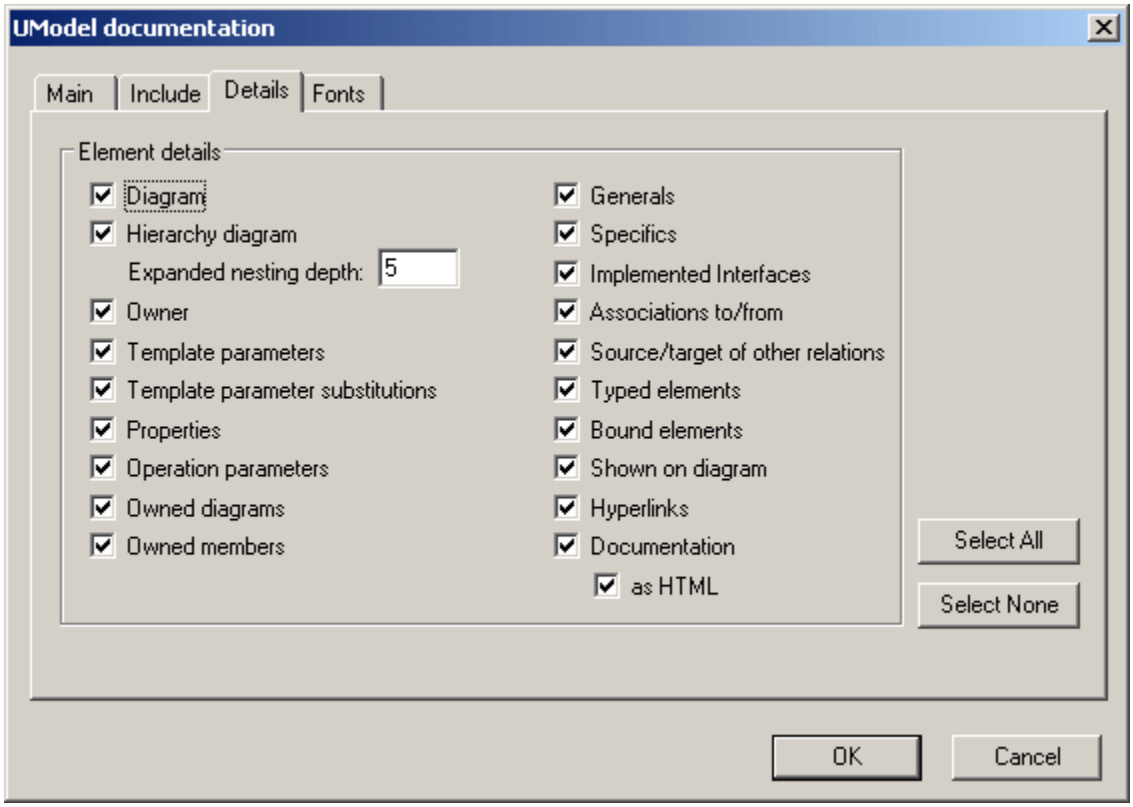
The **Embed diagrams** option is enabled for the Microsoft Word and RTF output options. When this option is selected, diagrams are embedded in the generated file. Diagrams are created as PNG files (for HTML), or PNG/EMF files (for MS Word and RTF), which are displayed in the result file via object links.

Split output to multiple files generates an output file for each modeling element that would appear in the TOC overview when generating a single output file e.g. a class C1 with a nested class CNest exists; C1.html contains all info pertaining to C1 and CNest as well as all their attributes, properties etc.

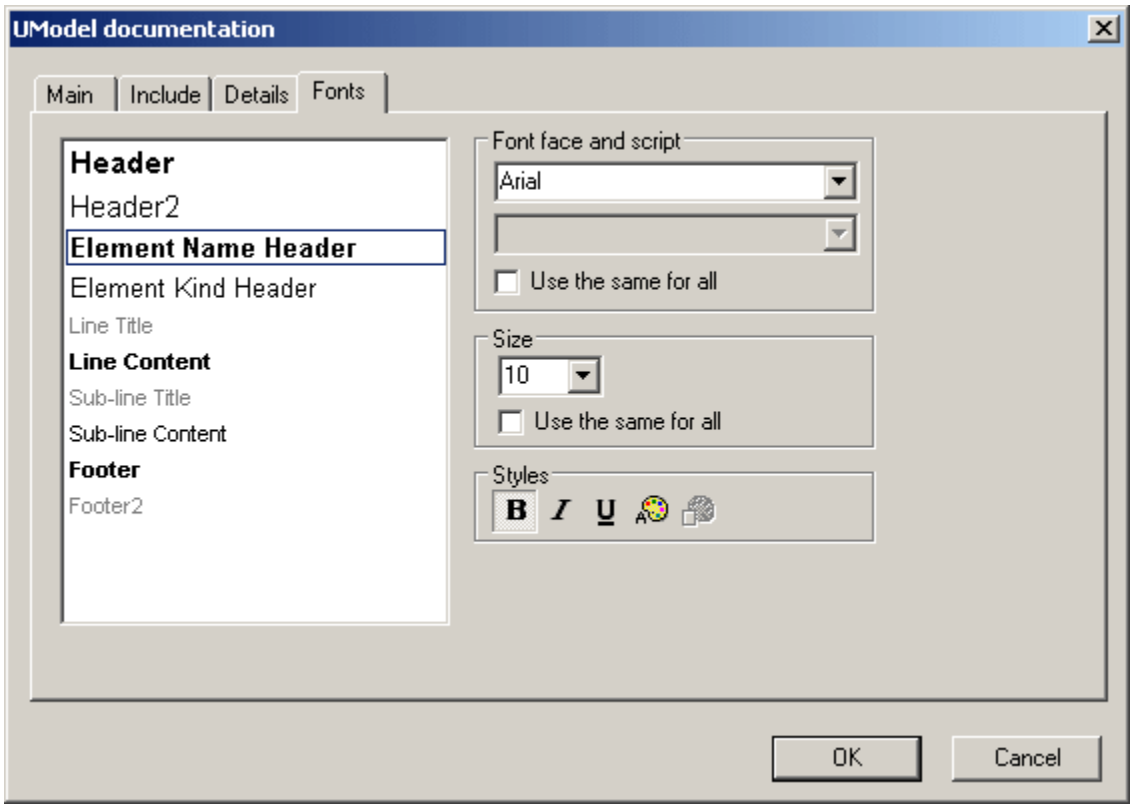
The **Include** tab allows you to select which diagrams and modeling elements are to appear in the documentation.



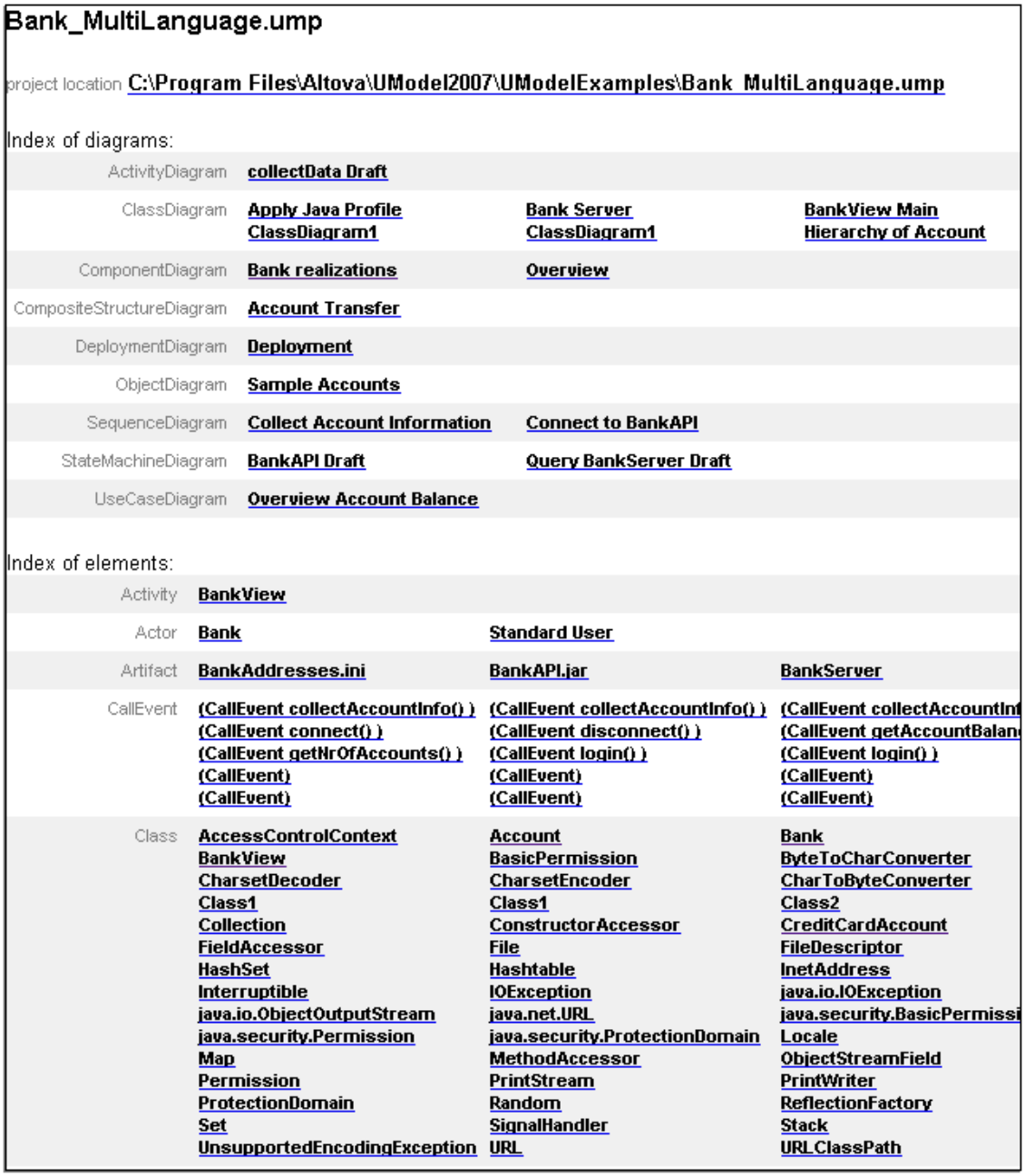
The **Details** tab allows you to select the element details that are to appear in the documentation.



The Fonts tab allows you to customize the font settings for the various headers and text content.



The following screenshots show the generated documentation for the Bank_MultiLanguage.ump file that is included in the ...\UModelExamples directory.



Bank_MultiLanguage.ump

project location C:\Program Files\Altova\UModel2007\UModelExamples\Bank_MultiLanguage.ump

Index of diagrams:

ActivityDiagram	collectData Draft		
ClassDiagram	Apply Java Profile ClassDiagram1	Bank Server ClassDiagram1	BankView Main Hierarchy of Account
ComponentDiagram	Bank realizations	Overview	
CompositeStructureDiagram	Account Transfer		
DeploymentDiagram	Deployment		
ObjectDiagram	Sample Accounts		
SequenceDiagram	Collect Account Information	Connect to BankAPI	
StateMachineDiagram	BankAPI Draft	Query BankServer Draft	
UseCaseDiagram	Overview Account Balance		

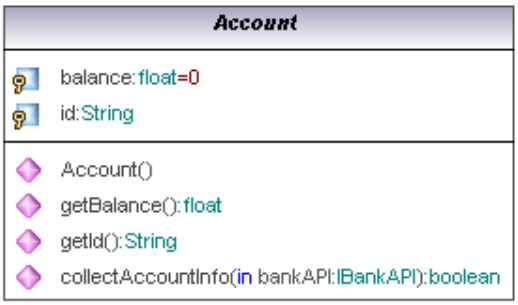
Index of elements:

Activity	BankView		
Actor	Bank	Standard User	
Artifact	BankAddresses.ini	BankAPI.jar	BankServer
CallEvent	(CallEvent collectAccountInfo()) (CallEvent connect()) (CallEvent getNrOfAccounts()) (CallEvent) (CallEvent)	(CallEvent collectAccountInfo()) (CallEvent disconnect()) (CallEvent login()) (CallEvent) (CallEvent)	(CallEvent collectAccountInf (CallEvent getAccountBalan (CallEvent login()) (CallEvent) (CallEvent)
Class	AccessControlContext BankView CharsetDecoder Class1 Collection FieldAccessor HashSet Interruptible java.io.ObjectOutputStream java.security.Permission Map Permission ProtectionDomain Set UnsupportedEncodingException	Account BasicPermission CharsetEncoder Class1 ConstructorAccessor File Hashtable IOException java.net.URL java.security.ProtectionDomain MethodAccessor PrintStream Random SignalHandler URL	Bank ByteToCharConverter CharToByteConverter Class2 CreditCardAccount FileDescriptor InetAddress java.io.IOException java.security.BasicPermissi Locale ObjectStreamField PrintWriter ReflectionFactory Stack URLClassPath

The screenshot above shows the generated documentation with the diagram and element index links at the top of the HTML file. The screenshot below shows the specifics of the Account class and its relation to other classes.

Note that the individual attributes and properties in the class diagrams are also hyperlinked to their definitions. Clicking a property takes you to its definition.

Class **Account**

diagram	
hierarchy	
owner	**bankview**
properties	qualified name Design View::BankView::com::altova::bankview::Account visibility public leaf false abstract true active false code file name Account.java code file path C:\UML_Bank_Sample\MultiLanguage\JavaCode\com\altova\b <<annotations>> false <<final>> false <<MyKeyValuePair>> false <<static>> false <<strictfp>> false
ownedMember	**Account** **balance** **collectAccountInfo** **getBalance** **getId** **id**
specific	**CheckingAccount** **CreditCardAccount** **SavingsAccount**
target of relation	ComponentRealization **BankView**
typedElements	Class **Bank** Property **accounts** Interaction **Collect Account Information** Property **b**

Chapter 9

UML Diagrams

9 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 thirteen diagrams of the UML 2.1.1 specification as well as an additional diagram: XML Schema diagram.

Behavioral diagrams include Activity, state machine, and use case diagrams as well as the interaction diagrams Communication Diagram, Interaction Overview Diagram Sequence Diagram Timing Diagram.

Structural diagrams include: class, composite structure, component, deployment, object, and package diagrams.

Additional diagrams XML schema diagrams.

9.1 Behavioral Diagrams

These diagrams depict behavioral features of a system or business process, and include a subset of diagrams which emphasize object interactions.

Behavioral Diagrams

 Activity Diagram
 State Machine Diagram
 Use Case Diagram

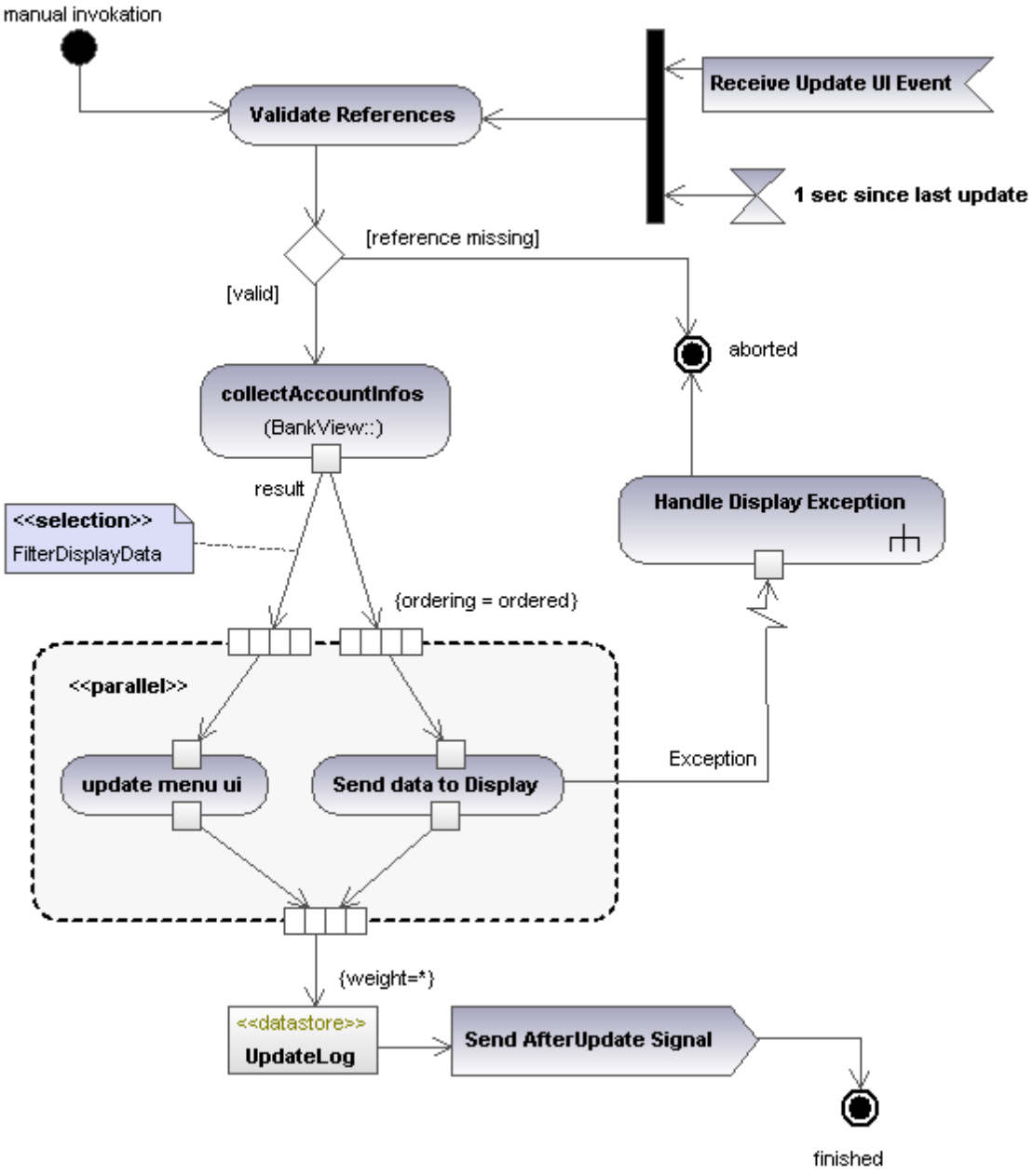
A subset of the Behavioral diagrams are those that depict the object interactions, namely:

 Communication Diagram
 Interaction Overview Diagram
 Sequence Diagram
 Timing Diagram

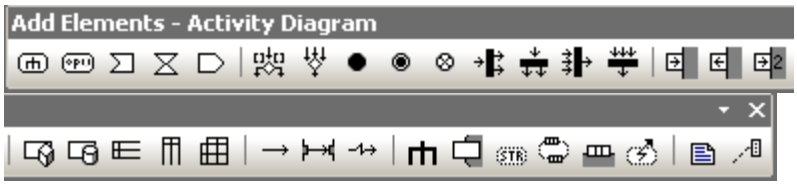
9.1.1 Activity Diagram

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.

Please note that the Activity diagram shown in the following section is available in the **Bank_MultiLanguage**.**ump** sample, in the ...**UModelExamples** folder supplied with UModel.



Inserting Activity Diagram elements



Using the toolbar icons:

1. Click the specific activity diagram icon in the Activity Diagram toolbar.
2. Click in the Activity Diagram to insert the element.
   Note that holding down CTRL and clicking in the diagram tab, allows you to insert multiple elements of the type you selected.

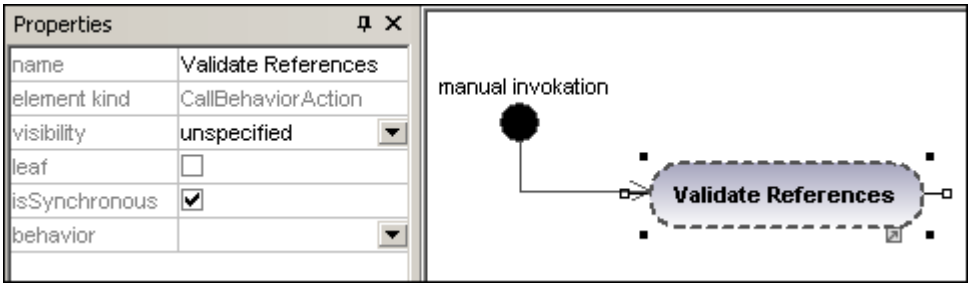
Dragging existing elements into the activity diagram:

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 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 activity diagram.

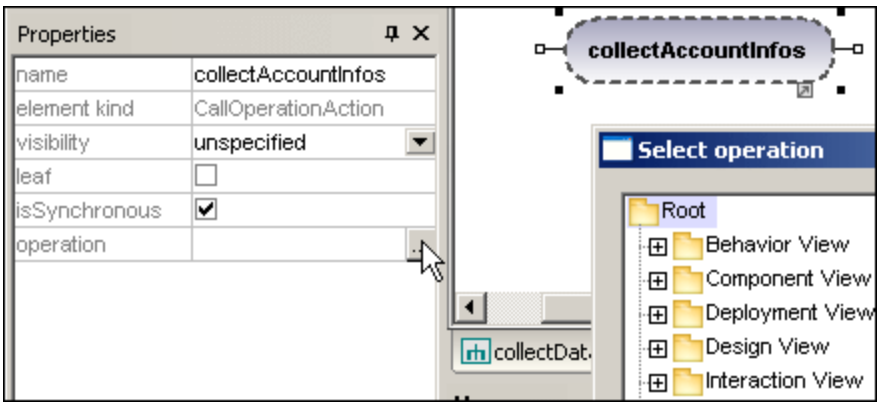
Inserting an action (CallBehavior):

1. Click the Action (CallBehavior) icon  in the icon bar, 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.



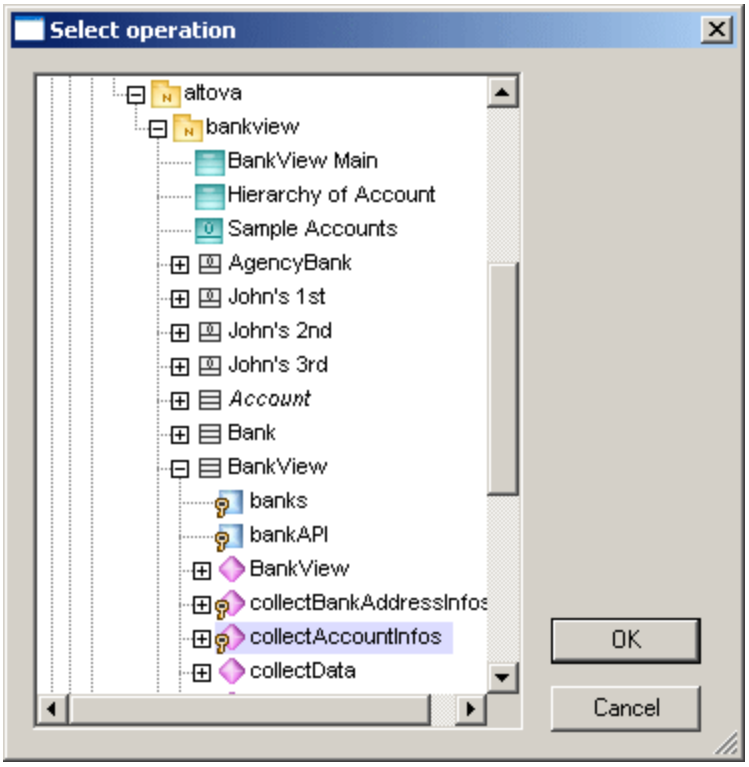
Inserting an action (CallOperation) and selecting a specific operation:

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.

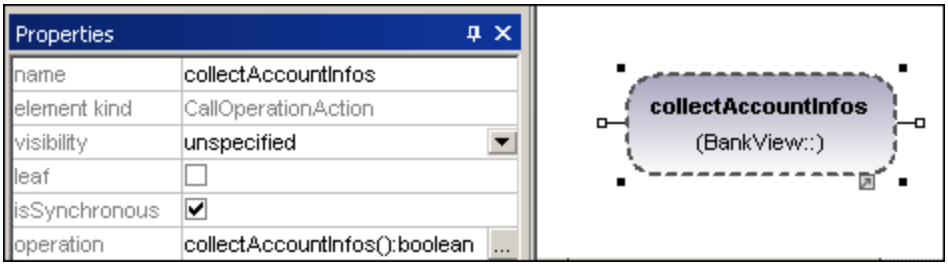


This opens the "Select Operation" dialog box in which you can select the specific operation.

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.



Creating branches and merges

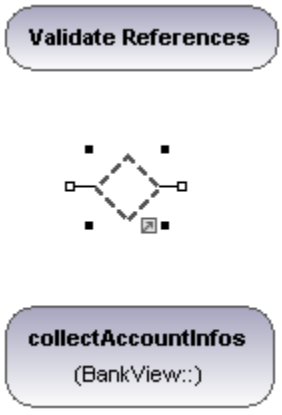
Creating a branch (alternate flow)

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.

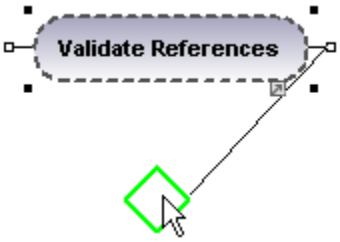
In this example the (BankView) references are to be validated:

- branch1 has the guard "reference missing", which transitions to the abort activity
- branch2 has the guard "valid", which transitions to the collectAccountInfos activity.

1. Click the **DecisionNode** icon  in the title bar, and insert it in the Activity diagram.

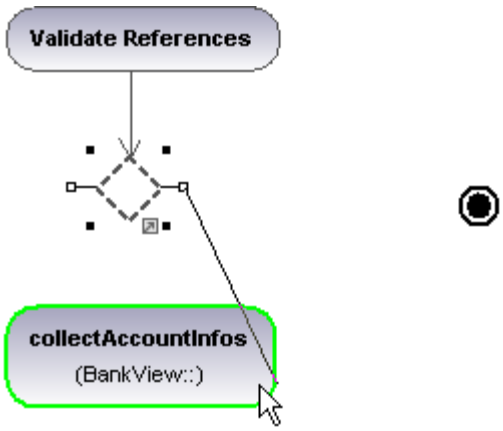


2. Click the ActivityFinalNode icon  which represents the abort activity, and insert it into the Activity diagram.
3. Click the Validate References activity to select it, then click the right-hand handle, **ControlFlow**, and drag the resulting connector onto the DecisionNode element.

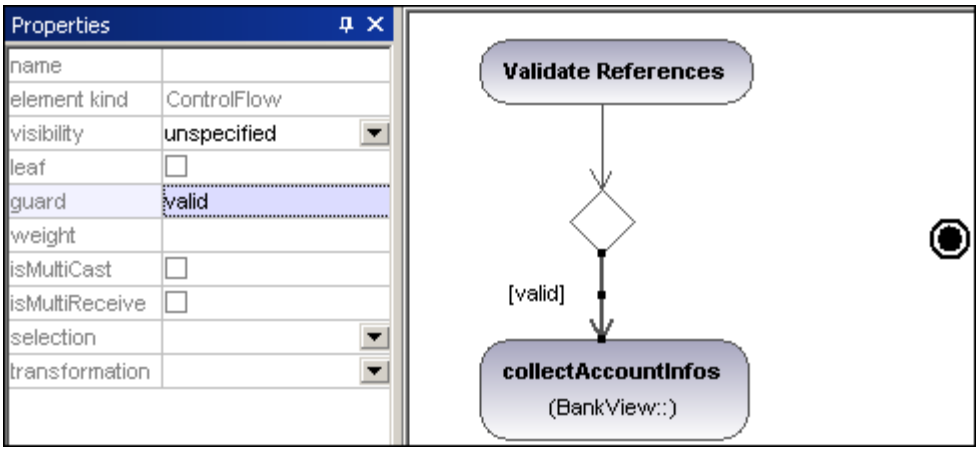


The element is highlighted when you can drop the connector.

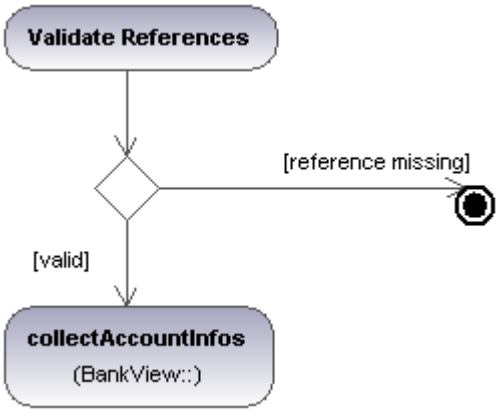
4. Click the DecisionNode element, click the right-hand connector, **ControlFlow**, and drop it on the collectAccountInfos action. Please see "Inserting an Action (CallOperation" for more information.



5. Enter the guard condition "valid", in the guard field of the Properties tab.



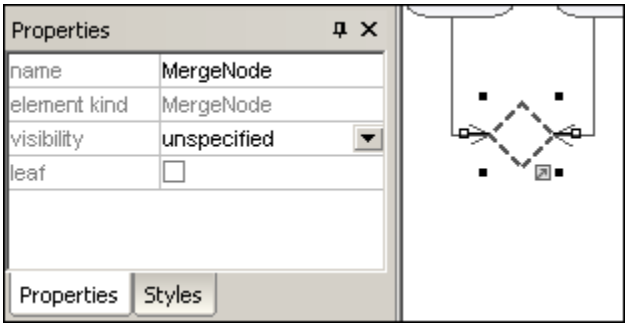
6. Click the **DecisionNode** element and drag from the right-hand handle, **ControlFlow**, and drop it on the ActivityFinalNode element.
The guard condition on this transition is automatically defined as "else". Double click the guard condition in the diagram to change it e.g. "reference missing".



Please note that UModel does not validate, or check, the number of Control/Object Flows in a diagram.

Creating a merge:

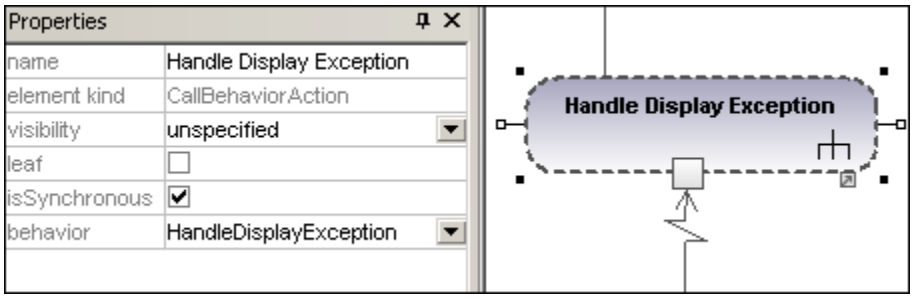
1. Click the MergeNode icon  in the icon bar, then click in the Activity diagram to insert it.



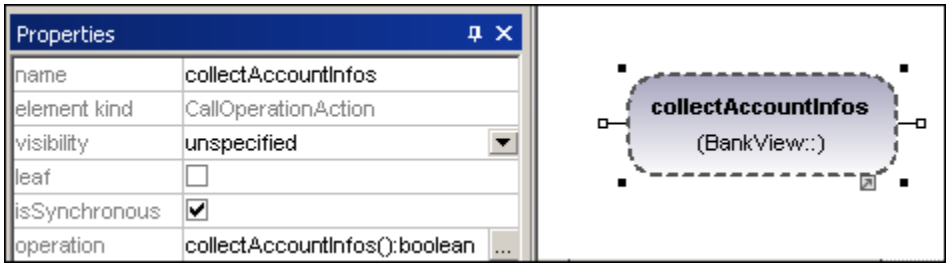
2. Click the ControlFlow (ObjectFlow) handles of the actions that are to be merged, and drop the arrow(s) on the MergeNode symbol.

Diagram elements

 Action (CallBehavior)
Inserts the Call Behavior Action element which directly invokes a specific behavior.
Selecting an existing behavior using the **behavior** combo box, e.g. HandleDisplayException, and displays a rake symbol within the element.

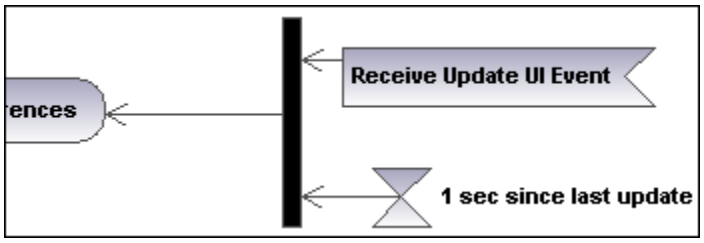


 Action (CallOperation)
Inserts the Call Operation Action which indirectly invokes a specific behavior as a method.
Please see "Inserting an action (CallOperation)" for more information.

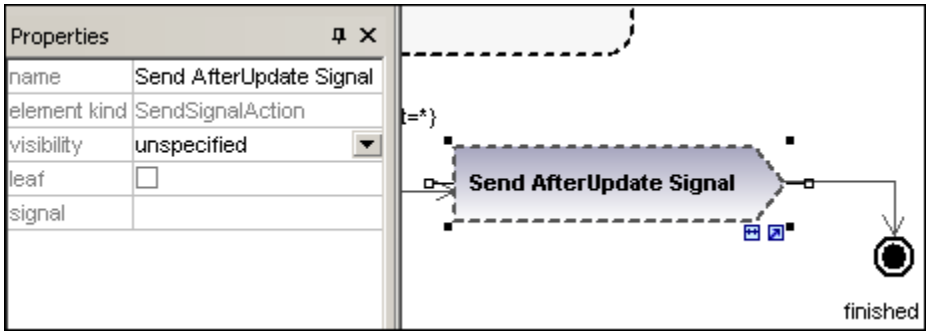


 AcceptEventAction
Inserts the Accept Event action which waits for the occurrence of an event which meets specific conditions.

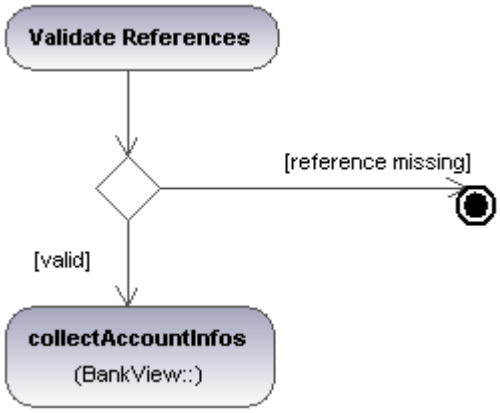

AcceptEventAction (TimeEvent)
Inserts a AcceptEvent action, triggered by a time event, which specifies an instant of time by an expression e.g. 1 sec. since last update.




SendSignalAction
Inserts the Send Signal action, which creates a signal from its inputs and transmits the signal to the target object, where it may cause the execution of an activity.




DecisionNode
Inserts a Decision Node which has a single incoming transition and multiple outgoing guarded transitions. Please see "Creating a branch" 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.

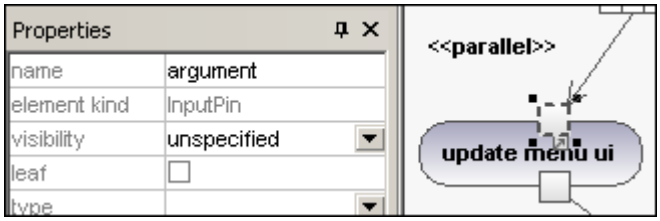
 ForkNode
Inserts a vertical Fork node.
Used to divide flows into multiple concurrent flows.

 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.

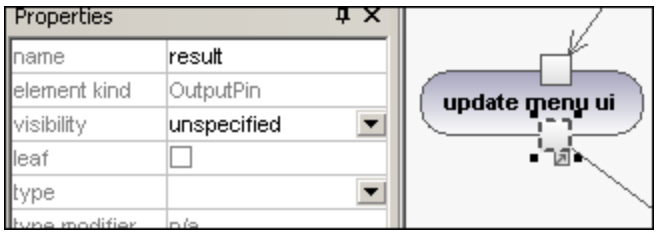
 Join Node (horizontal)
Inserts a horizontal Fork node.
A Join node synchronizes multiple flows defined by the Fork node.

 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.



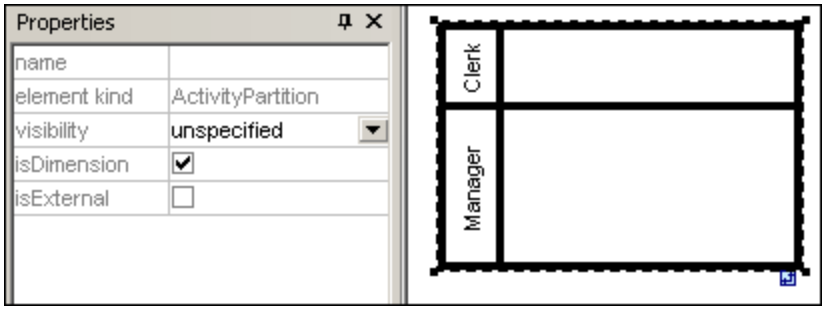
The output pin symbol can only be placed onto those activity elements where the mouse pointer changes to the hand symbol. Dragging the symbol repositions it on the element border.

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.

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.

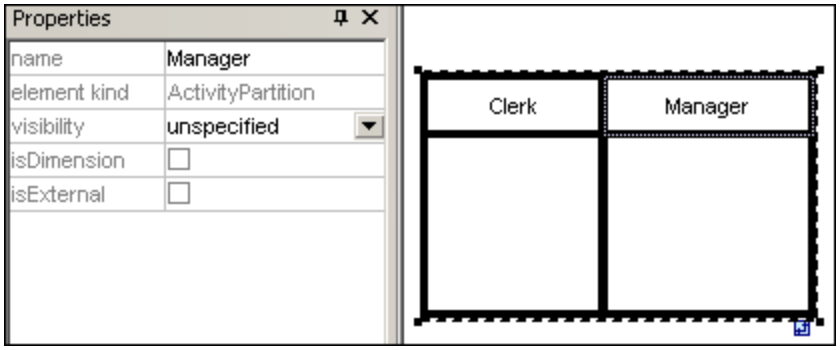
ActivityPartition (horizontal)
Inserts a horizontal Activity Partition, which is a type of activity group used to identify actions that have some characteristic in common. This often corresponds to organizational units in a business model.



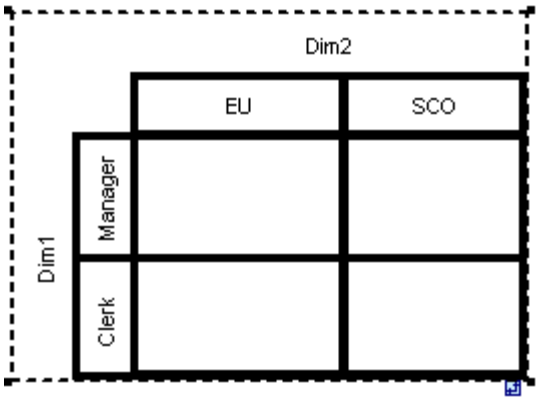
Double clicking a label allows you to edit it directly; pressing Enter orients the text correctly.

Please note that Activity Partitions are the UML 2.0 update to the "swimlane" functionality of previous UML versions.

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.

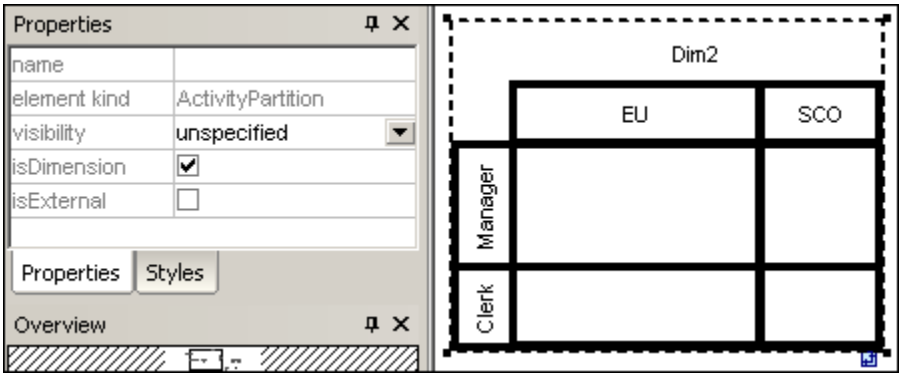



ActivityPartition (2 Dimensional)
Inserts a two dimensional Activity Partition, which is a type of activity group used to identify actions that have some characteristic in common. Both axes have editable labels.



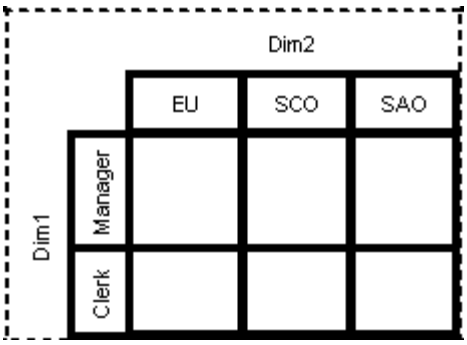
To remove the Dim1, Dim2 dimension labels:

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.

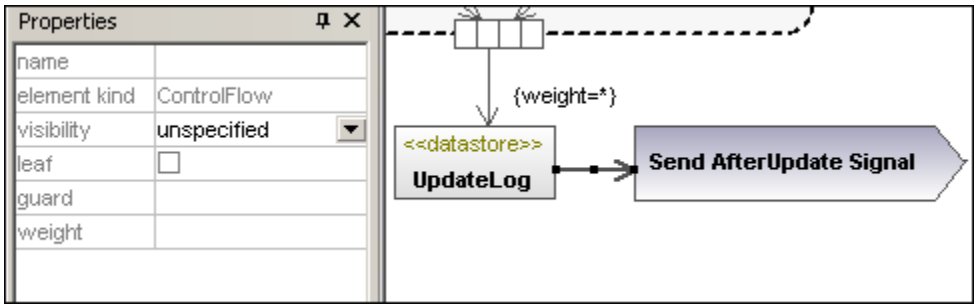


Note that Activity Partitions can be nested:

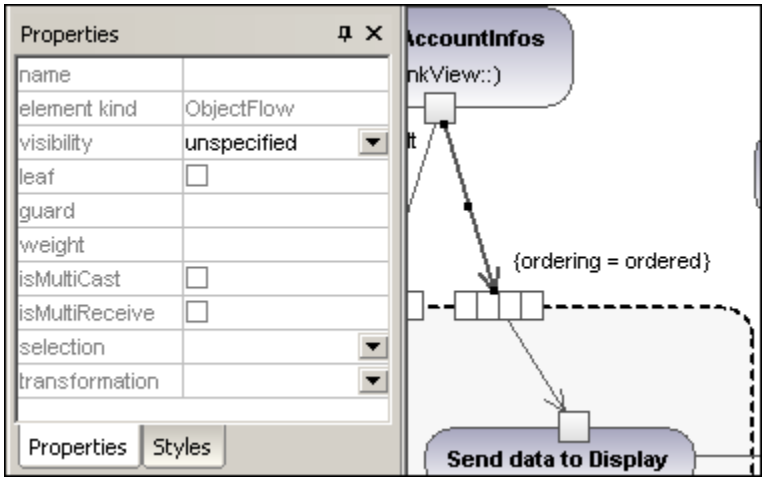
1. Right click the label where you want to insert a new partition.
2. Select **New | ActivityPartition**.



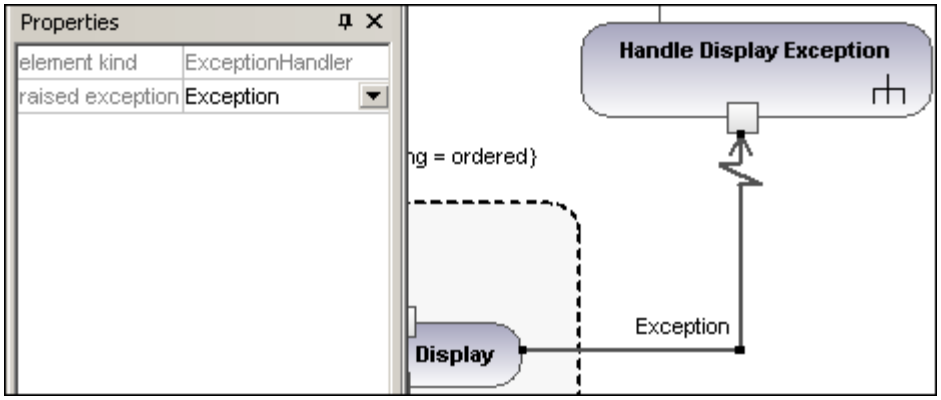
ControlFlow
A Control Flow is an edge, i.e. an arrowed line, that connects two activities/behaviours, and starts an activity after the previous one has been completed.



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.

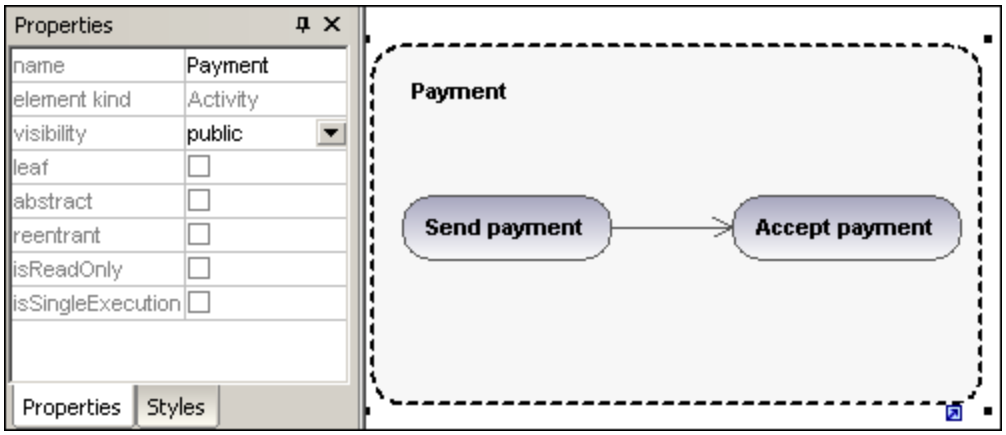


ExceptionHandler
An Exception Handler is an element that specifies what action is to be executed if a specified exception occurs during the execution of the protected node.

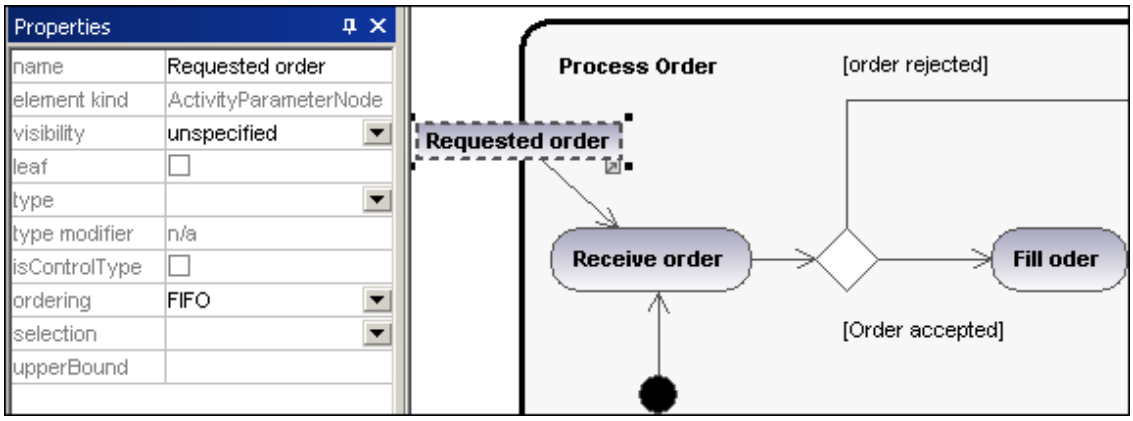


An Exception Handler can only be dropped on an Input Pin of an Action.

Activity
Inserts an Activity into the activity diagram.

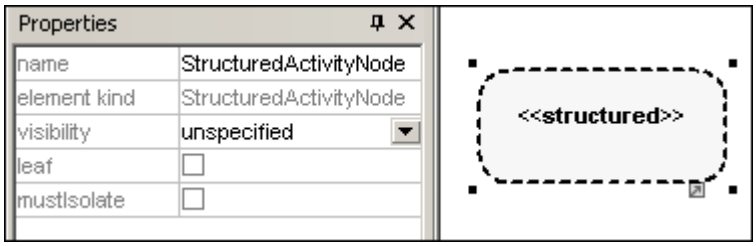


ActivityParameterNode
Inserts an Activity Parameter node onto an activity. Clicking anywhere in the activity places the parameter node on the activity boundary.

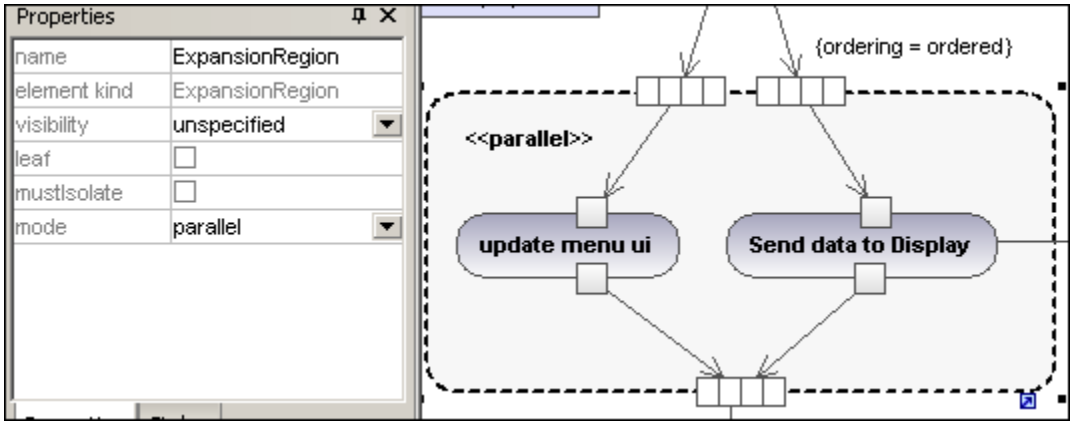


StructuredActivityNode

Inserts a Structured Activity Node which is a structured part of the activity, that is not shared with any other structured node.

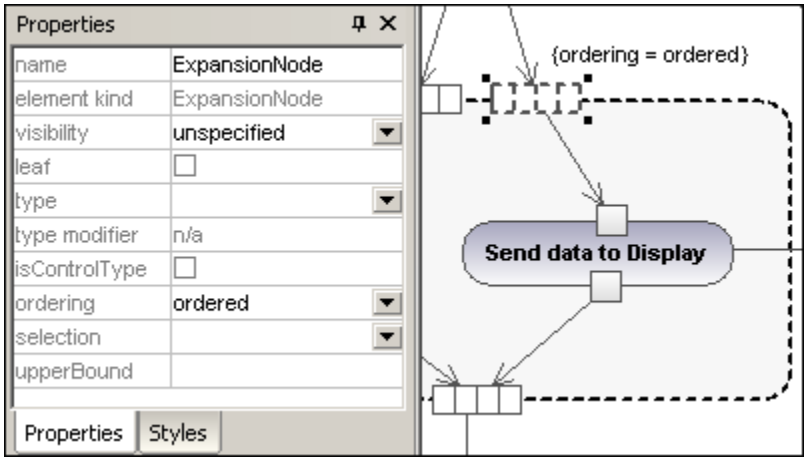


ExpansionRegion
An expansion region is a region of an activity having explicit input and outputs (using ExpansionNodes). Each input is a collection of values.



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.

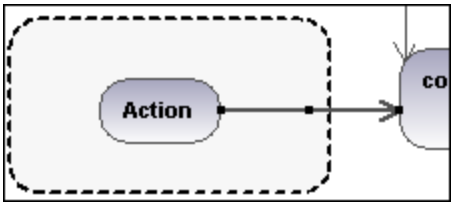
ExpansionNode
Inserts an Expansion Node onto an Expansion Region. Expansion nodes are input and output nodes for the Expansion Region, where each input/output is a collection of values. The arrows into, or out of, the expansion region, determine the specific type of expansion node.



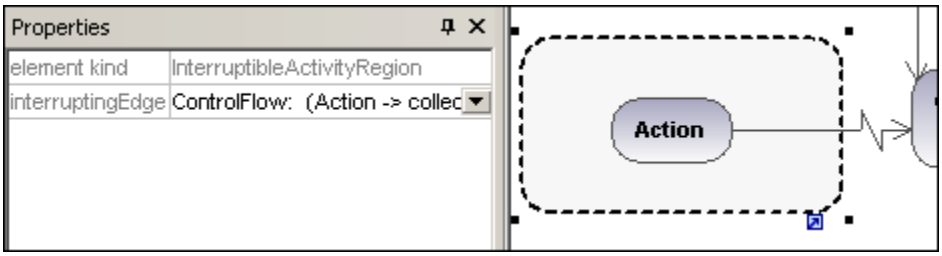
 InterruptableActivityRegion
An interruptible region contains activity nodes. When a control flow leaves an interruptible region all flows and behaviors in the region are terminated.

To add an interrupting edge:
Making sure that:

- an Action element is present in the InterruptableActivityRegion, as well as an outgoing Control Flow to another action:



1. Right click the Control Flow arrow, and select **New | InterruptingEdge**.



Please 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.

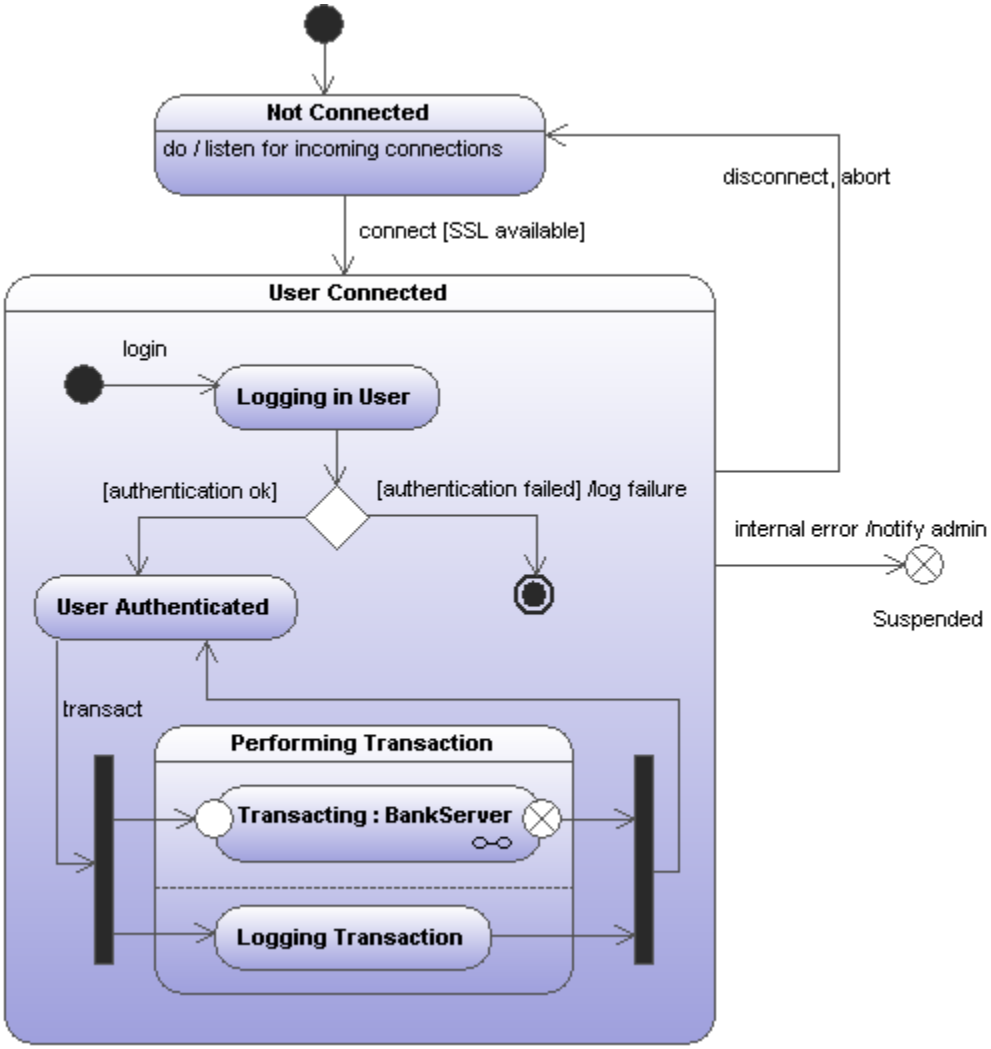
9.1.2 State Machine Diagram

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. A state machine can have any number of State Machine Diagrams (or State Diagrams) UModel.

Two types of processes can achieve this:
Actions, which are associated to **transitions,** are short-term processes that cannot be interrupted. E.g. an initial transition, **internal error /notify admin**.

State **Activities** (behaviors), which are associated to **states,** are longer-term processes that may be interrupted by other events. E.g. **listen for incoming connections**.

Please note that the State machine diagrams shown in the following section are available in the **Bank_MultiLanguage**.ump sample, in the ...**UModelExamples** folder supplied with UModel.



Inserting state machine diagram elements

Using the toolbar icons:

1. Click the specific state machine diagram icon in the State Machine Diagram toolbar.

2. Click in the State Diagram to insert the element.
   Note that holding down CTRL and clicking in the diagram tab, allows you to insert multiple elements of the type you selected.

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.

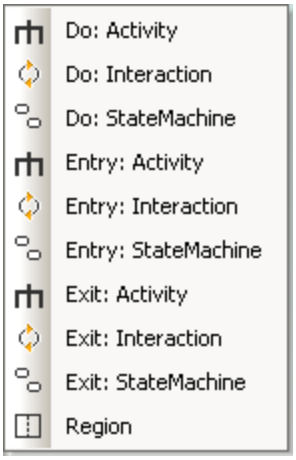
Creating states, activities and transitions

To insert a simple state:

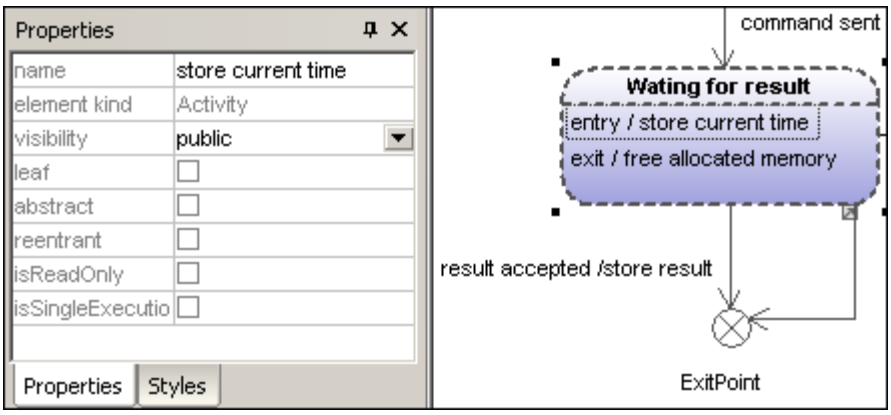
1. Click the state icon  in the icon bar and click in the State diagram to insert it.
2. Enter the name of the state and press Enter to confirm.
   Simple states do not have any regions or any other type of substructure. UModel allows you to add activities as well as regions to a simple state through the context menu.

To add an activity to a state:

1. Right click the state element, select New, and then one of the entries from the context menu.



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**.

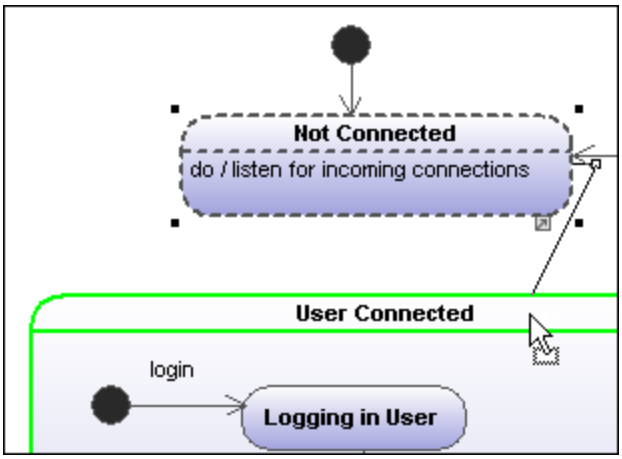


To delete an activity:

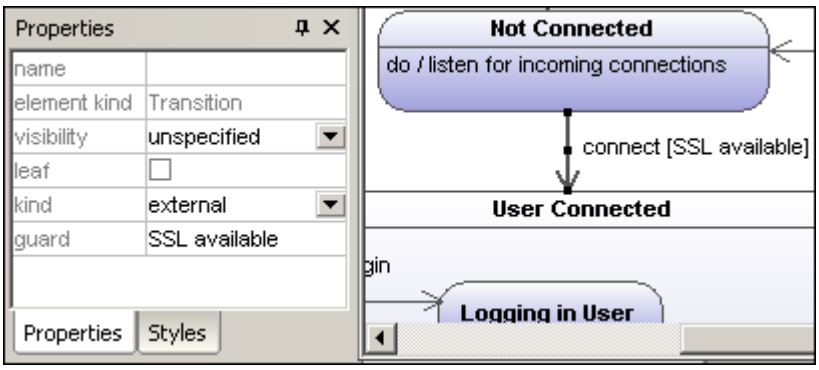
1. Click the respective activity in the state element and press the Del. key.

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 create a transition trigger:

1. Right click a previously created transition (arrow).

2. Select **New | Trigger**.

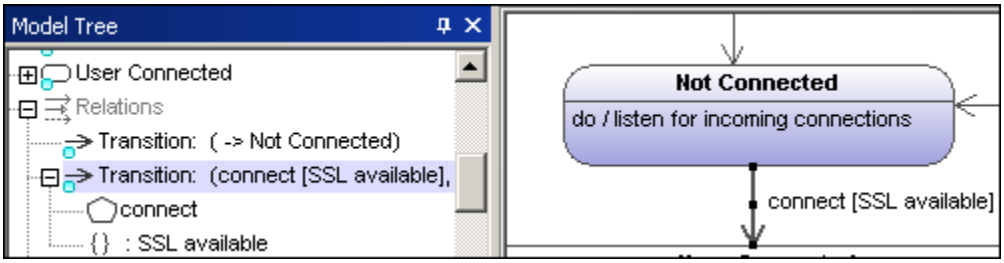
An "a" character appears in the transition label above the transition arrow, if it is the first trigger in the state diagram. Triggers are assigned default values of the form alphabetic letter, source state -> target state.

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, between brackets the guard condition, and after the slash, the activity. Manipulating this string automatically creates or deletes the respective elements in the Model Tree.

Please note:

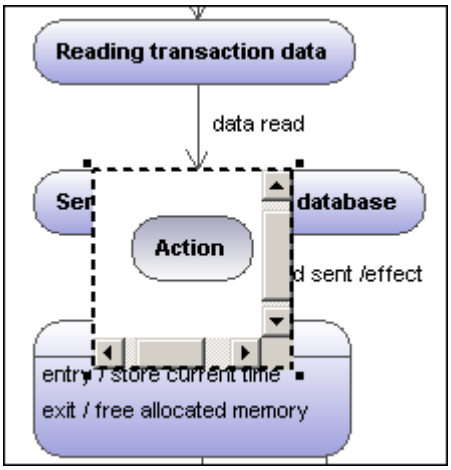
To see the individual transition properties, right click the transition (arrow) and select "Select in Model Tree". The event, activity and constraint elements are all shown below the selected transition.



Adding an Activity diagram to a transition:

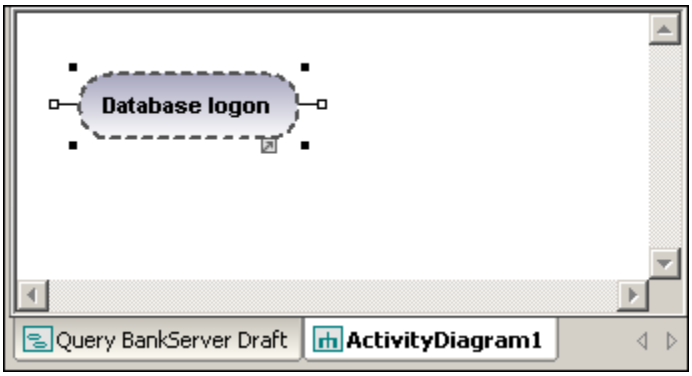
UModel has the unique capability of allowing you to add an Activity diagram to a transition, to describe the transition in more detail.

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.



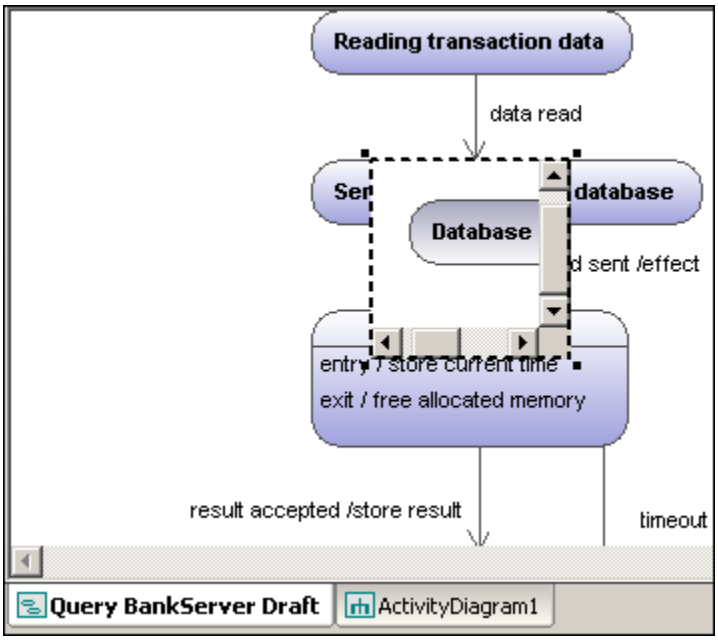
3. Double click the Action window to switch into the Activity diagram and further define the

transition, e.g. change the Action name to Database logon.

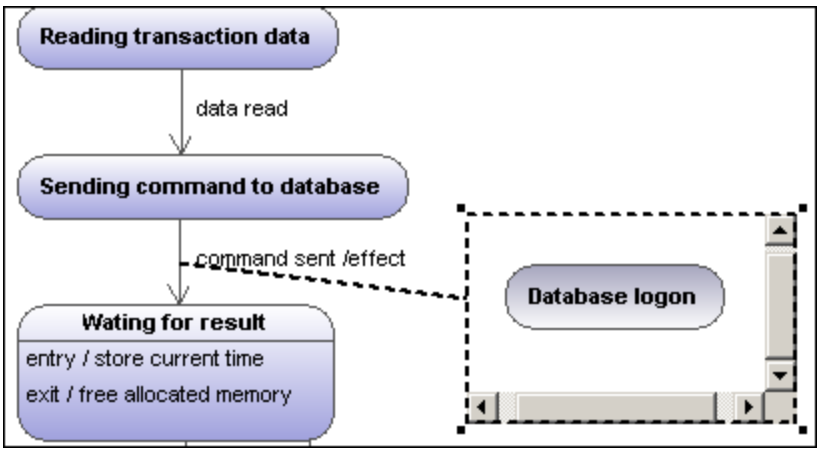


Note that a new Activity Diagram tab has now been added to the project. You can add any activity modeling elements to the diagram, please see "Activity Diagram" for more information.

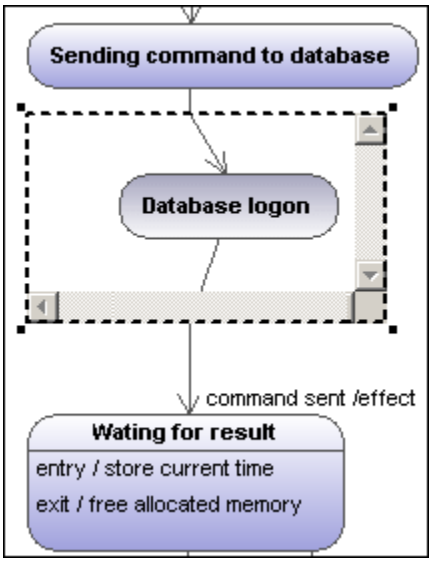
4. Click the State Machine Diagram tab to switch back to see the update transition.



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.



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:

1. 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.

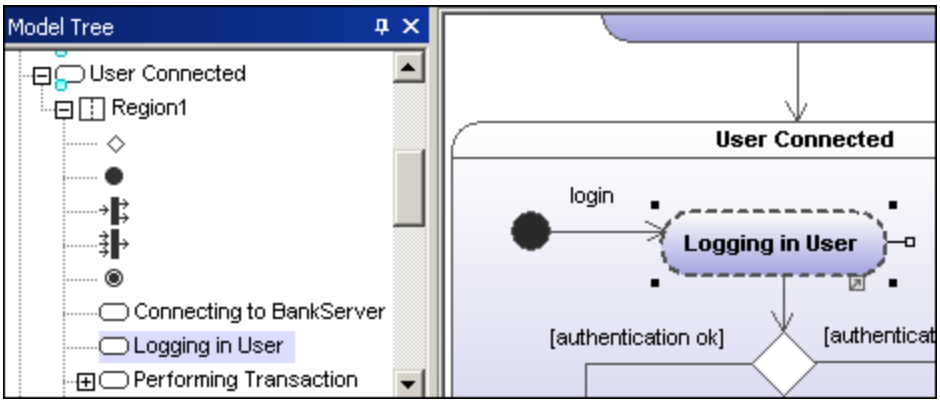
To delete a region:

1. 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:

1. 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.

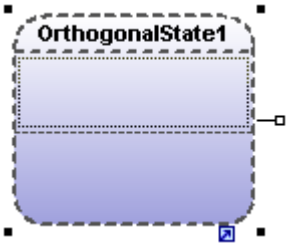


Moving the composite state moves all contained states along with it.

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.

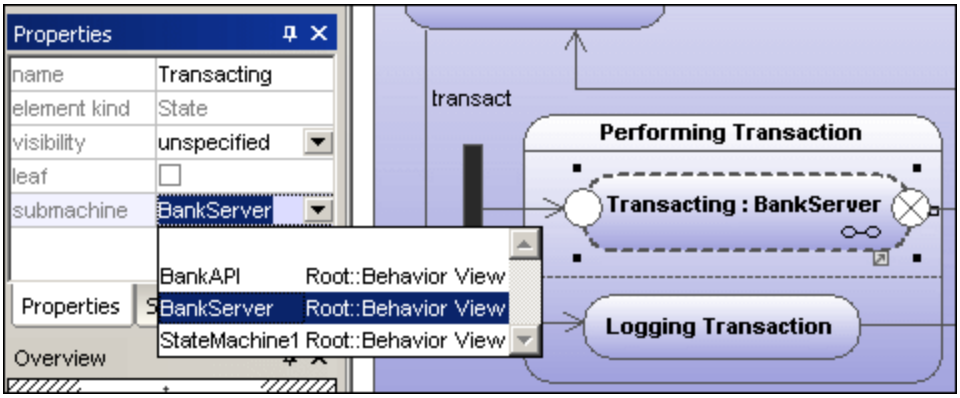


Submachine state

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:

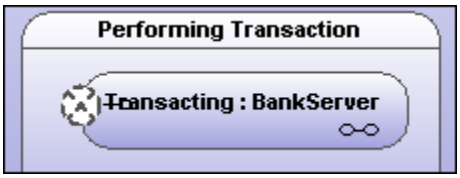
1. Having selected a state, click the **submachine** combo box in the Properties tab.
   A list containing the currently defined state machines appears.
2. Select the state machine that you want this submachine to reference.



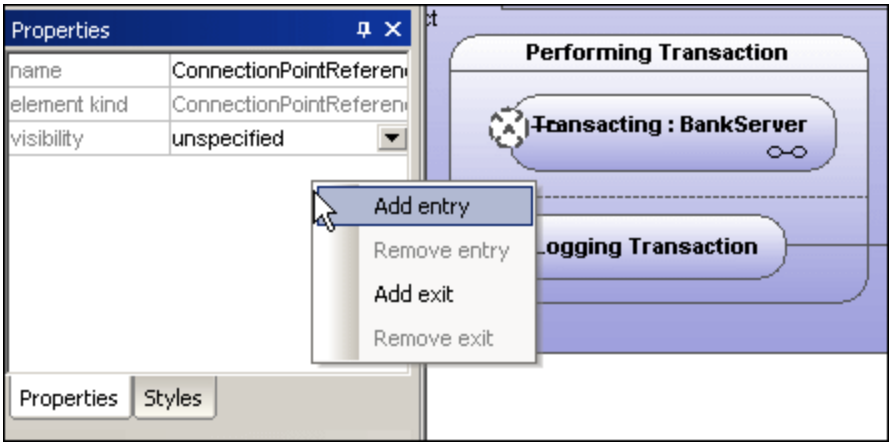
To add entry / exit points to a submachine state:

- 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

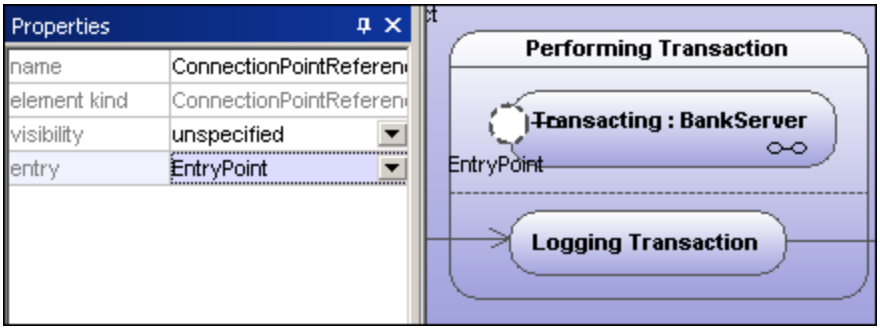
1. Click the **ConnectionPointReference** icon  in the title bar, then click the submachine state that you want to add the entry/exit point to.



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 ConnectionPointReferece element.



3. Use the same method to insert an ExitPoint, by selecting "Add exit" from the context menu.

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)

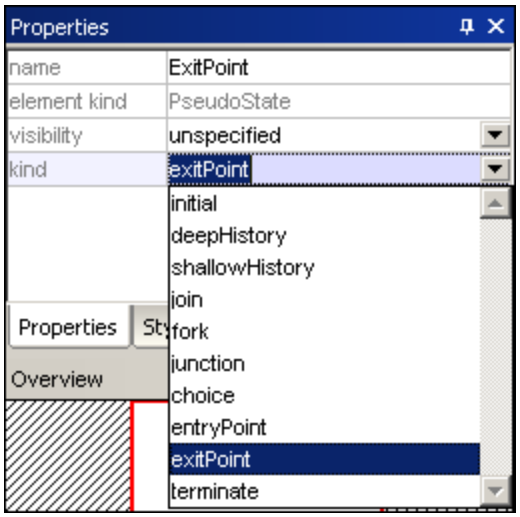
Joins/merges previously defined subsequences. All activities have to be completed before progress can continue.

 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.



 ConnectionPointReference
A connection point reference represents a usage (as part of a submachine state) of an entry/exit point defined in the
statemachine reference by the submachine state.

To add Entry or Exit points to a connection point reference:

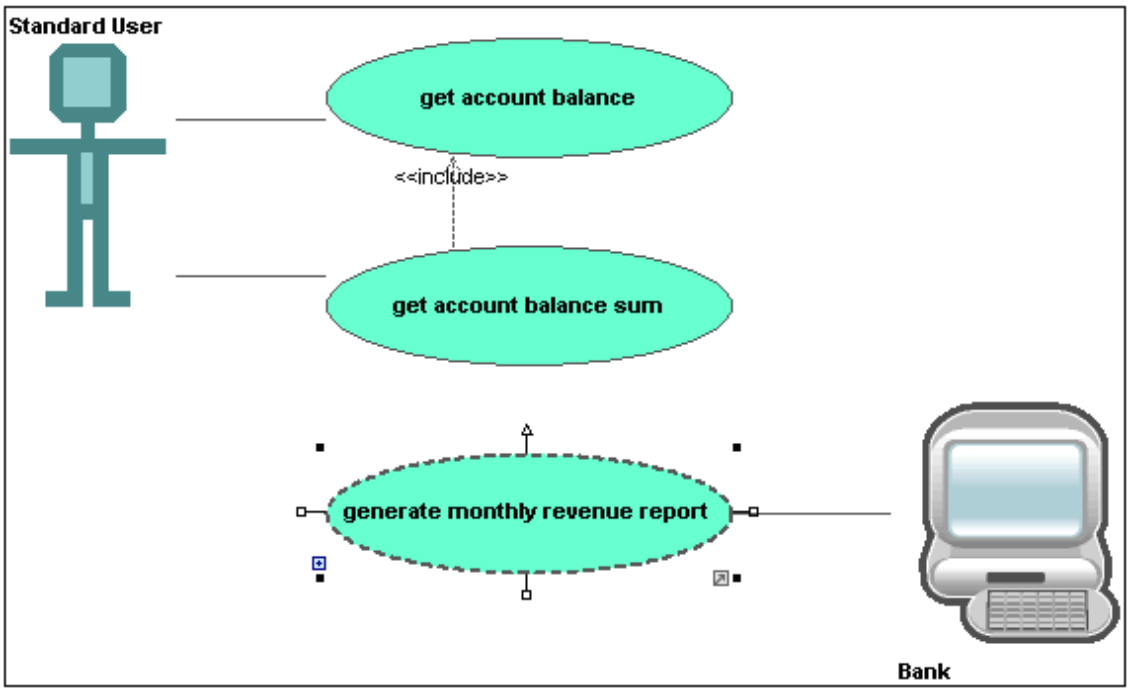
- The state which the point is connected to, must itself reference a submachine State Machine (visible in the Properties tab).
- This submachine must contain one or more Entry and Exit points

 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.

9.1.3 Use Case Diagram

Please see the Use Cases section in the tutorial for more information on how to add use case elements to the diagram.

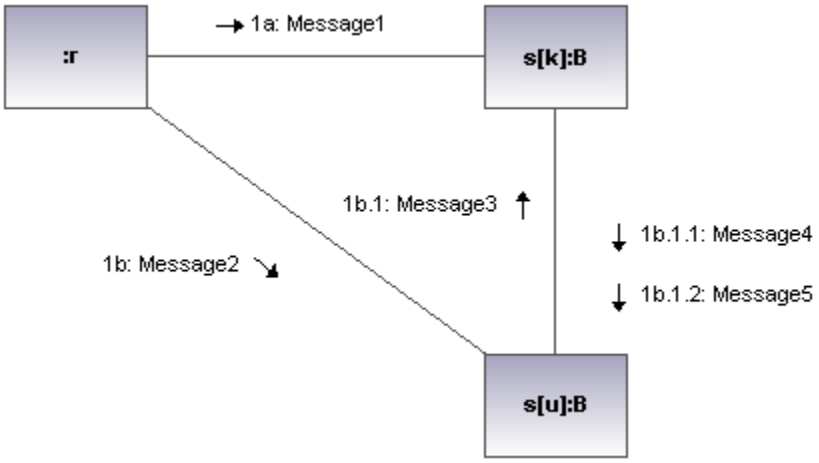


9.1.4 Communication Diagram

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" for more information.



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.
   Note that holding down CTRL and clicking in the diagram tab, allows you to insert multiple elements of the type you selected.

Dragging existing elements into the Communication Diagram:
Elements occurring in other diagrams, e.g. classes, can be inserted into a Communication diagram.

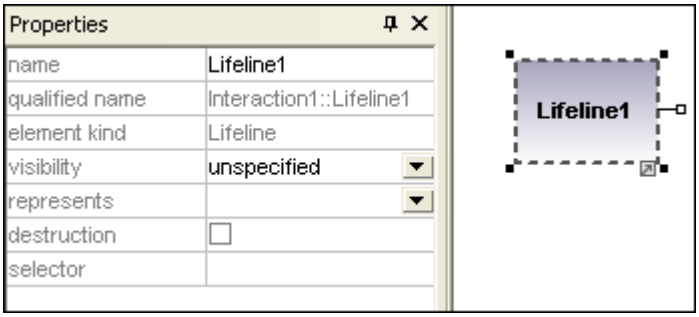
1. Locate the element you want to insert in the Model Tree tab (you can use the search function text box, or press CTRL + F, to search for any element).
2. Drag the element(s) into the Communication diagram.

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 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

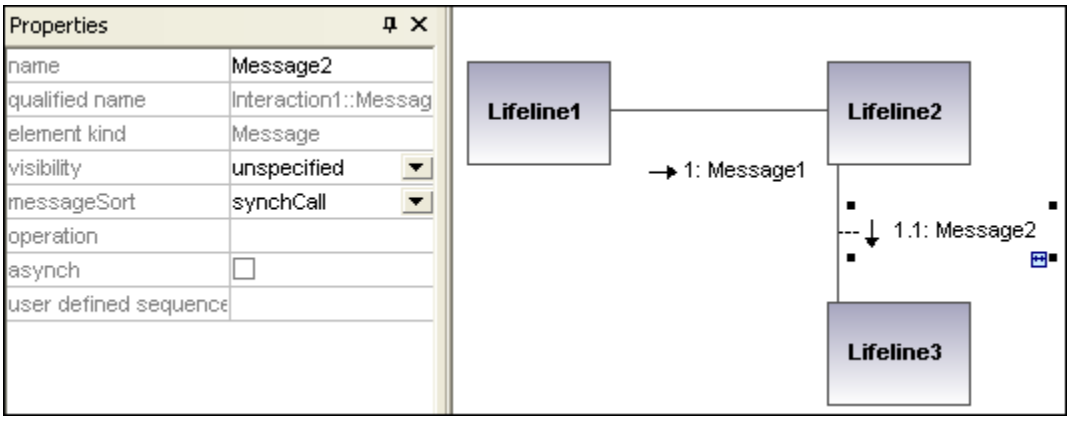
A Message is a modeling element that defines a specific kind of communication in an interaction. A communication can be e.g. raising a signal, invoking an Operation, creating or destroying an instance. The message specifies the type of communication as well as the sender and the receiver.

 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.

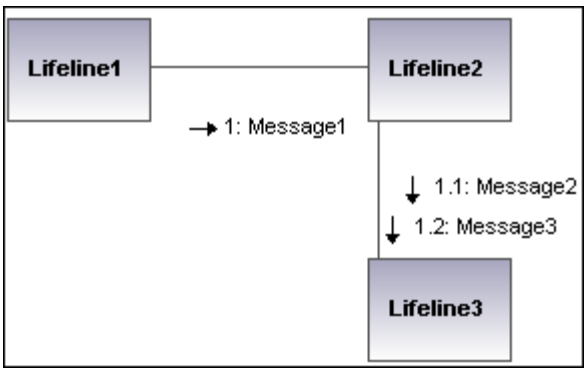
   Lifelines are highlighted when the message can be dropped.



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**.

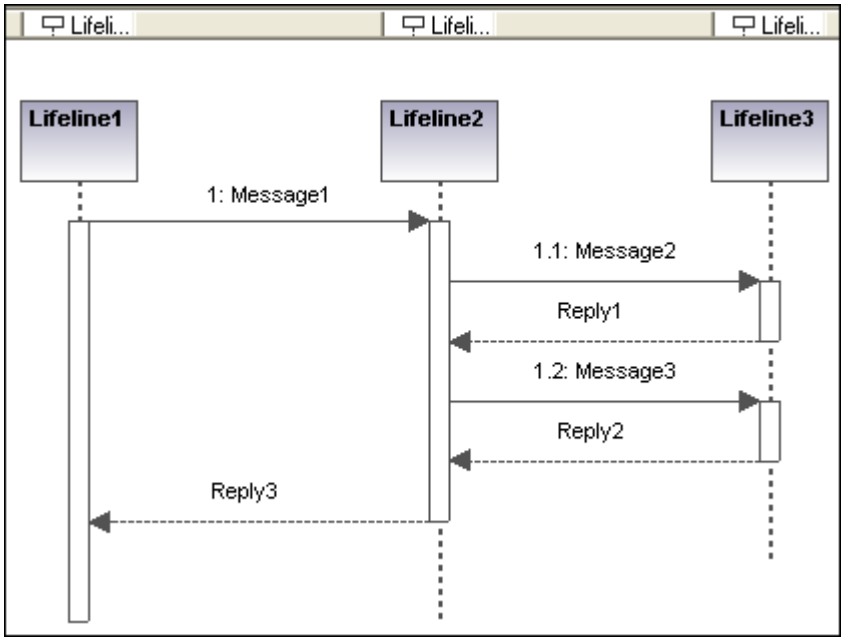


- The direction in which you drag the arrow defines the message direction. Reply messages can point in either direction.
- Having clicked a message icon and holding down CTRL, allows you to insert multiple messages by repeatedly clicking and dragging in the diagram tab.

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.

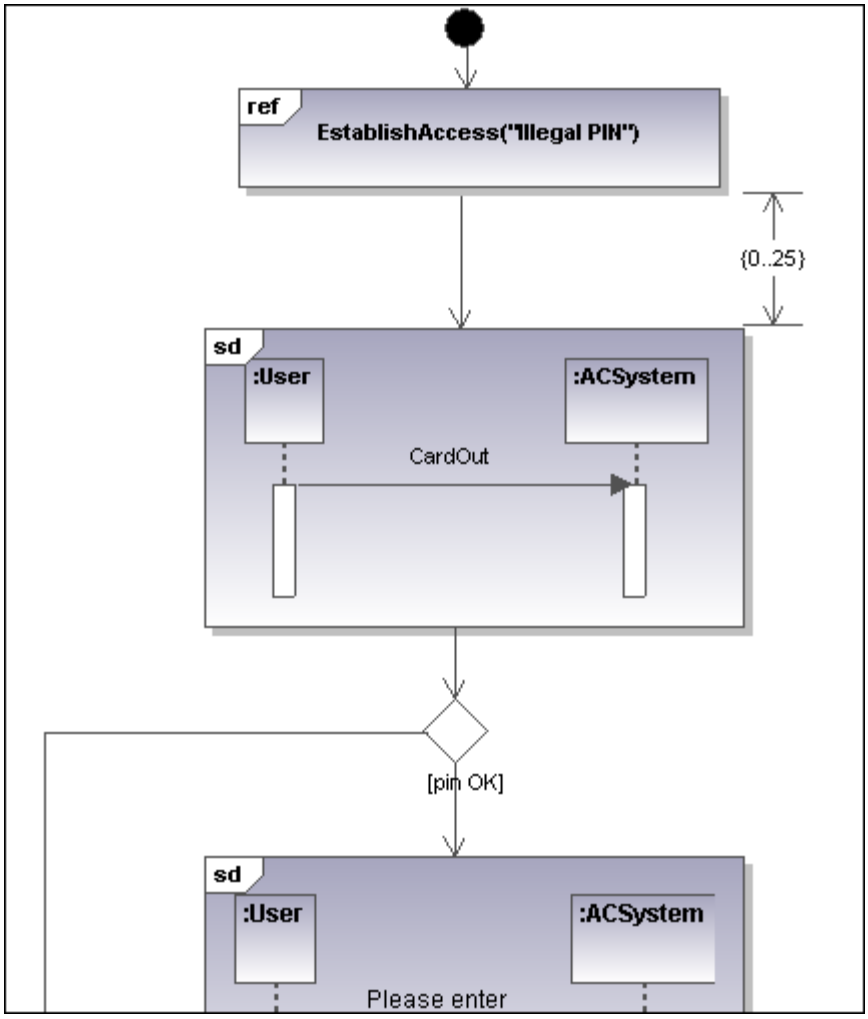
Generating Sequence diagrams from Communication diagrams:
UModel allows you to generate Communication diagrams from Sequence diagrams and vice versa, in one simple action:

- Right click anywhere in a Communication diagram and select **Generate Sequence Diagram** from the context menu.



9.1.5 Interaction Overview Diagram

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.



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.

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.
   Note that holding down CTRL and clicking in the diagram tab, allows you to insert multiple elements of the type you selected.

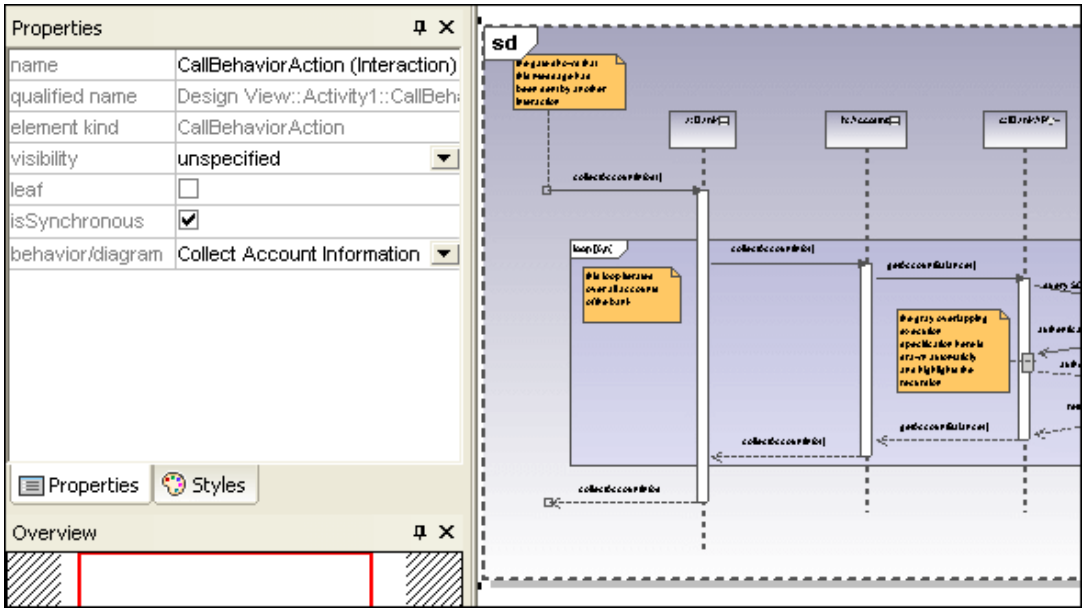
Dragging existing elements into the Interaction Overview Diagram:

Elements occurring in other diagrams, e.g. Sequence, Activity, Communication, or Timing diagrams can be inserted into a Interaction Overview diagram.

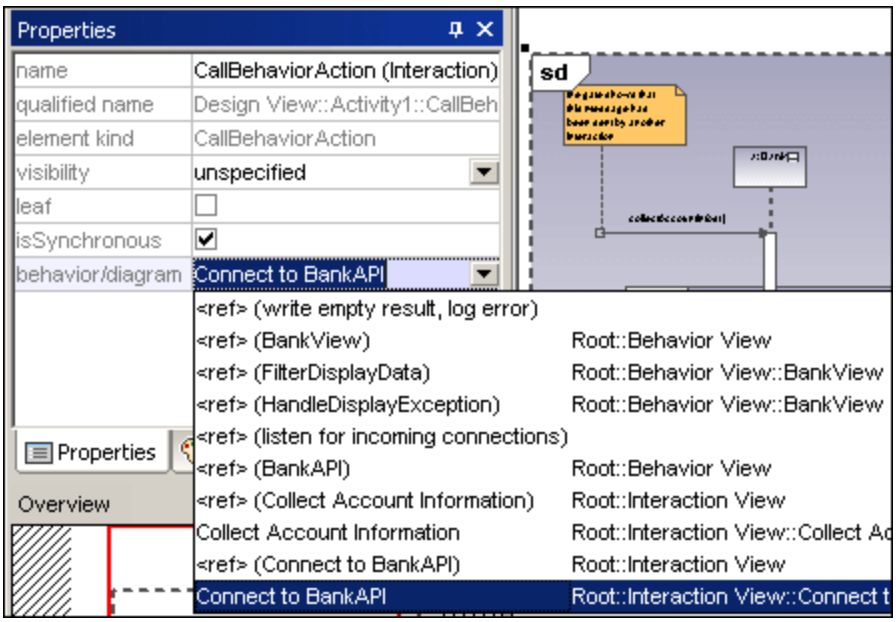
1. Locate the element you want to insert in the Model Tree tab (you can use the search function text box, or press CTRL + F, to search for any element).
2. Drag the element(s) into the diagram.

Inserting an Interaction element:

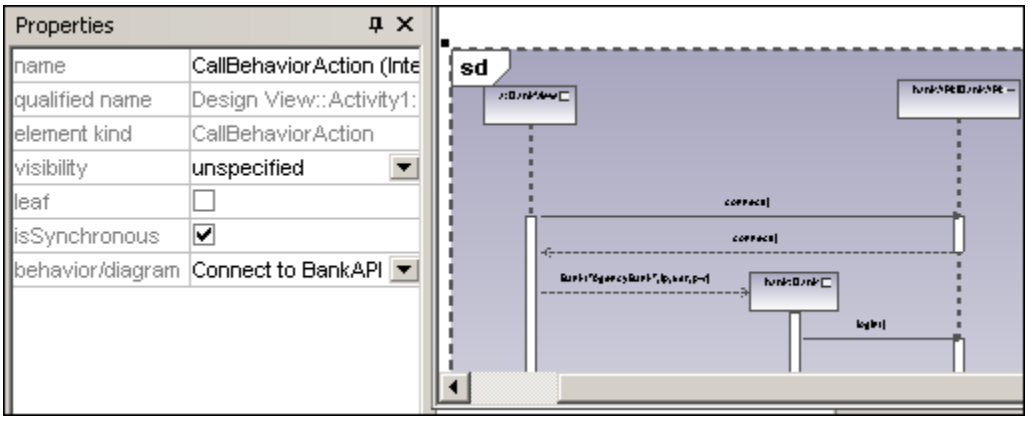
1. Click the CallBehaviorAction (Interaction) icon  in the icon bar, and click in the Interaction Overview diagram to insert it.



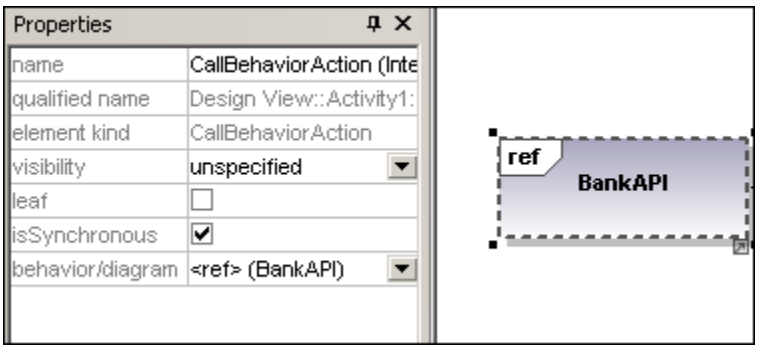
   The Collect Account Information sequence diagram is automatically inserted if you are using the Bank_Multilanguage.ump example file from the ...**UModelExamples** folder. The first sequence diagram, found in the model tree, is selected per default.
2. To change the default interaction element: Click the **behavior/diagram** combo box in the Properties tab.
   A list of all the possible elements that can be inserted is presented.



3. Click the element you want to insert to e.g. Connect to BankAPI.



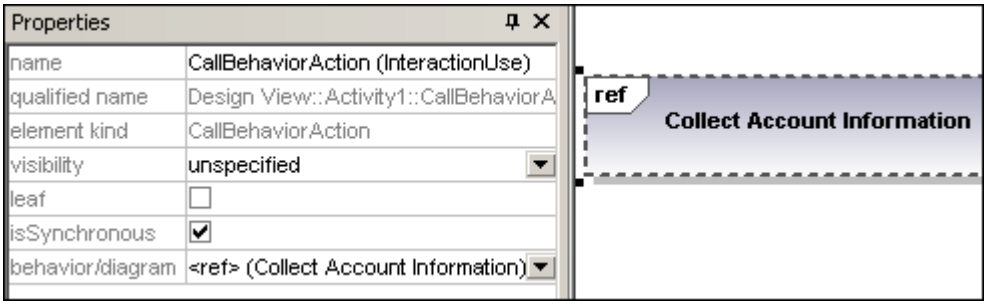
As this is also a sequence diagram, the Interaction element appears as an iconized version of the sequence diagram.
If you select **<ref> BankAPI**, then the Interaction element occurrence is displayed.



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 that 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" 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.

 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.

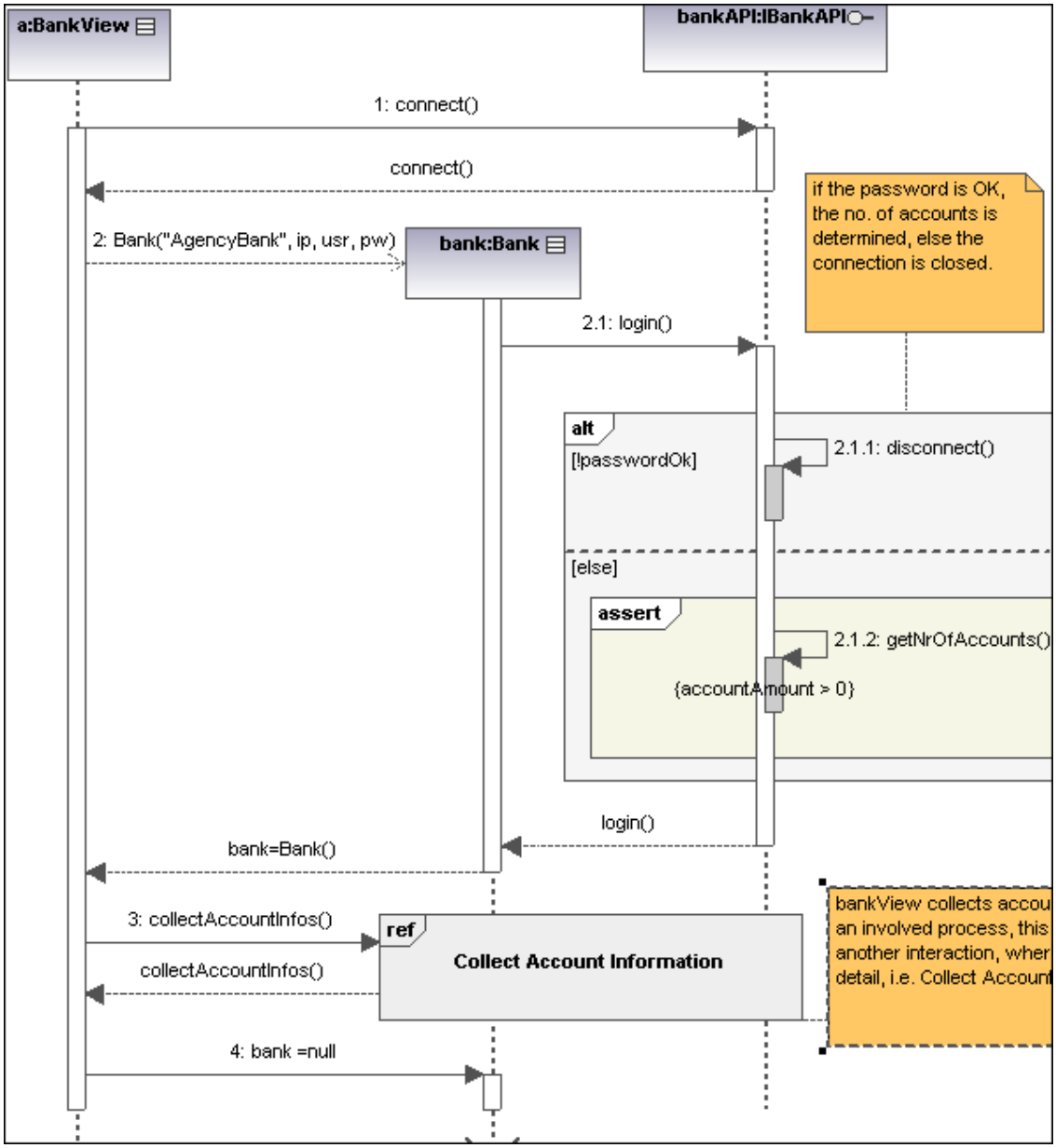
 Join Node (horizontal)
Inserts a horizontal Fork node.
A Join node synchronizes multiple flows defined by the Fork node.

 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.

9.1.6 Sequence Diagram

UModel supports the standard Sequence diagram defined by UML, and allows easy manipulation of objects and messages to model use case scenarios. Please note that the sequence diagrams shown in the following sections are only available in the **Bank_Java**.ump, **Bank_CSharp**.ump and **Bank_MultiLanguage**.ump samples, in the ...**UModelExamples** folder supplied with UModel.



Inserting sequence diagram elements

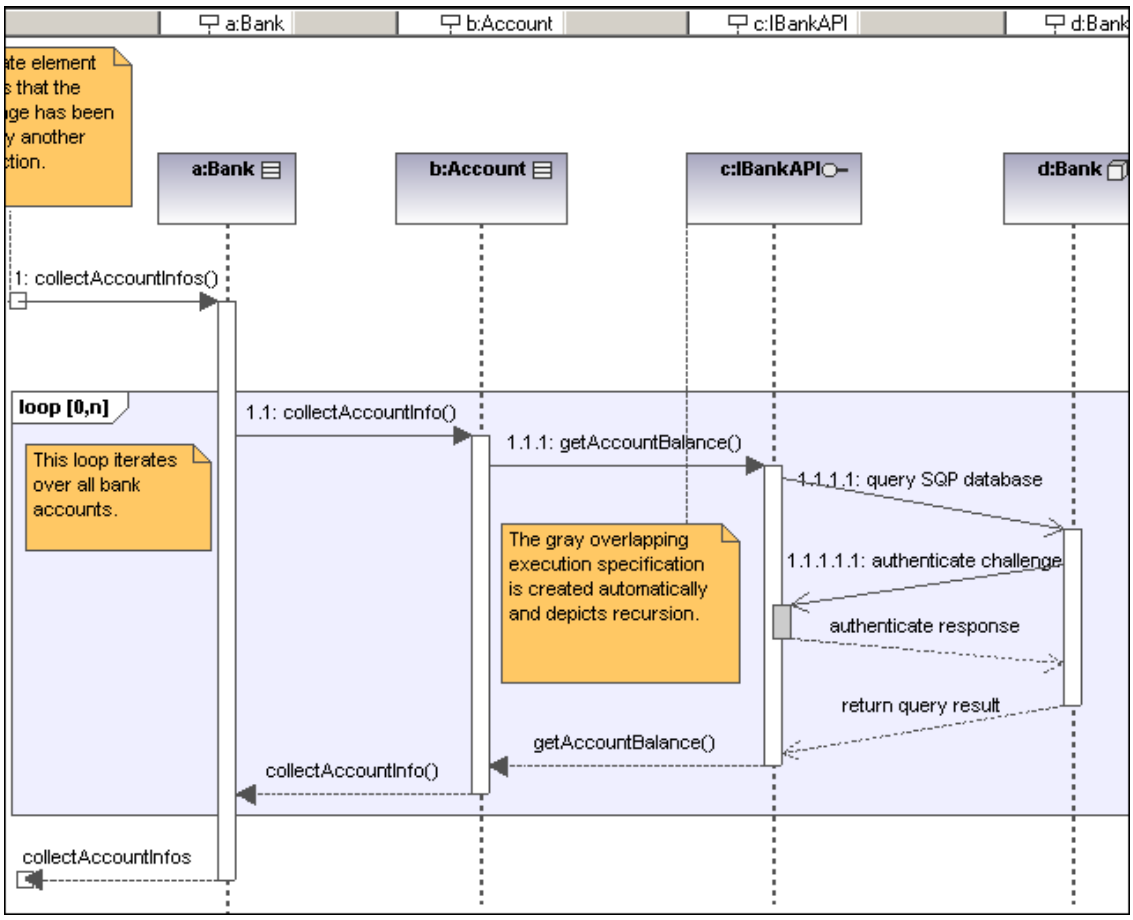
A sequence diagram models runtime dynamic object interactions, using messages. Sequence diagrams are generally used to explain individual use case scenarios.

- **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. Please see Combined Fragment for more information.

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.
   Note that holding down CTRL and clicking in the diagram tab, allows you to insert multiple elements of the type you selected.

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.

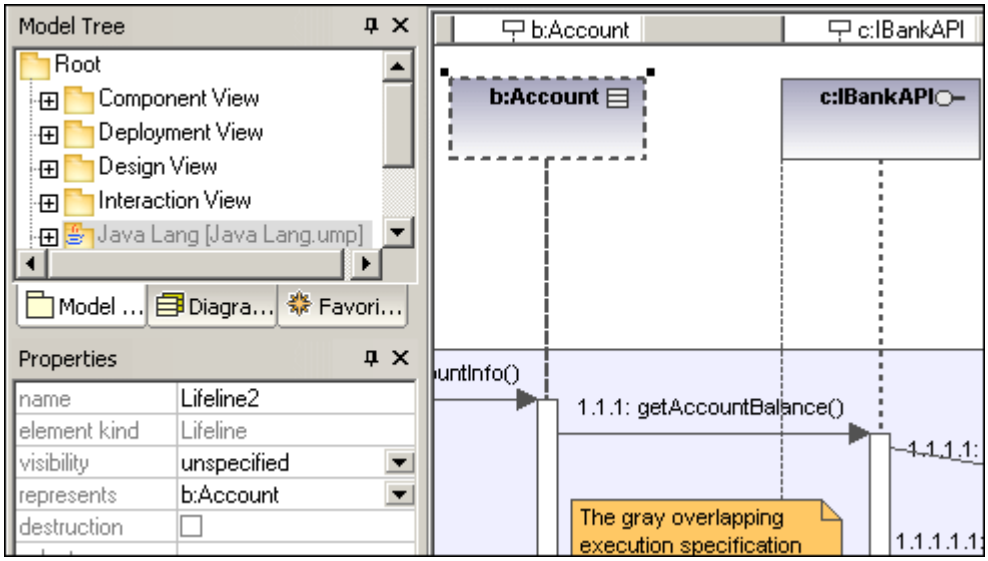
Lifeline

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.

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.



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.

Displaying/hiding activation boxes:

1. 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.

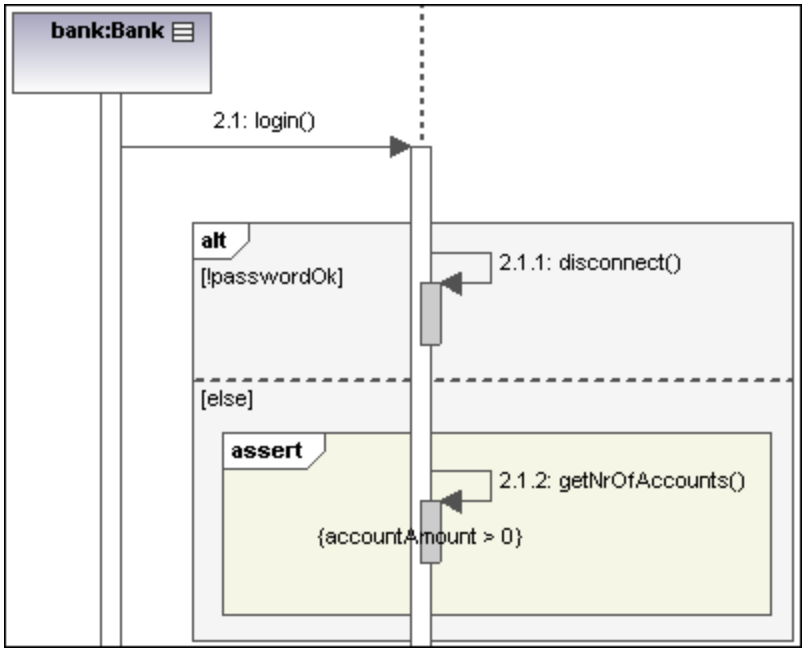
Lifeline attributes:
The **destruction** check box allows you to add a destruction marker, or stop, to the lifeline without having to use a destruction message.

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.

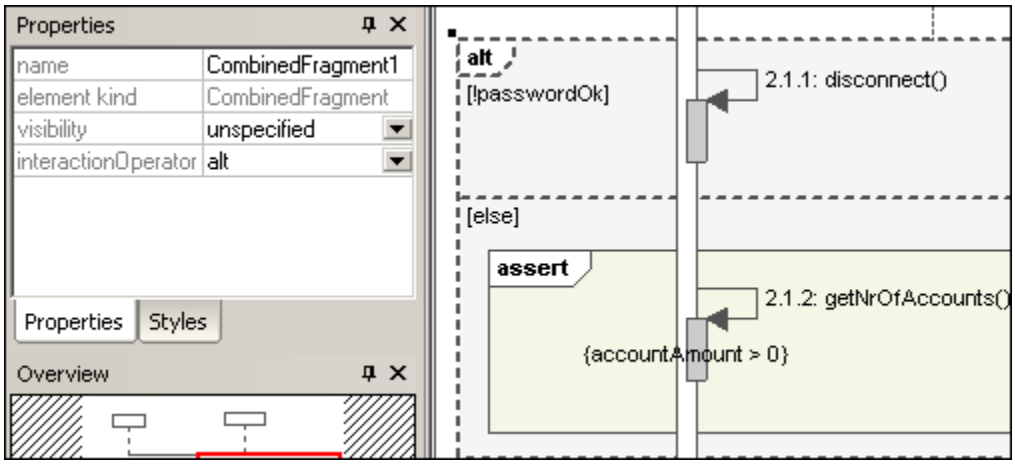
Combined Fragment

CombinedFragment

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.



InteractionOperators

Weak sequencing **seq** 

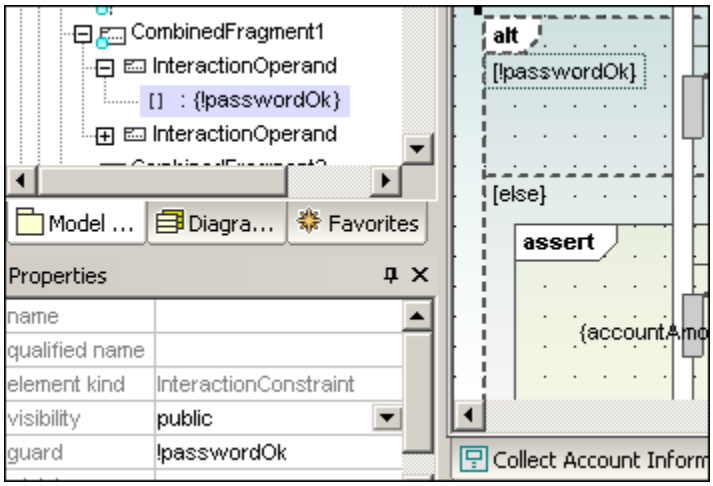
The combined fragment represents weak sequencing between the behaviours of the operands.

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.

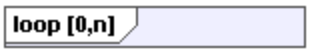
Break **break**
The break operator is chosen when the guard is true, the rest of the enclosing fragment is ignored.

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.

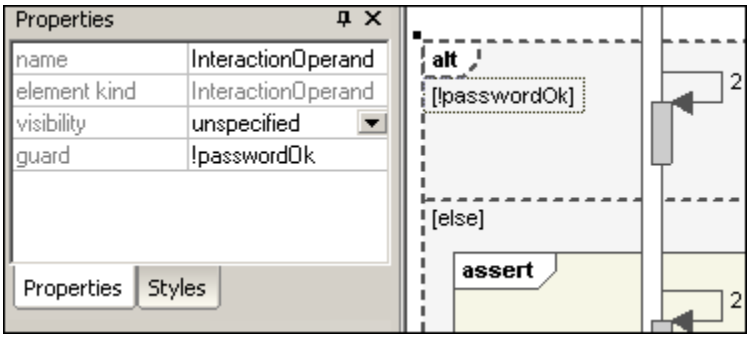
Assert **assert**
Designates the valid combined fragment, and its sequences. Often used in combination with consider, or ignore operands.

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 e.g. !passwordOK and press Enter to confirm.



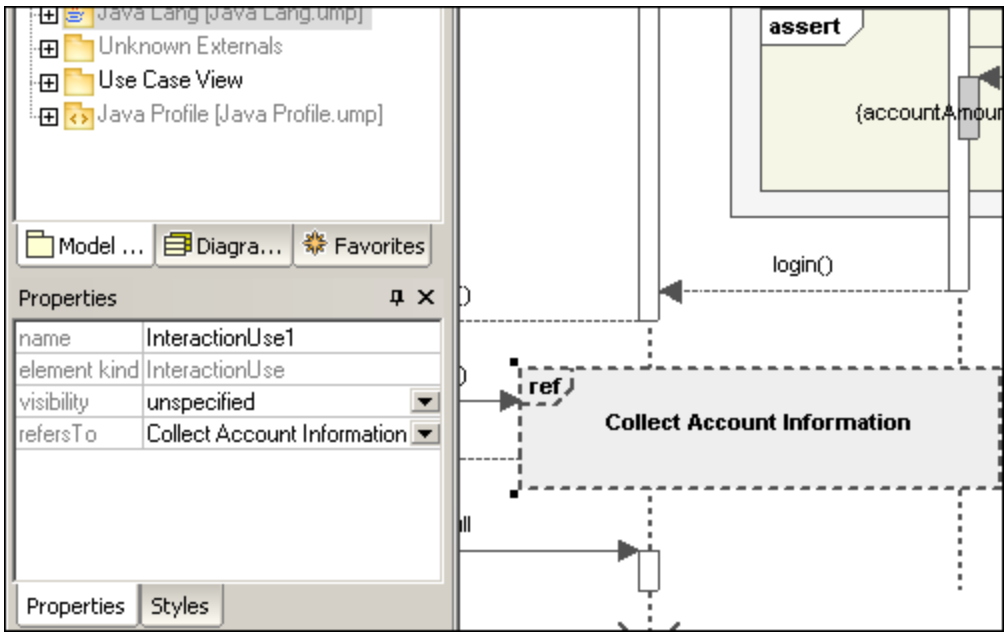
3. Use the same method to add the second interaction operand with the guard condition "else".
   Dashed lines separate the individual operands in the fragment.

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.

Interaction Use

InteractionUse
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.

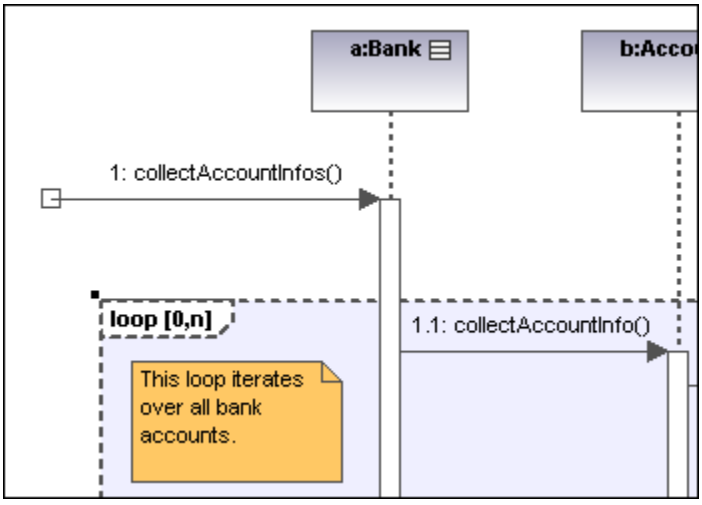
Please note:
You can also drag an existing Interaction Use element from the Model Tree into the diagram tab.

Gate

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.



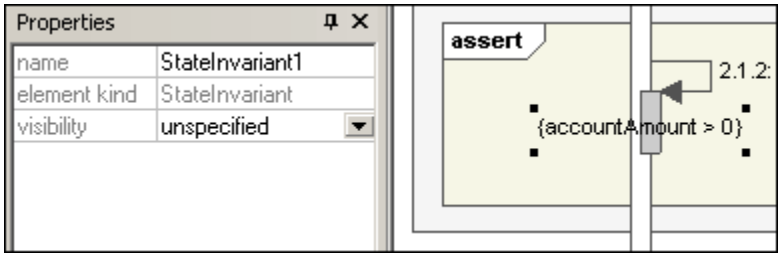
State Invariant

StateInvariant

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.



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.
- 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, 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.

To position dependent messages:

1. Click the respective message and drag vertically to reposition it.
   The default action when repositioning messages, is it to move all dependent messages related to the active one.

   Using CTRL+ click, allows you to select multiple messages.

To position messages individually:

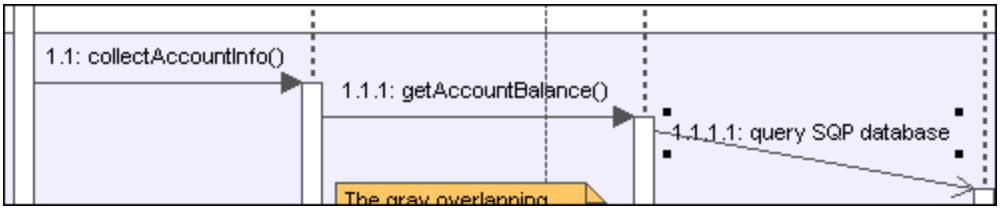
1. Click the "**Toggle dependent message movement**" icon  to deselect it.
2. Click the message you want to move and drag to move it.
   Only the selected message moves during dragging. You can position the message anywhere in the vertical axis between the object lifelines.

To automatically create reply messages:

1. Click the "**Toggle automatic creation of replies for messages**" icon  .
2. Create a new message betwween two lifelines.
   A reply message is automatically inserted for you.

Message numbering:
UModel supports different methods of message numbering: nested, simple and none.

- **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.



To select the message numbering scheme:
There are two methods of selecting the numbering scheme:

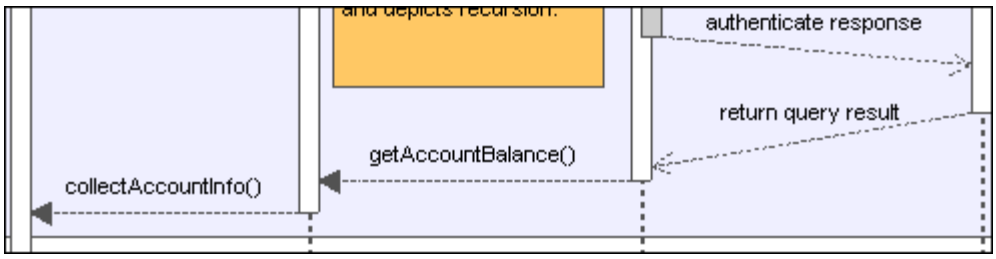
- Click the respective icon in the icon bar.
- Use the Styles tab to select the scheme.

To select the numbering scheme using the Styles tab:

1. Click the Styles tab and scroll down to the **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.

Please 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.

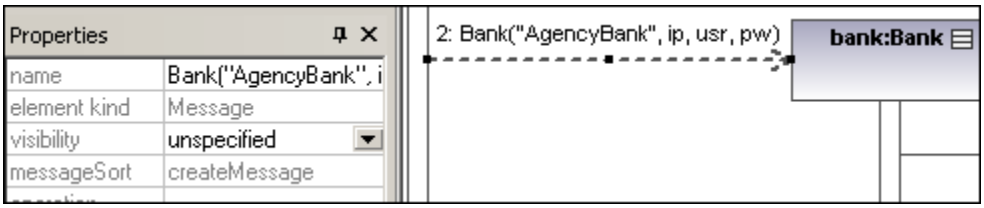
Message replies:
Message reply icons are available to create reply messages, and are displayed as dashed arrows.



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.

Creating objects with messages:

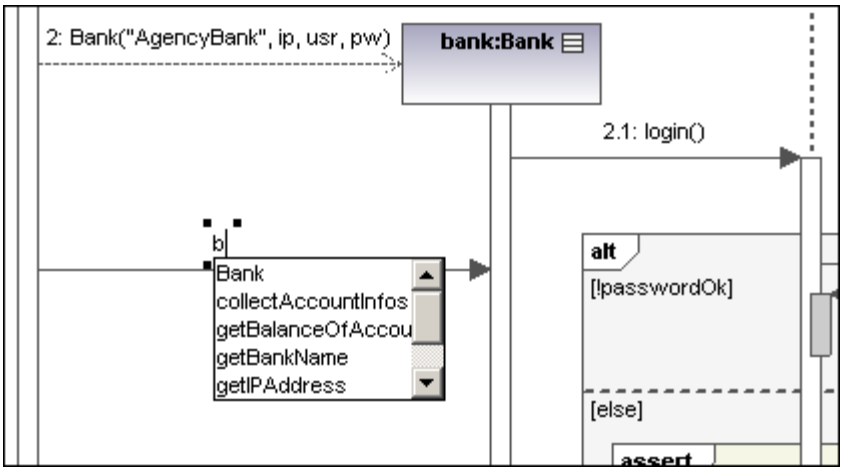
1. Messages can create new objects. This is achieved using the Message Creation icon  .
2. Drag the message arrow to the lifeline of an existing object to create that object. This type of message ends in the middle of an object rectangle, and often repositions the object box vertically.



Sending messages to specific class methods/operations in sequence diagrams

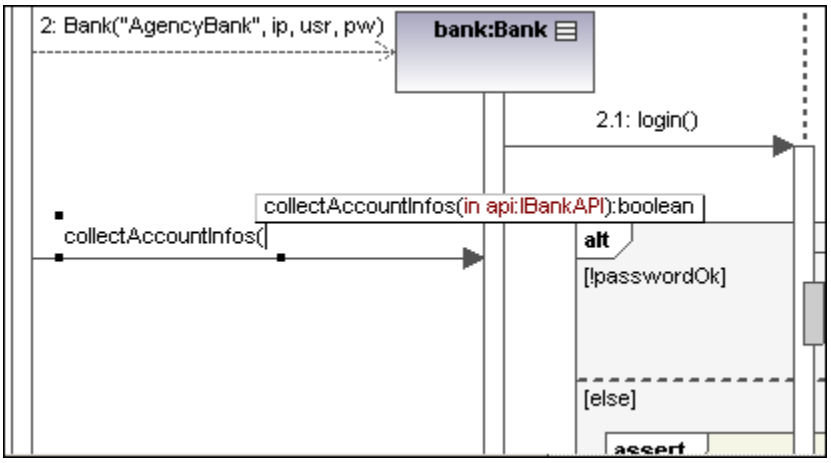
Having inserted a class from the Model Tree into a sequence diagram, you can then create a message from a lifeline to a specific method of the receiver class (lifeline) using UModel's syntax help and autocompletion functions.

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 spacebar and press Enter to select the parenthesis character that is automatically supplied.
   A syntax helper popup now appears, allowing you to enter the parameter correctly.



Message icons:

 Message (Call)

 Message (Reply)

 Message (Creation)

 Message (Destruction)

 Asynchronous Message (Call)

 Asynchronous Message (Reply)

 Asynchronous Message (Destruction)

 Toggle dependent message movement

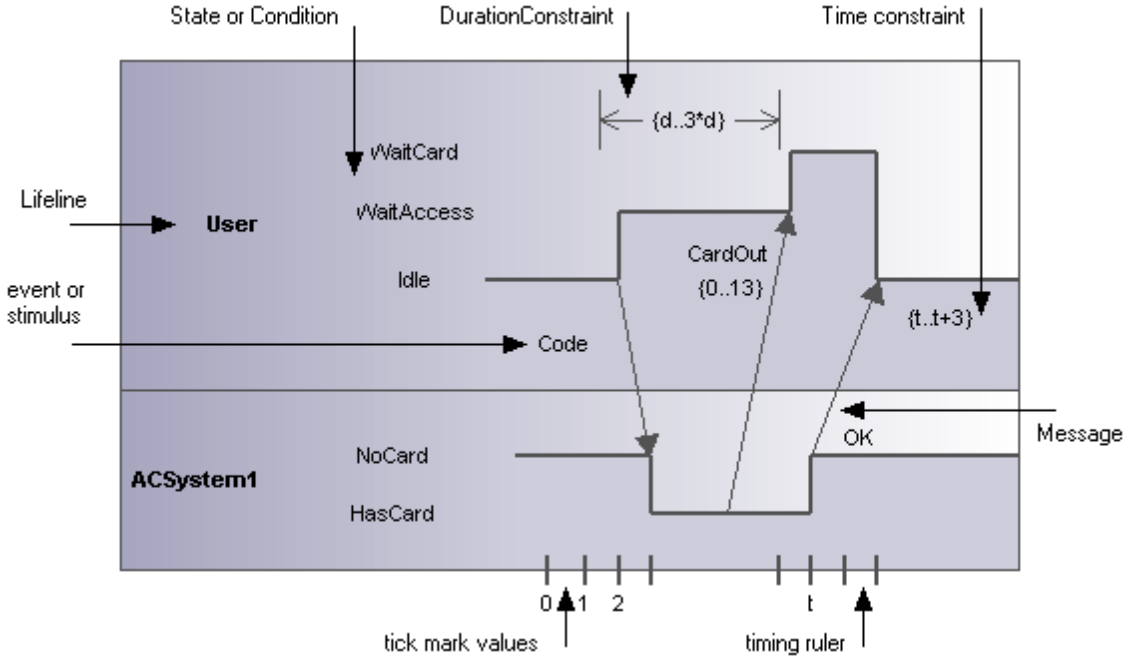
 Toggle automatic creation of replies for messages

9.1.7 Timing Diagram

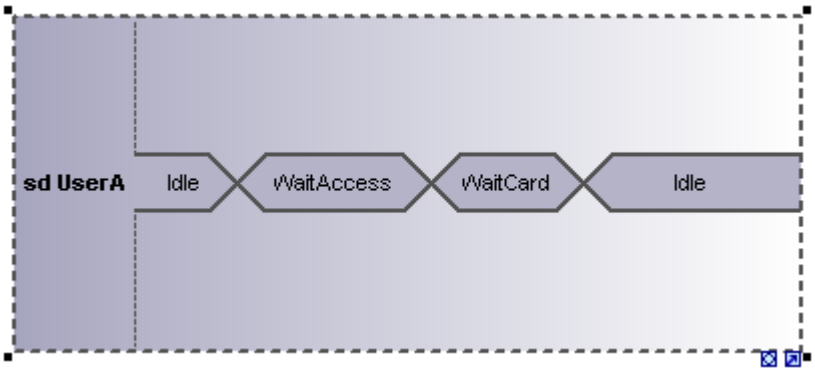
Timing diagrams depict the changes in state, or condition, of one or more interacting objects over a given period of time. States, or conditions, are displayed as timelines responding to message events, where a lifeline represents a Classifier Instance or Classifier Role.

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.



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.
   Note that holding down CTRL and clicking in the diagram tab, allows you to insert multiple elements of the type you selected.

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.

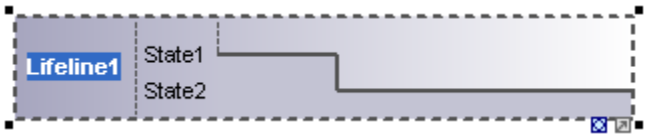
Lifeline

 or  **Lifeline**

The lifeline element is an individual participant in an interaction, and is available in two different representations: State/Condition timeline or General Value lifeline.

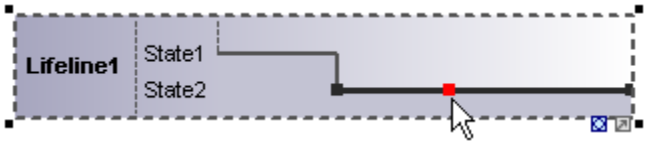
To insert a State Condition (StateInvariant) lifeline and define state changes:

1. Click the Lifeline (State/Condition) icon  in the title bar, then click in the Timing Diagram to insert it.

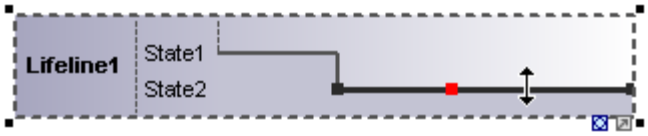


2. Enter the lifeline name to change it from the default name, Lifeline1, if necessary.
3. Place the mouse cursor over a section of one of the timelines and click left. This selects the line.
4. Move the mouse pointer to the position you want a state change to occur, and click again.

   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.



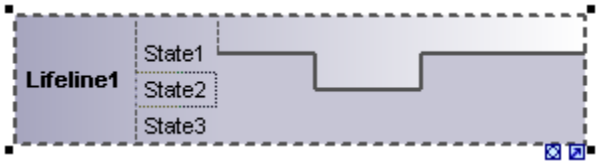
   Note that lines can only be moved between existing states of the current lifeline.

   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:

1. Right click the lifeline and select **New | State/Condition (StateInvariant)**.
   A new State e.g. State3 is added to the lifeline.



To move a state within a lifeline:

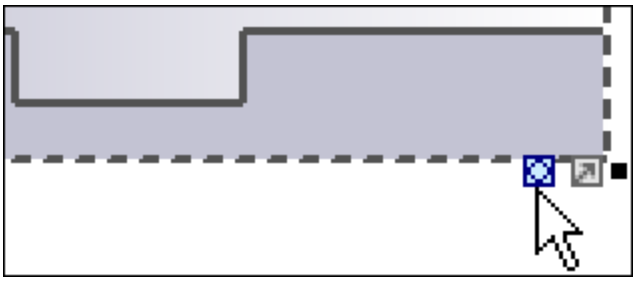
1. Click the state label that you want to move.
2. Drag it to a different position in the lifeline.

To delete a state from a lifeline:

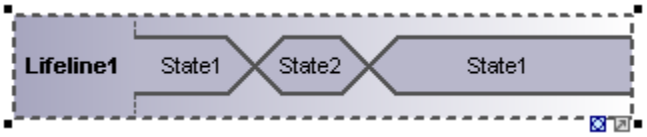
1. Click the state and press the Del. key, or alternatively, right click and select Delete.

To switch between timing diagram types:

1. Click the "toggle notation" icon at the bottom right of the lifeline.



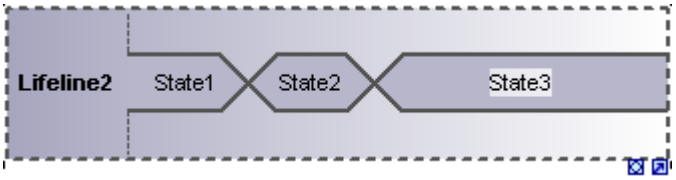
This changes the display to the General Value lifeline, the cross-over point represents a state/value change.



Please note that 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.

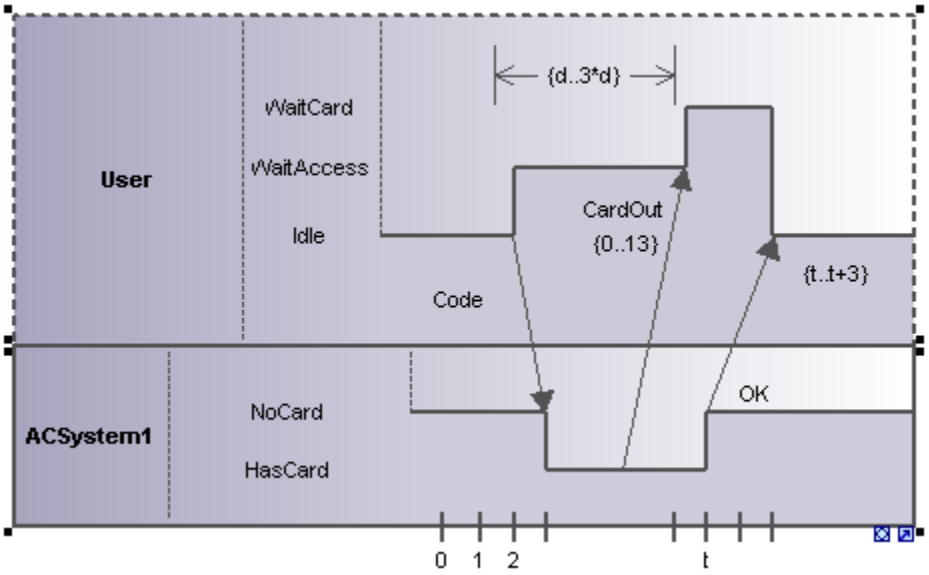


A new State is added to the lifeline.

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.



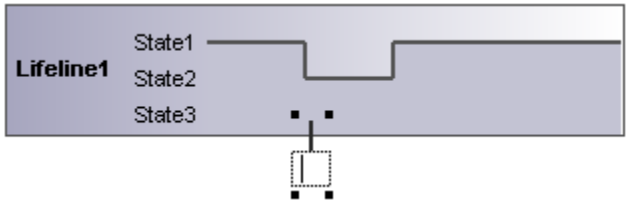
Tick Mark

TickMark

The tick mark is used to insert the tick marks of a timing ruler scale onto a lifeline.

To insert a TickMark:

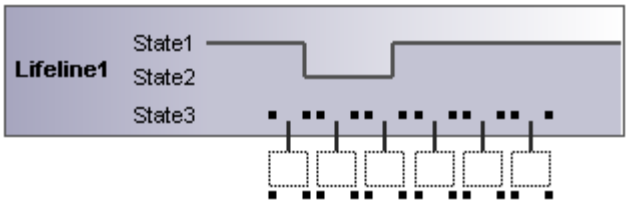
1. Click the tick mark icon and click on the lifeline to insert it.



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.

To evenly space tick marks on a lifeline:

1. Use the marque, by dragging in the main window, to mark the individual tick marks.
2. Click the **Space Across** icon in the icon bar.



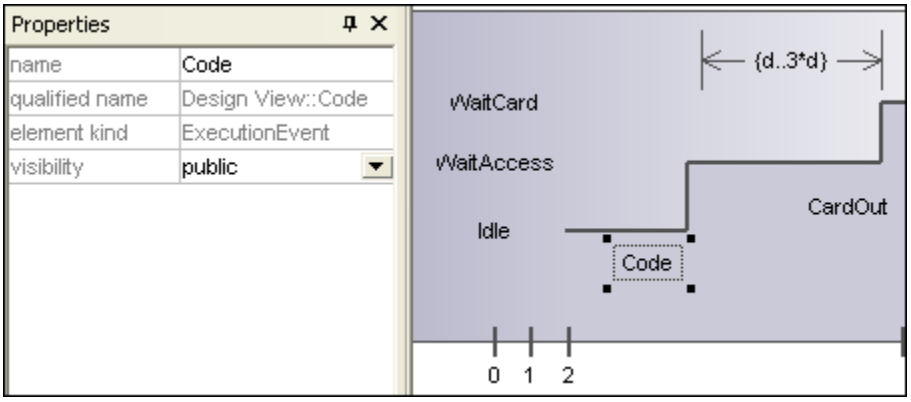
Event/Stimulus

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.

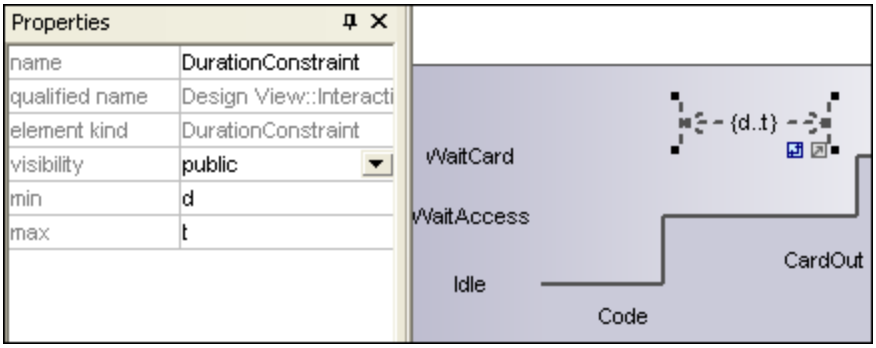
DurationConstraint

DurationConstraint

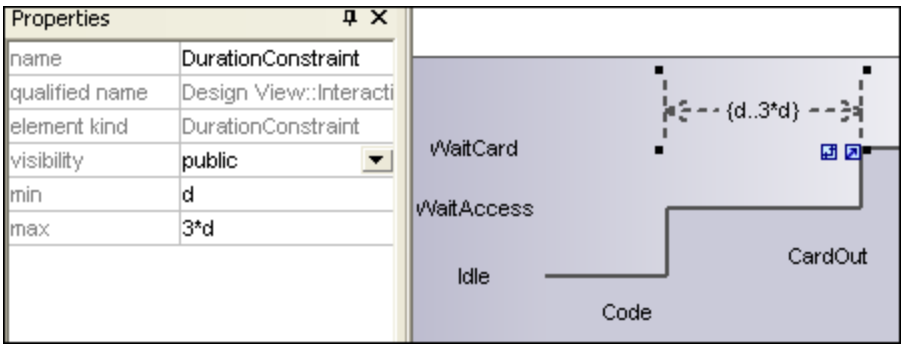
A Duration defines a ValueSpecification that denotes a duration in time between a start and endpoint. A duration is often an expression representing the number of clock ticks, which may elapse during this duration.

To insert an DurationConstraint:

1. Click the DurationConstraint icon, then click the specific position on the lifeline where the constraint is to be displayed.

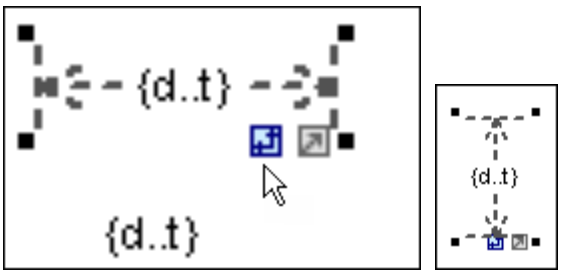


   The default minimum and maximum values, "d..t", are automatically supplied. These values can be edited by double clicking the time constraint, or by editing the values in the Properties window.
2. Use the "handles" to resize the object if necessary.



Changing the orientation of the DurationConstraint:

1. Click the "Flip" icon to orient the constraint vertically.



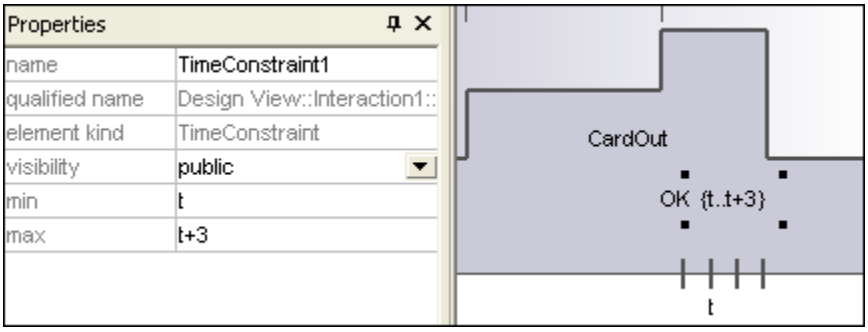
TimeConstraint

TimeConstraint

A TimeConstraint is generally shown as graphical association between a TimeInterval and the construct that it constrains. Typically this graphical association between an EventOccurrence and a TimeInterval.

To insert a TimeConstraint:

1. 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.

Message

 **Message (Call)**  **Message (Reply)**  **Async message (Call)**

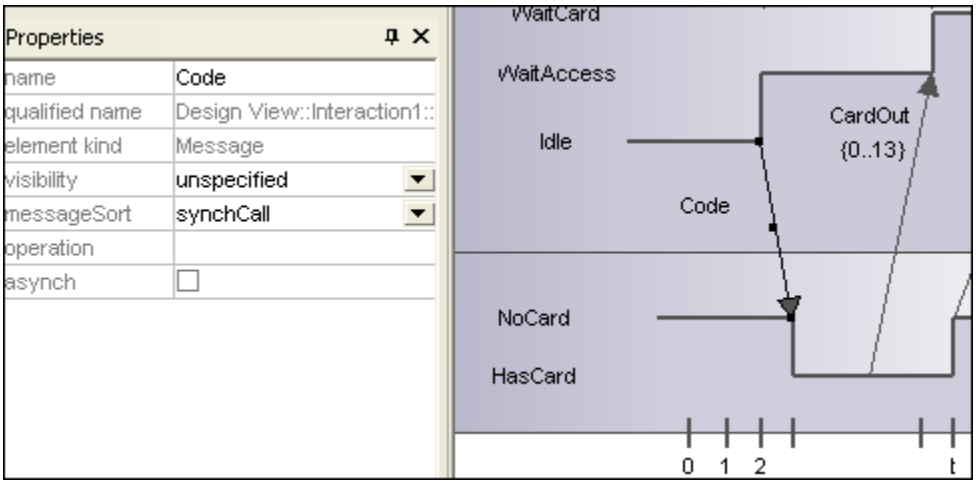
A Message is a modeling element that defines a specific kind of communication in an Interaction. A communication can be e.g. raising a signal, invoking an Operation, creating or destroying an Instance. The Message specifies the type of communication defined by the dispatching ExecutionSpecification, as well as the sender and the receiver.

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.



- 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.

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".

9.2 Structural Diagrams

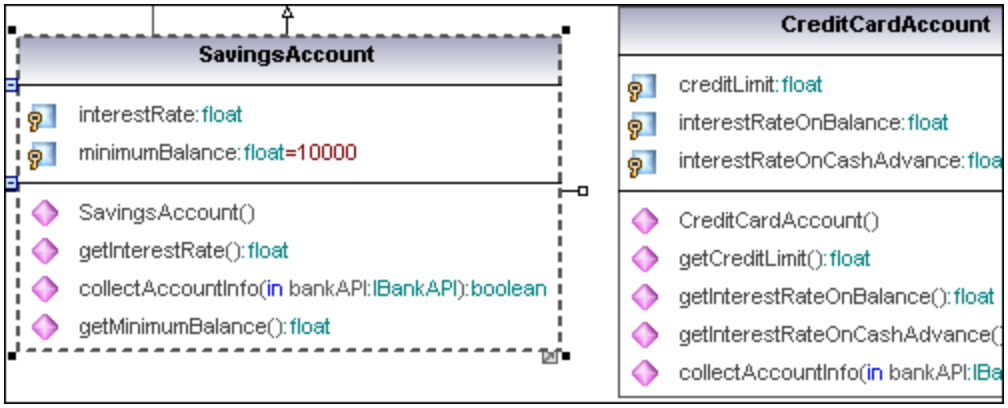
These diagrams depict the structural elements that make up a system or function. Both the static, e.g. Class diagram, and dynamic, e.g. Object diagram, relationships are presented.

Structural Diagrams

- Class Diagram
- Component Diagram
- Composite Structure Diagram
- Deployment Diagram
- Object Diagram
- Package Diagram

9.2.1 Class Diagram

Please see the Class Diagrams section in the tutorial for more information on how to add classes to a diagram.



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.

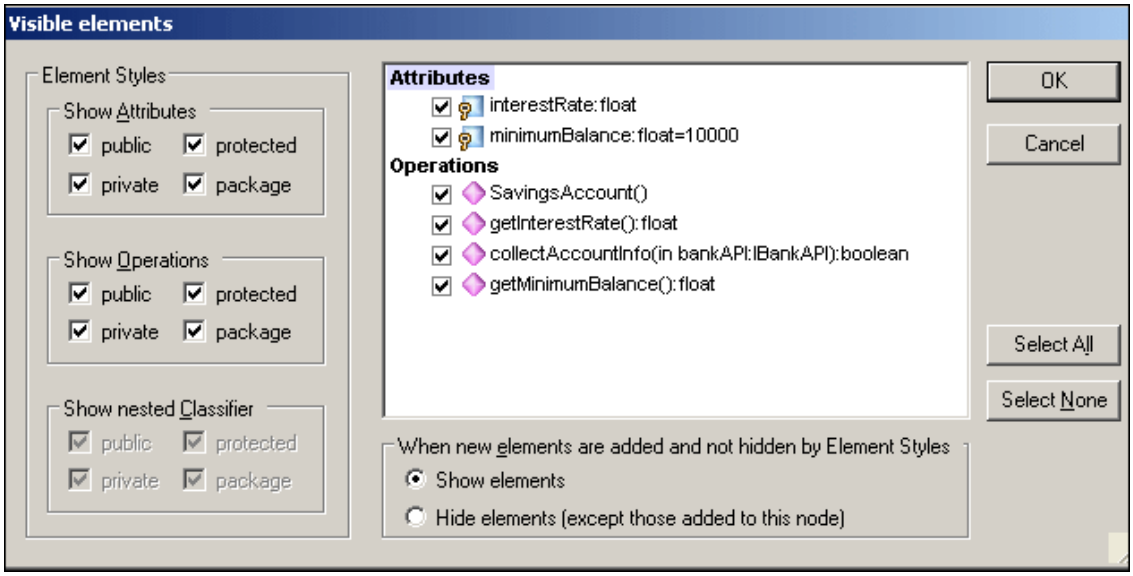
Expanding / collapsing class compartments in the Model Tree:
In the Model Tree classes are subelements of packages and you can affect either the packages or the classes.

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.

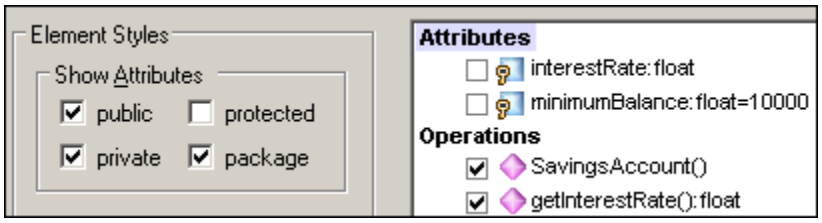
To **collapse** the packages/classes, press the **-** keyboard key.
Note that you can use the standard keyboard keys, or the numeric keypad keys to achieve this.

Showing / Hiding class attributes or operations
UModel now allows you to individually display the attributes or operations of a class, as well as define which should be shown when adding them as new elements.

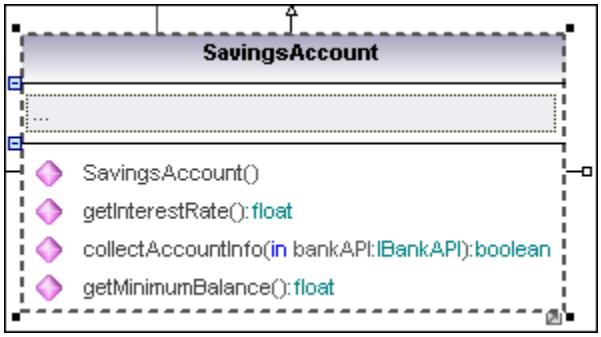
Right click a class, e.g. SavingsAccount, and select the menu option **Show/Hide Node content**
.



Deselecting the **protected** checkbox in the Show Attributes group, deselects the protected attributes in the preview window.



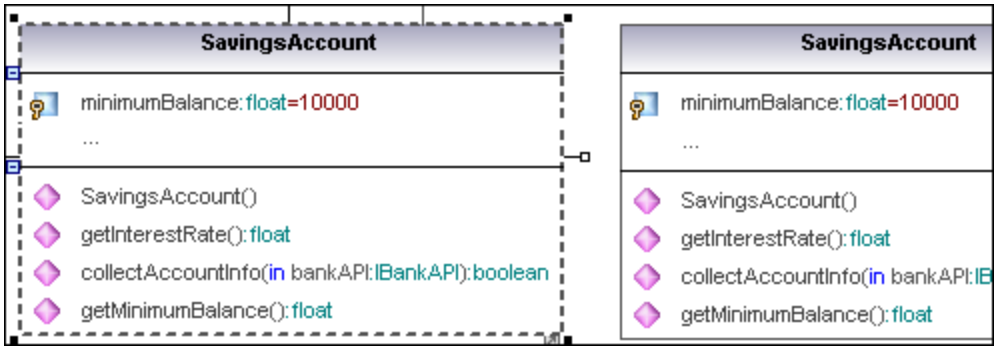
Having confirmed with OK, the protected attributes in the class are replaced with ellipsis "...". Double clicking the ellipsis opens the dialog box.



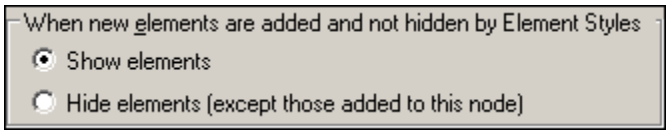
Note that individual attributes can be affected by only deselecting the check box in the preview window.

Showing / Hiding class attributes or operations - Element styles

UModel allows you to insert **multiple instances** of the same class on a **single** diagram, or even different diagrams. The visibility settings can be individually defined for each of these "views" to the class. The screenshot below shows two views to the same class i.e. SavingsAccount.



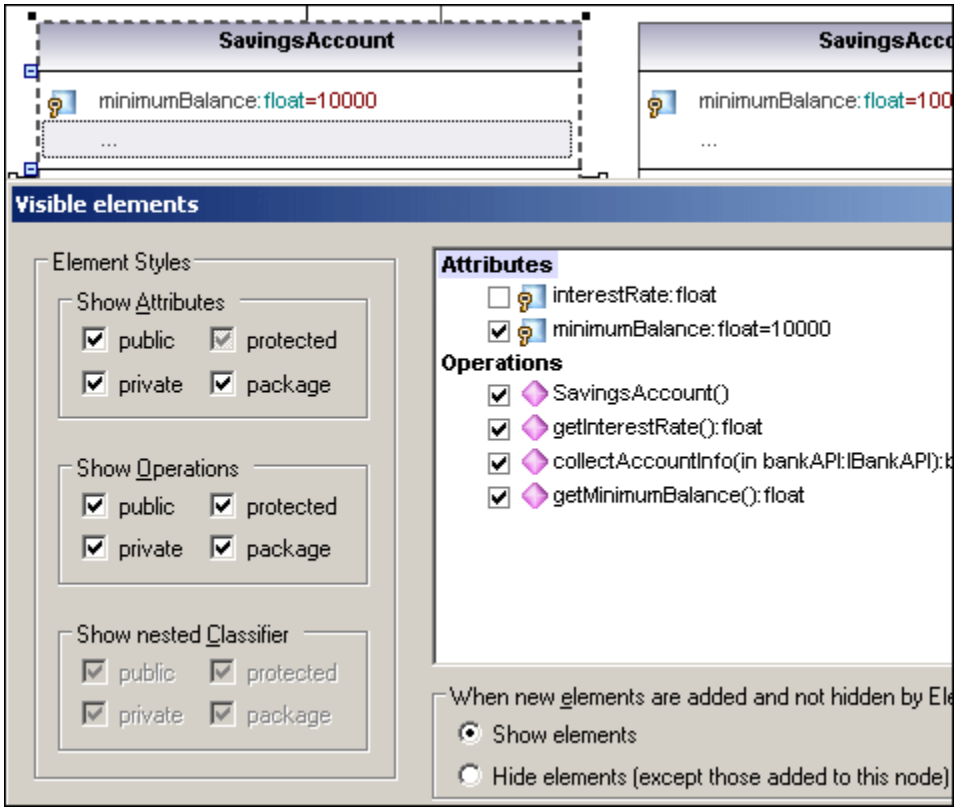
The "When new elements are added and not hidden by Element Styles" option allows you to define what will be made visible when new elements are added to the class. Elements can be added manually in the model diagram and in the Model Tree, or automatically during the code engineering process.



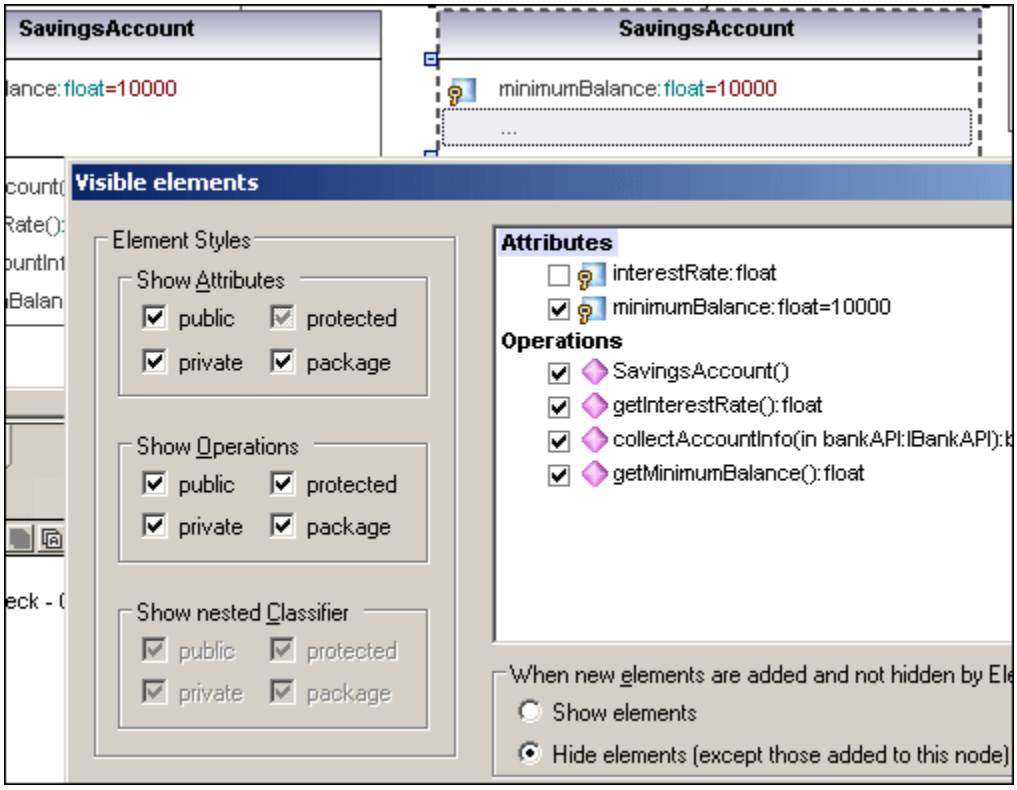
Show elements: displays all new elements that are added to any view of the class.

E.g. The interestRate:float attribute has been hidden in both "views" of SavingsAccount, leaving the minimumBalance attribute visible. The "Show elements" radio button is active for the left-hand class.

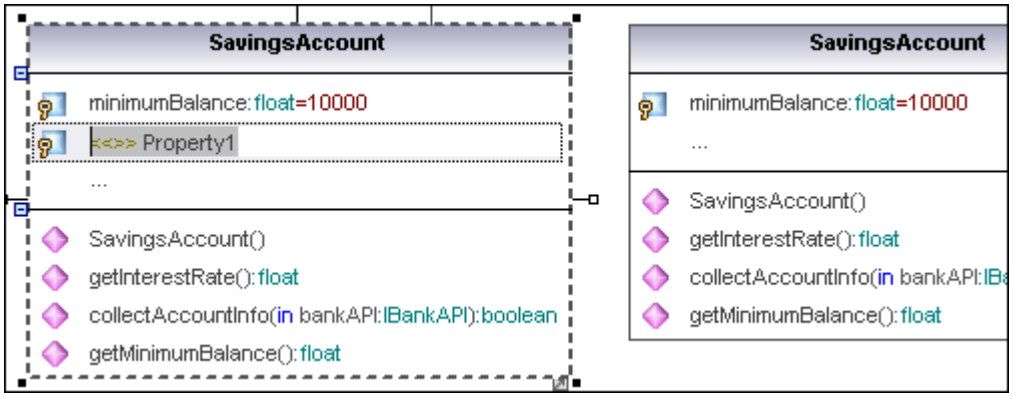
Double clicking the ellipsis "..." in the attribute compartment of the **left**-hand class shows that the "Show elements" radio button is active.



Double clicking the ellipsis "..." in the attribute compartment of the **right**-hand class shows that the "Hide elements (except those added to this node)" radio button is active.

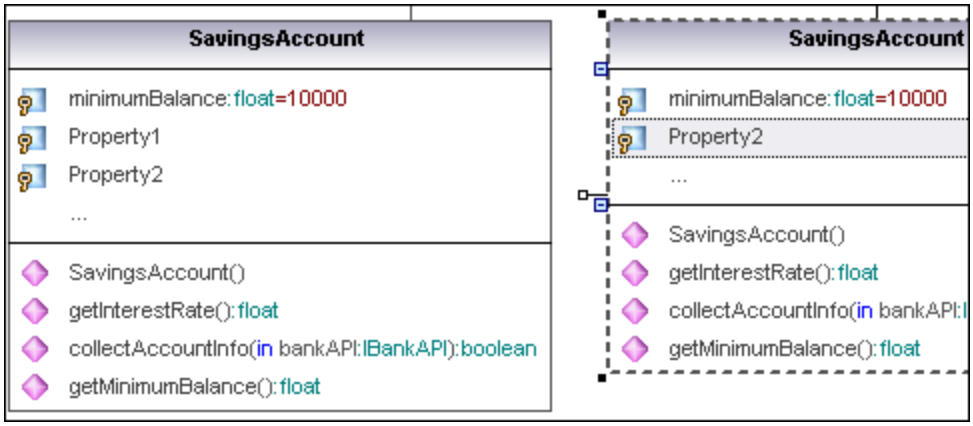


Clicking the **left**-hand class and pressing F7, (or clicking the class in the Model Tree and pressing F7) adds a new attribute (Property1) to the class.



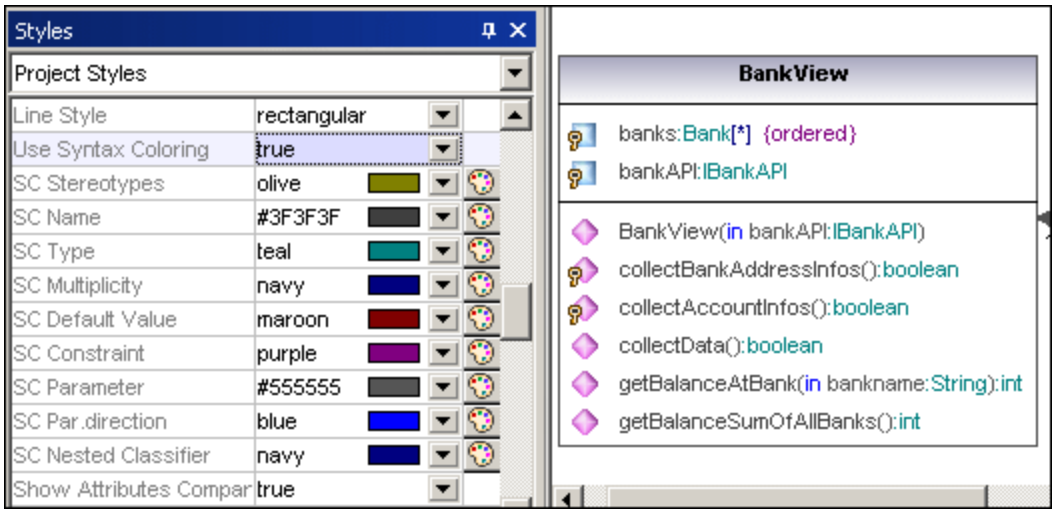
The new element is only visible in the left-hand class, because "Show elements" is set as active. The right-hand class setting is "Hide elements...", so the new element is not shown there.

Clicking the **right**-hand class and pressing F7 adds a new attribute (Property2) to the class. This new attribute is now visible because the Hide elements... setting has the qualifier "**except those added to this node**", where "node" generically means this class, or modelling element.



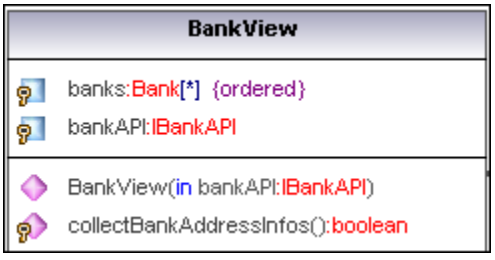
The Property2 attribute is also visible in the left hand class, because the setting there is "Show elements"

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.



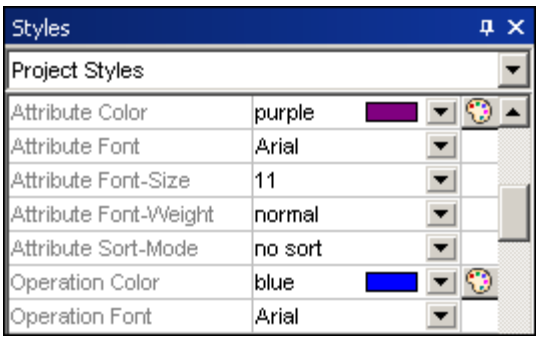
To change the default syntax coloring options (shown below):

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.



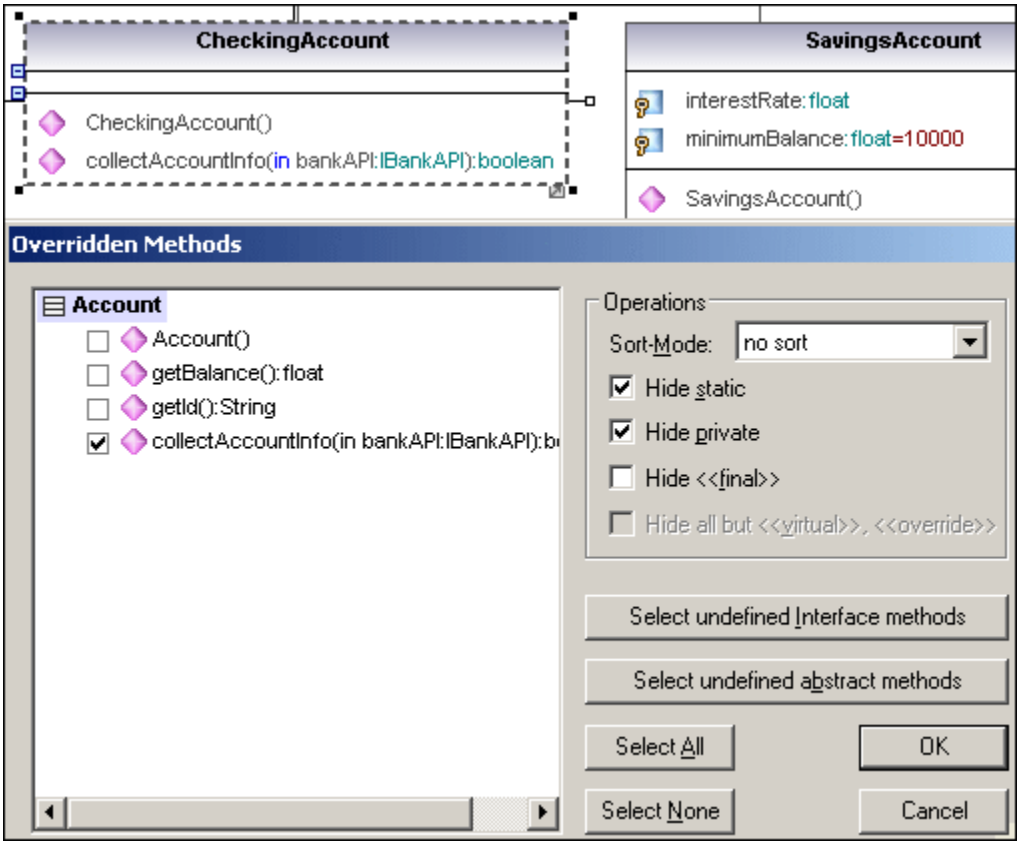
To disable syntax coloring:

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.



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**.



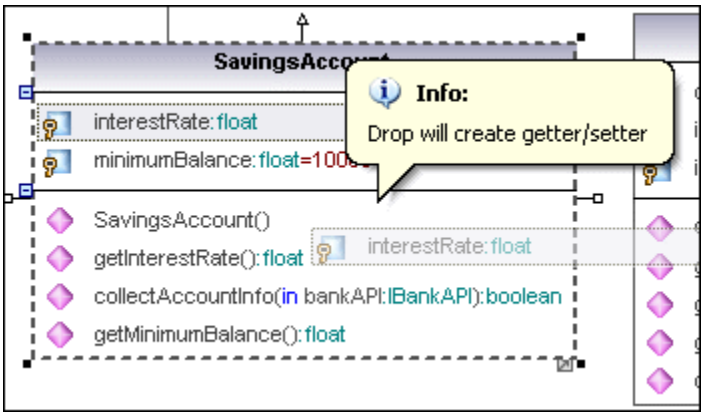
2. Select the Operations that you want to override and confirm with OK.

Creating getter / 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:

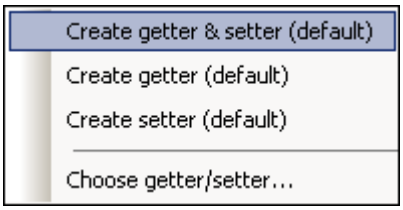
- Drag and drop an attribute into the operation compartment
- Use the context menu to open a dialog box allowing you to manage get/set methods

To create getter/setter methods using drag and drop:

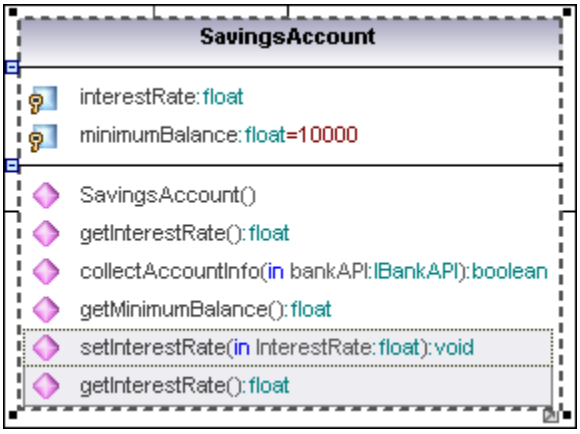
1. Drag an attribute from the Attribute compartment and drop it in the Operations compartment.



A popup appears at this point allowing you to decide what type of get/set method you want to create.

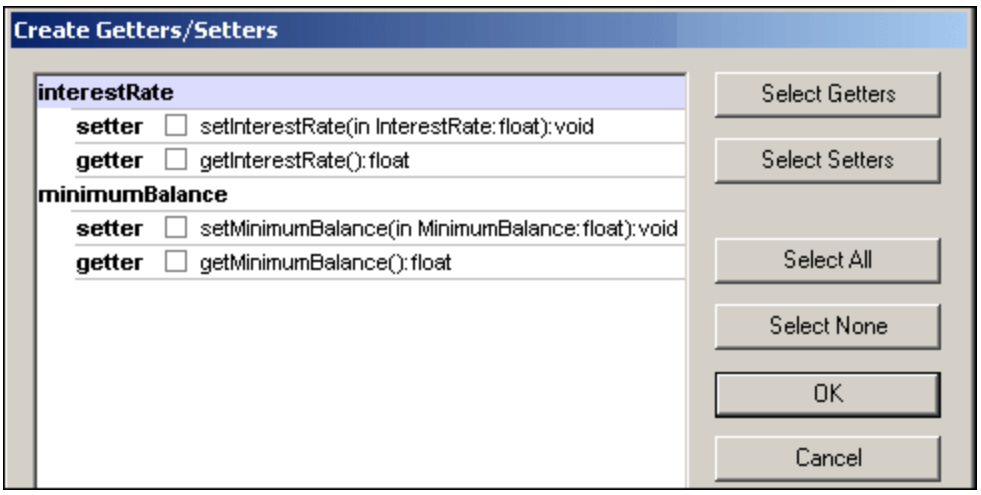


Selecting the first item creates a get and set method for interestRate:float.



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.

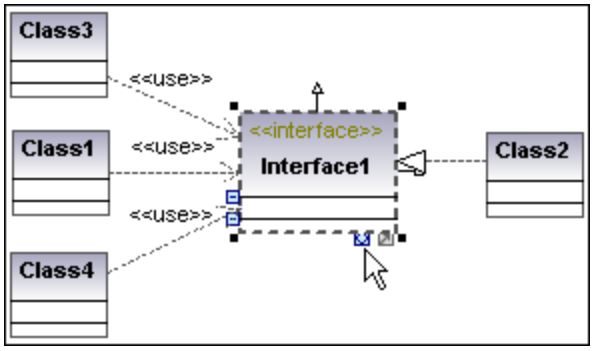
2. Use the buttons to select the items as a group, or click the getter/setter check boxes individually.

Please note:

You can also right click a single attribute and use the same method to create an operation for it.

Ball and socket notation

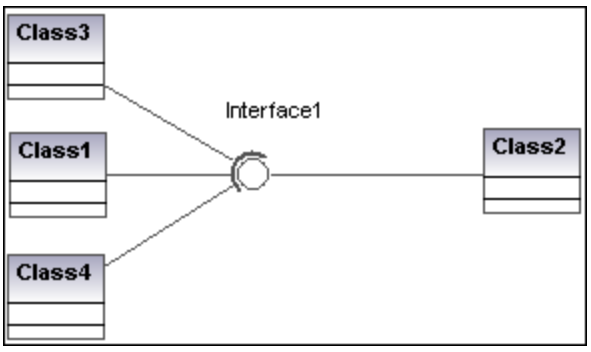
UModel now supports the ball and socket notation of UML 2.0. Classes that require an interface, display a "socket" and the interface name, while classes that implement an interface display the "ball".



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.

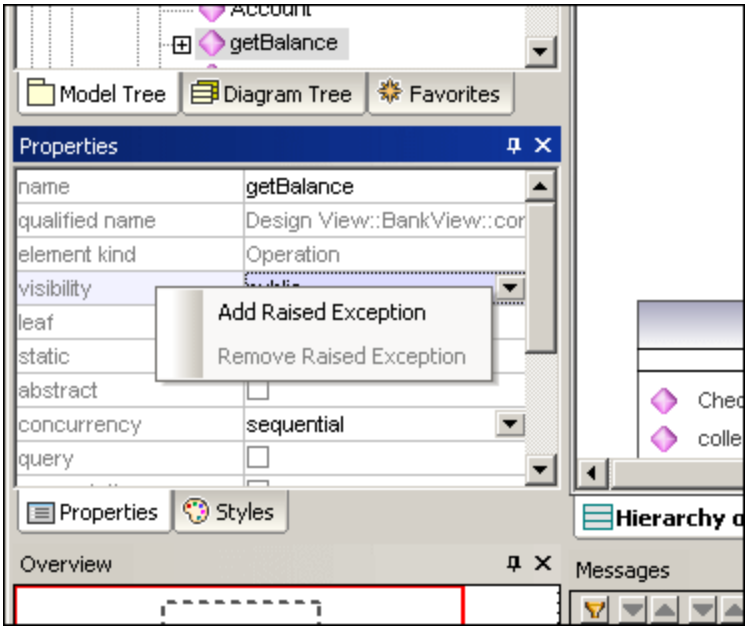
To switch between the standard and ball-and-socket view:

- Click the Toggle Interface notation icon at the base of the interface element.



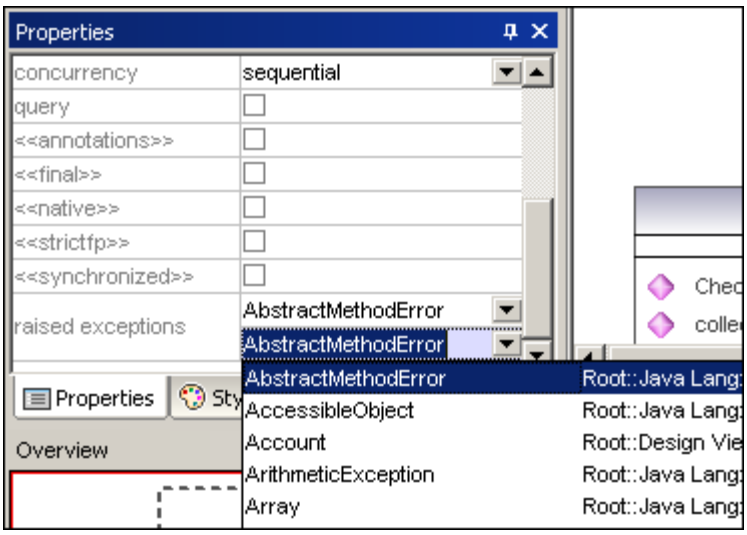
Adding Raised Exceptions to methods of a class

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 in the Properties window and select **Add Raised Exception** from the popup menu.



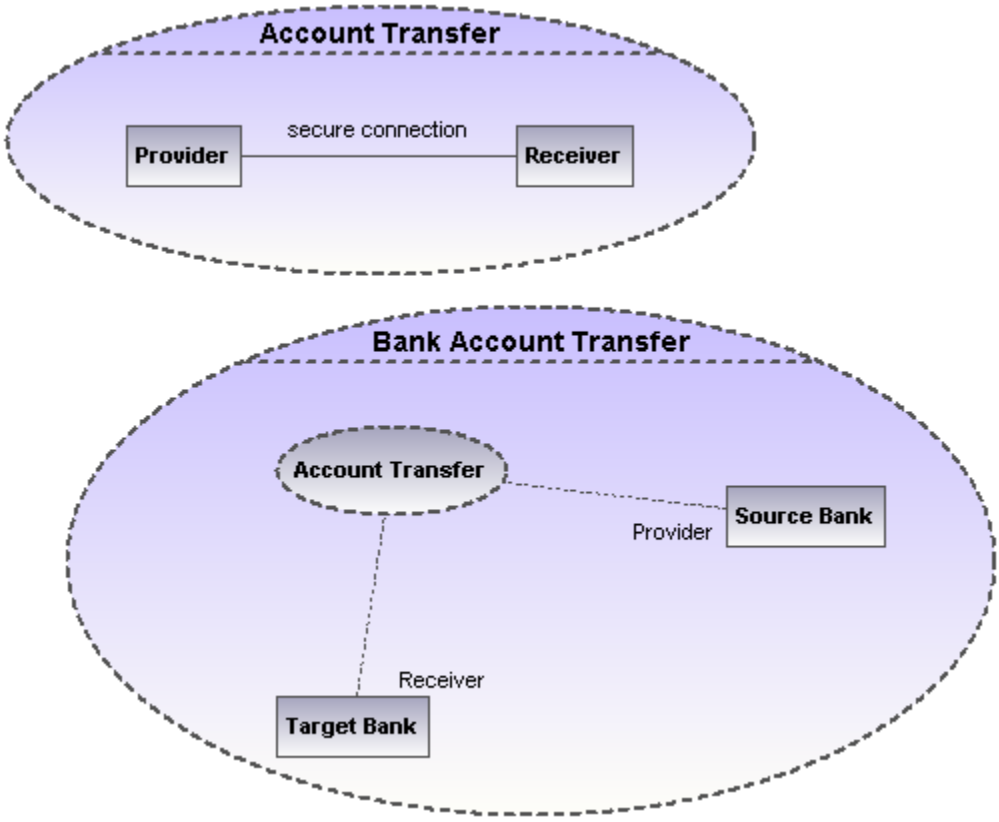
This adds the raised exceptions field to the Properties window, and automatically selects the first entry in the popup menu.

3. Select an entry from the popup, or enter your own into the field.


Properties
concurrency
sequential
query
<<annotations>>
<<final>>
<<native>>
<<strictfp>>
<<synchronized>>
raised exceptions
AbstractMethodError
AbstractMethodError
AbstractMethodError
Root::Java Lang
AccessibleObject
Root::Java Lang
Account
Root::Design Vie
ArithmeticException
Root::Java Lang
Array
Root::Java Lang
Properties
Overview

9.2.2 Composite Structure Diagram

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.



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.
   Note that holding down CTRL and clicking in the diagram tab, allows you to insert multiple elements of the type you selected.

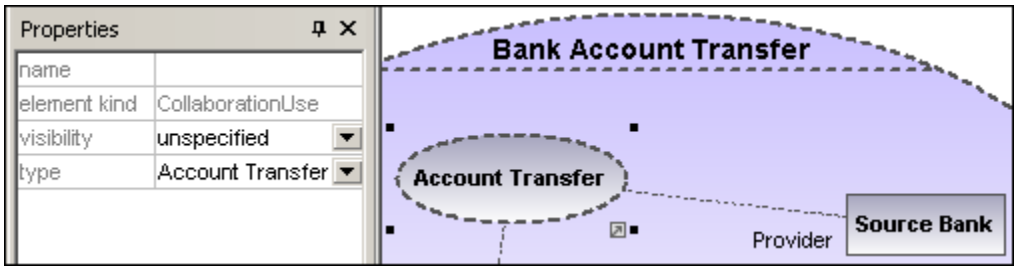
Dragging existing elements into the Composite Structure diagram:

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.


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.

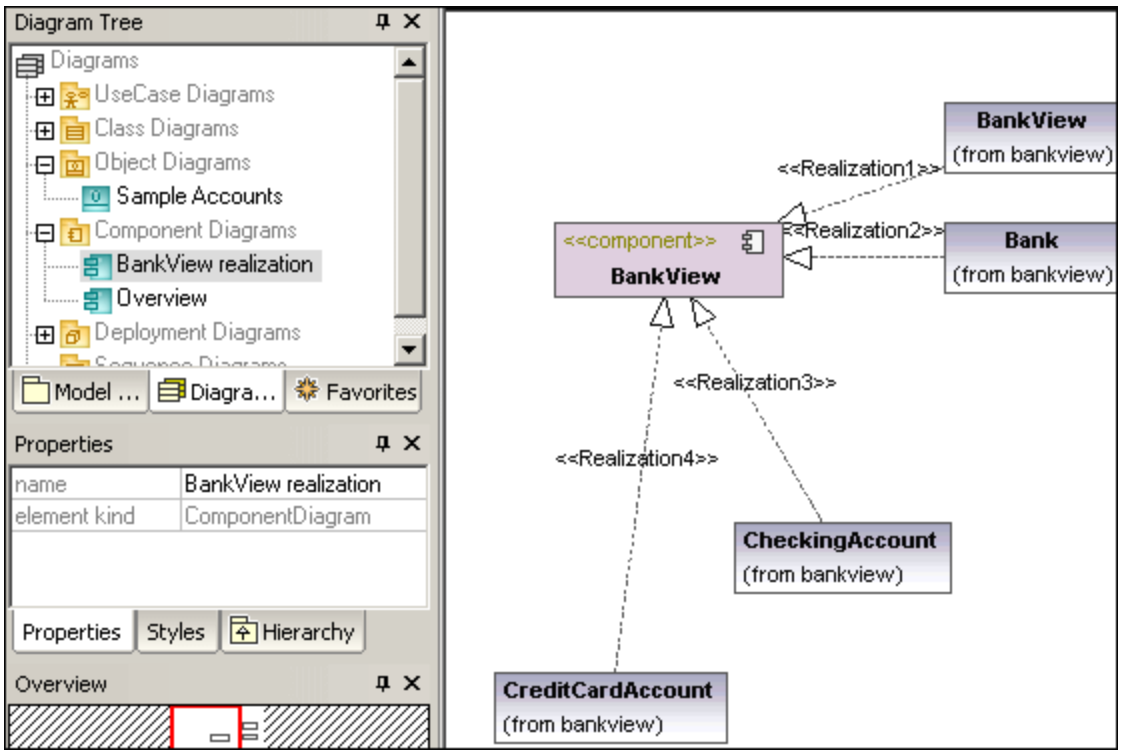

Class
Inserts a Class element, which is the actual classifier that occurs in that particular use of the collaboration.


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.

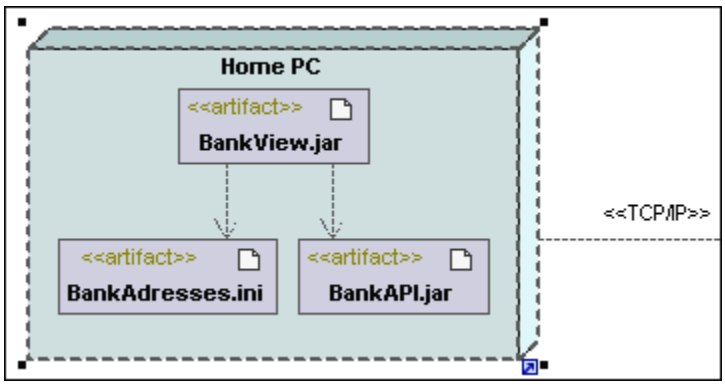
9.2.3 Component Diagram

Please see the Component Diagrams section in the tutorial for more information on how to add component elements to the diagram.



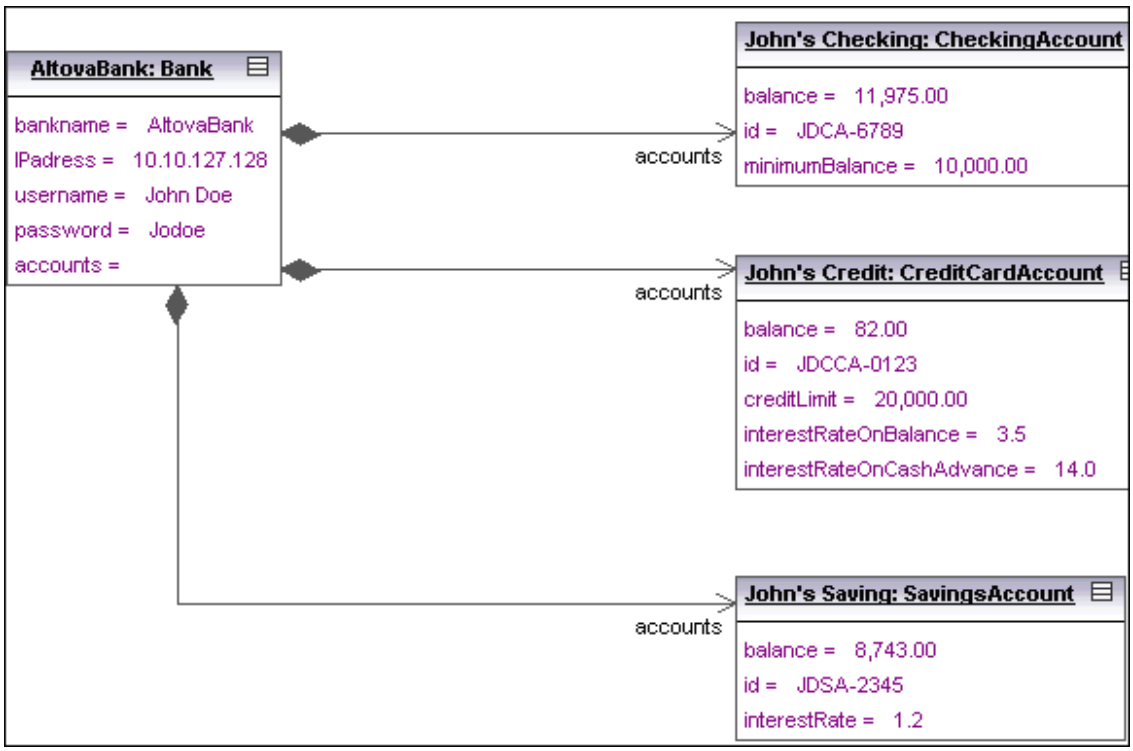
9.2.4 Deployment Diagram

Please see the Deployment Diagrams section in the tutorial for more information on how to add nodes and artifacts to the diagram.



9.2.5 Object Diagram

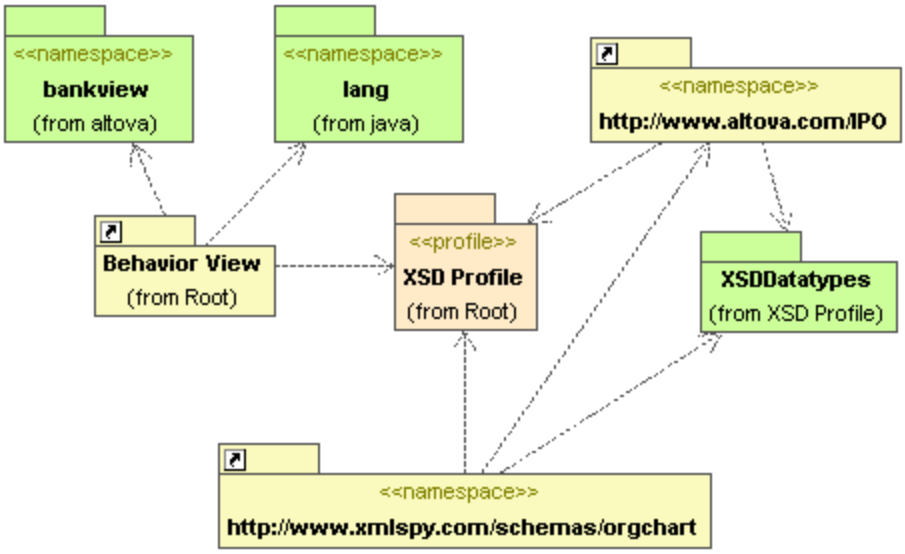
Please see the Object Diagrams section in the tutorial for more information on how to add new objects/instances to the diagram.



9.2.6 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.

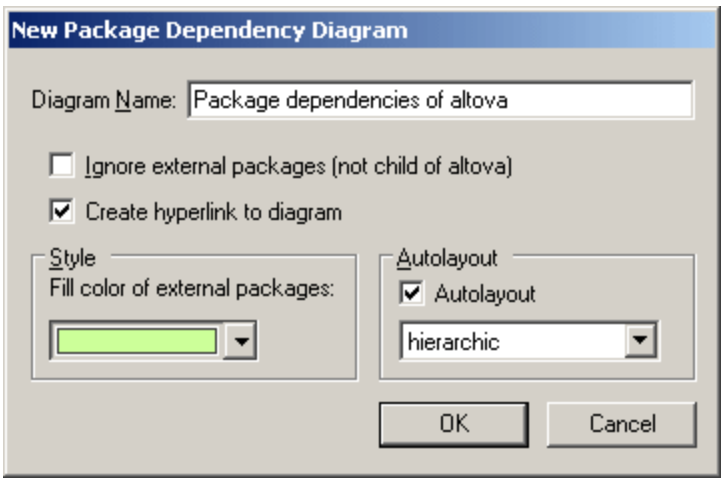


Automatic Package Dependency diagram generation
UModel has the capability to generate a package dependency diagram for any package in the Model Tree.

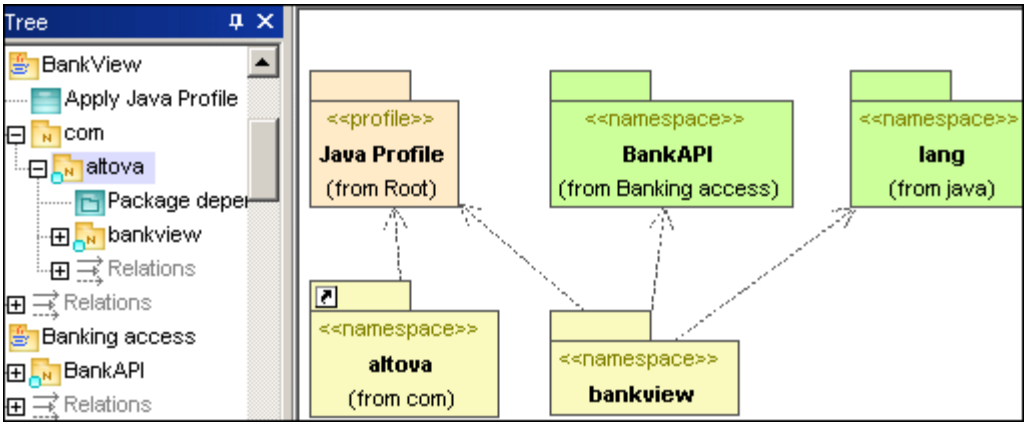
Dependency links between packages are created if there are any references between the modeling elements of those packages. E.g. Dependencies between classes, derived classes, or if attributes have types that are defined in a different package.

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.



A new diagram is generated and displays the package dependencies of the altova package.

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.
   Note that holding down CTRL and clicking in the diagram tab, allows you to insert multiple elements of the type you selected.

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 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.

9.3 Additional Diagrams

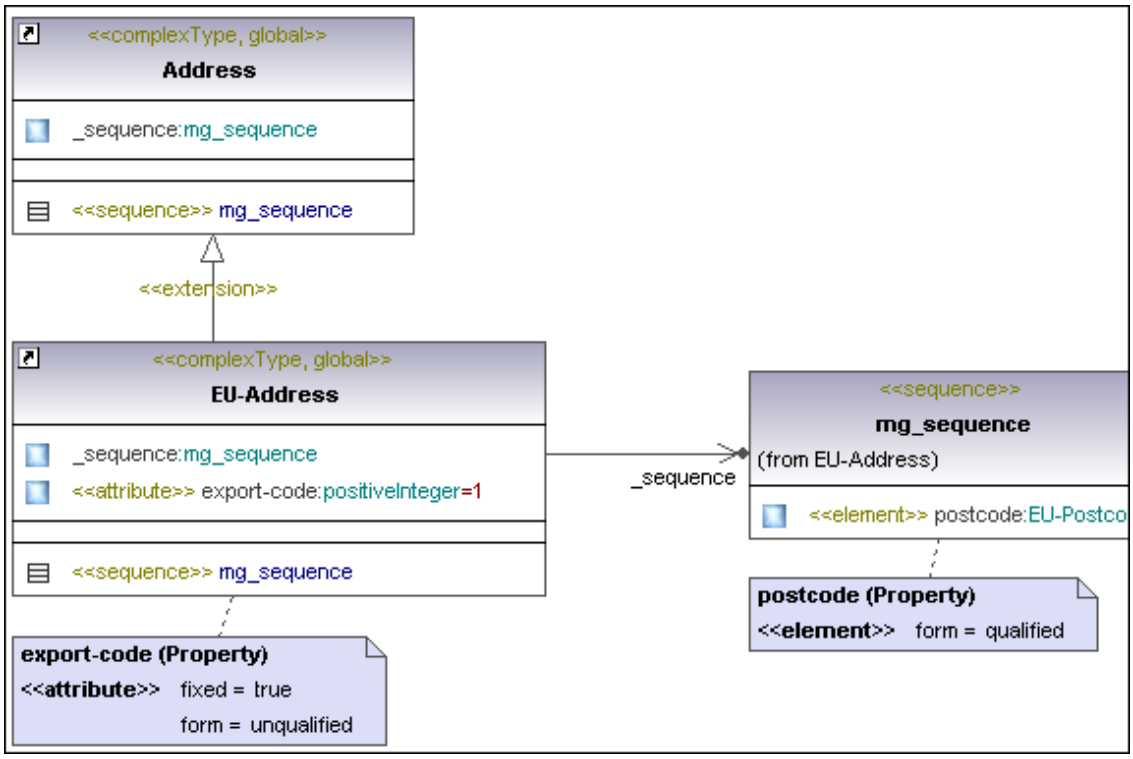
UModel now supports the import and generation of W3C XML Schemas as well as their forward and reverse-engineering in the code-engineering process.


XML Schema

9.3.1 XML Schema Diagrams

XML Schema diagrams display schema components in UML notation. Global elements i.e. elements, simpleTypes, complexTypes are shown as classes, or datatypes, with attributes in the attributes compartment. There are no operations in the Operation compartment. The Tagged Value note modeling element is used to display the schema details.

To see how the UML elements and XML schema elements/attributes are mapped, navigate to XML Schema to/from UModel elements.



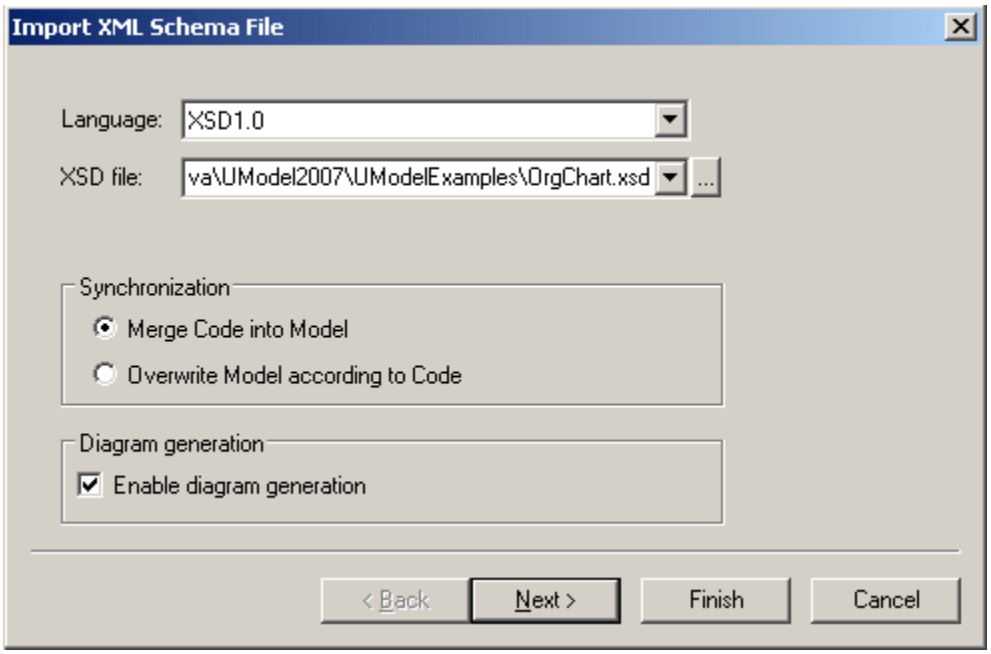
Please note:

Invalid XML Schemas cannot be imported into UModel. XML Schemas are not validated when importing, or creating them in UModel. XML Schemas are also not taken into account during the project syntax check. A well-formed check is however performed when importing an XML schema.

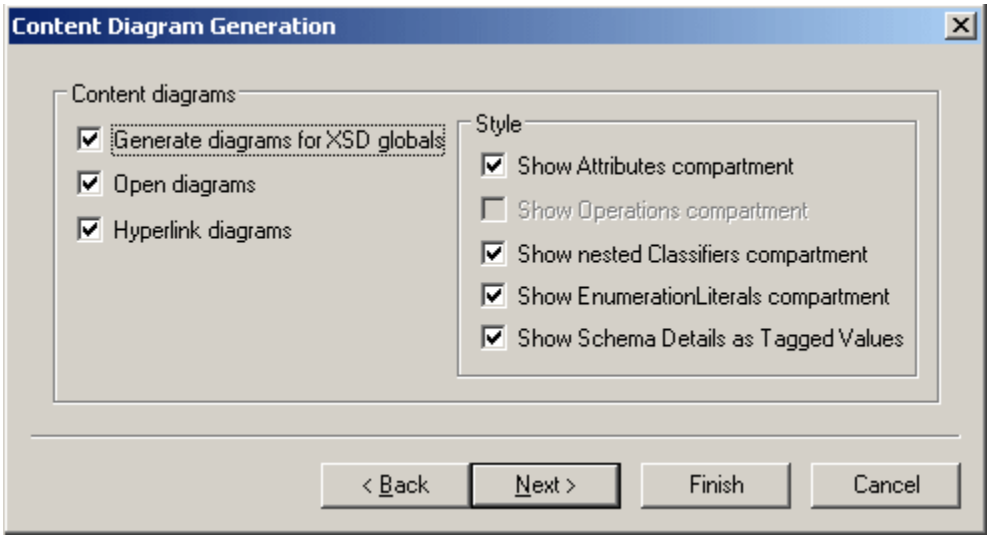
Importing an XML Schema

To import an XML Schema:

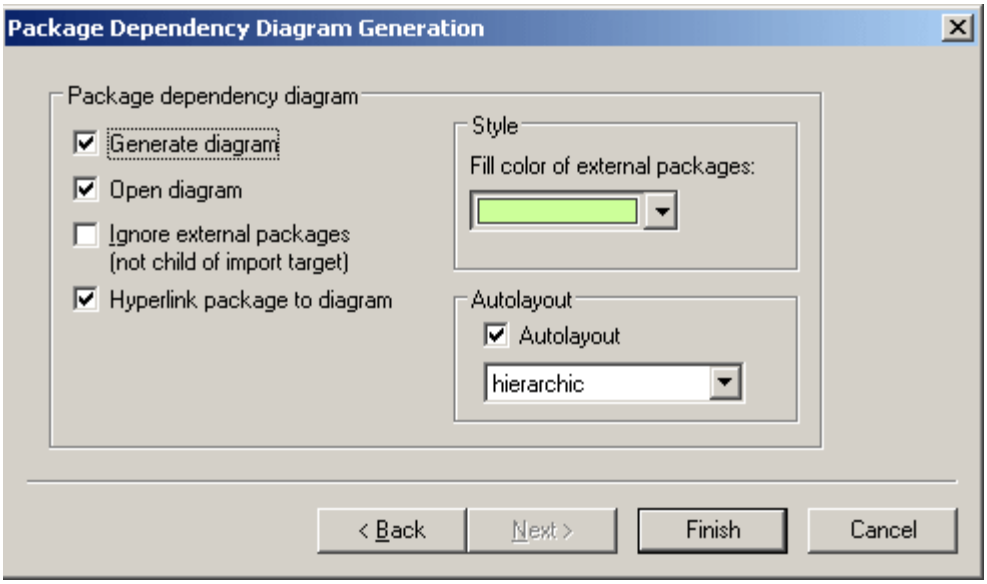
1. Select the menu option **Project | Import XML Schema file**.



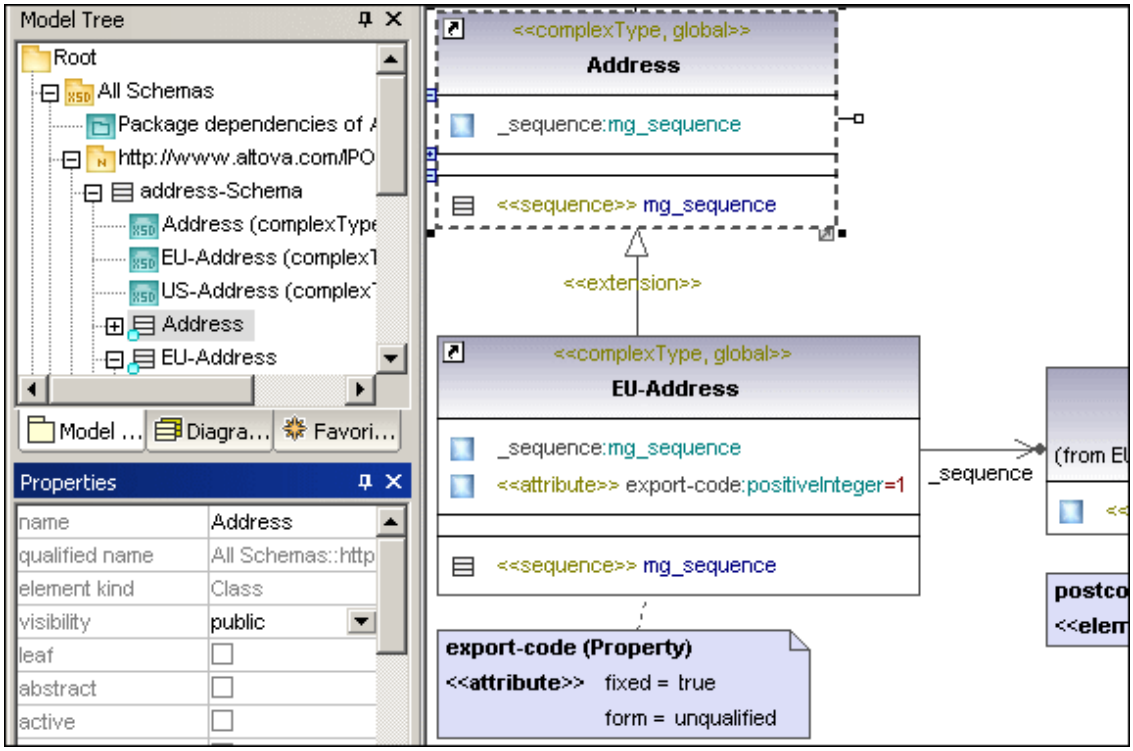
2. Make sure that the **Enable diagram generation** check box is active and click Next, to continue.



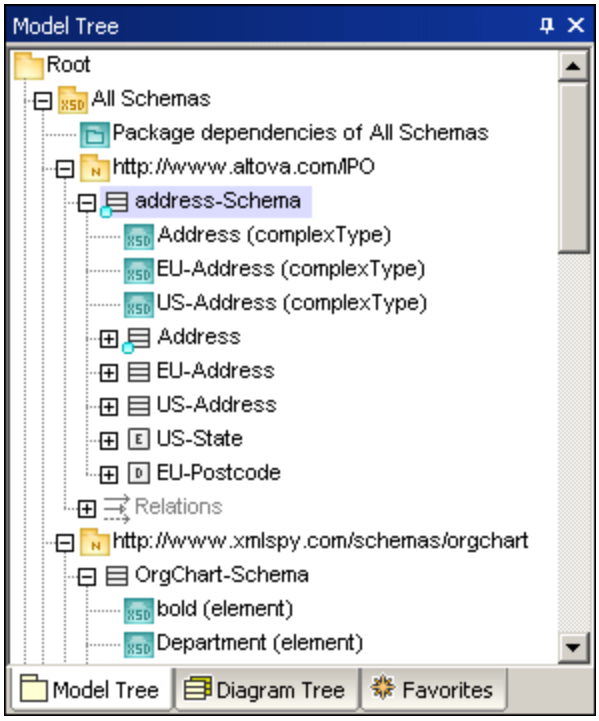
3. Define the Content diagram options in the group of that name. The first option creates a separate diagram for each schema global element.
4. Select the compartments that are to appear in the class diagrams in the Style group. The "Show schema details as tagged values" option displays the schema details in the Tagged Value note modeling element.
5. Click Next to define the Package dependency diagram.



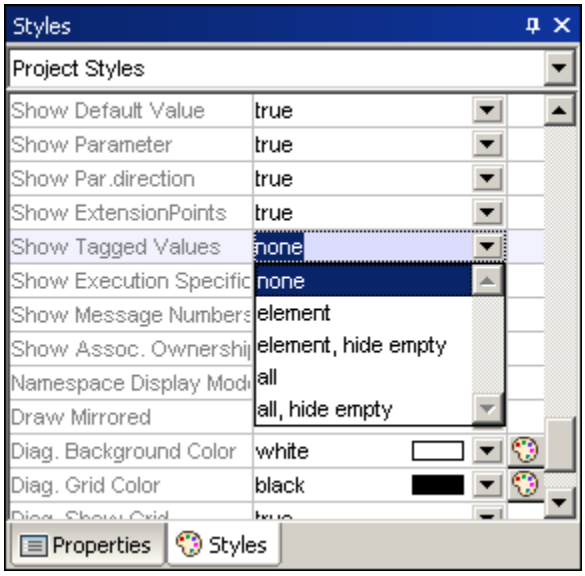
6. Click Finish to start the XML Schema import.
   The schema(s) are imported into UModel and all diagrams are available as tabs. The screenshot below shows the content of the EU-Address (complexType) diagram.



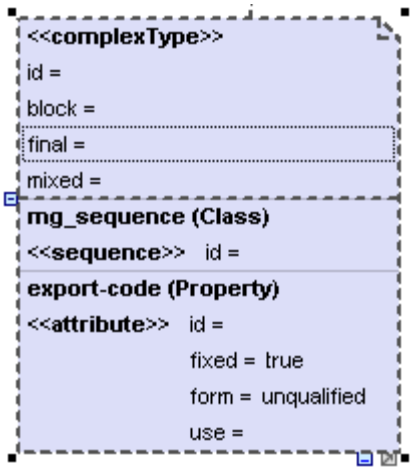
Please note:
A new package called All Schemas was created and set as the XSD Namespace Root. All XSD globals generate an XML Schema diagram, with the diagrams under the respective namespace packages.



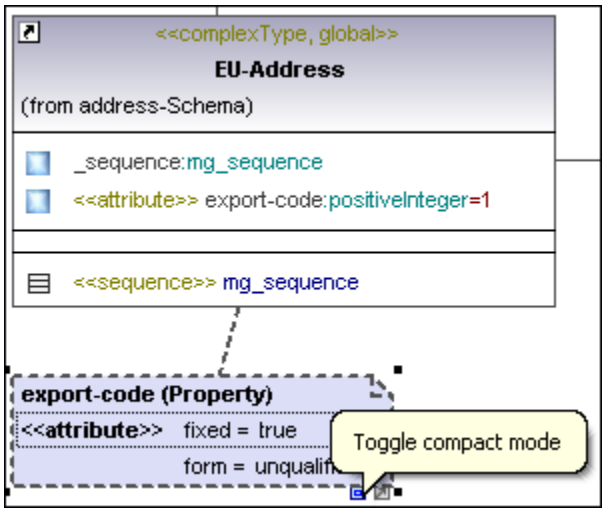
Schema details display - tagged values
Schema details displayed as tagged values in the Tagged Value note element, can be configured using the Show Tagged Values in the Styles tab, or by clicking the "Toggle compact mode" icon at the bottom right of the Tagged Value note. This switches between the two states "all" and "all, hide empty", both of which are shown below.



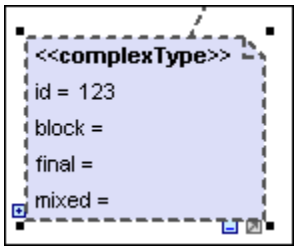
Show tagged values: **all**
Displays the tagged values of the class as well as those of the owned attributes, operations etc.



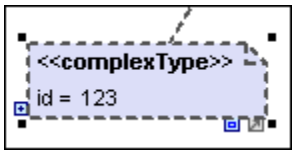
Show tagged values: **all, hide empty**
Displays only those tagged values where a value exists e.g. fixed=true.



Show tagged values: **element**
Displays the tagged values of the class but **not** those of the owned attributes, operations etc.



Show tagged values: **element, hide empty**
Displays only those tagged element values of a class, without the owned attributes, where a value exists e.g. id=123



XML Schema annotation:
When importing XML schemas, please note that only the first annotation of a complex- or simpleType is displayed in the Documentation window.

Inserting XML Schema elements

Using the toolbar icons:

1. Click the specific XML Schema diagram icon in the toolbar.
2. Click in the XML Schema diagram to insert the element.
   Note that holding down CTRL and clicking in the diagram tab, allows you to insert multiple elements of the type you selected.

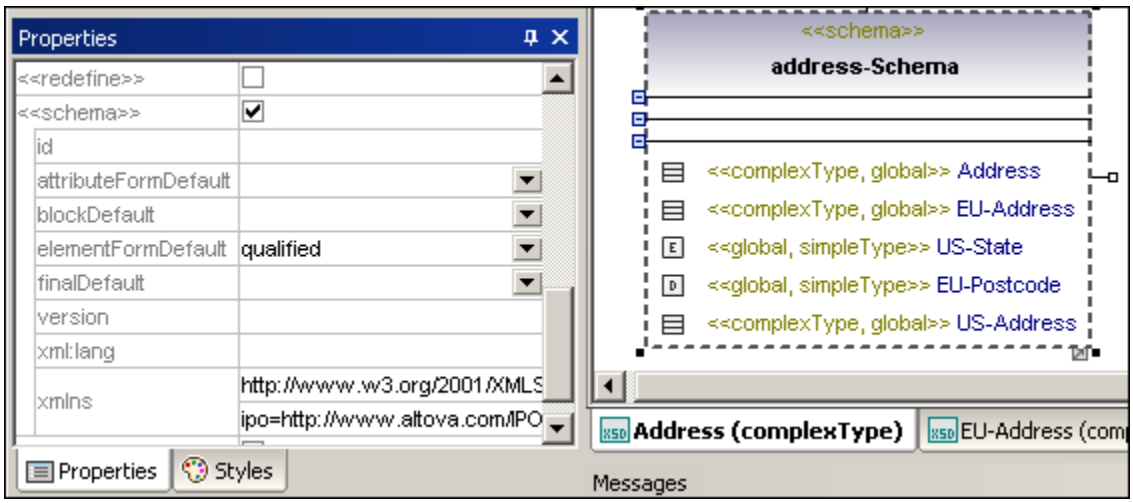
Dragging existing elements into the XML Schema diagram:
Elements occurring in other diagrams can be inserted into an existing XML Schema diagram.

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 XML Schema diagram.

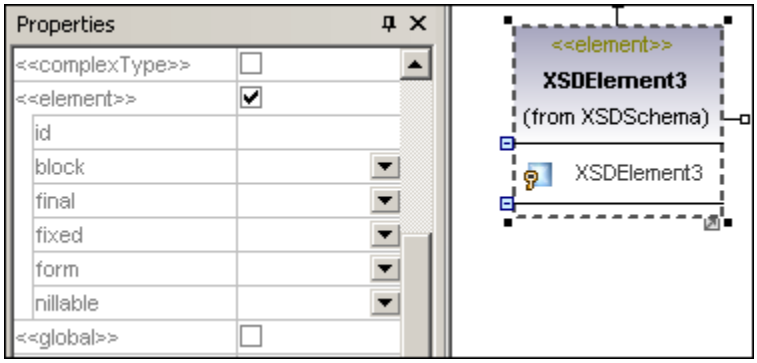
   Note: you can also use the Copy and "Paste in diagram only" commands to insert elements.


XSD Target Namespace
Inserts/defines the target namespace for the schema. The XSD Target Namespace must belong to an XSD Namespace Root package.

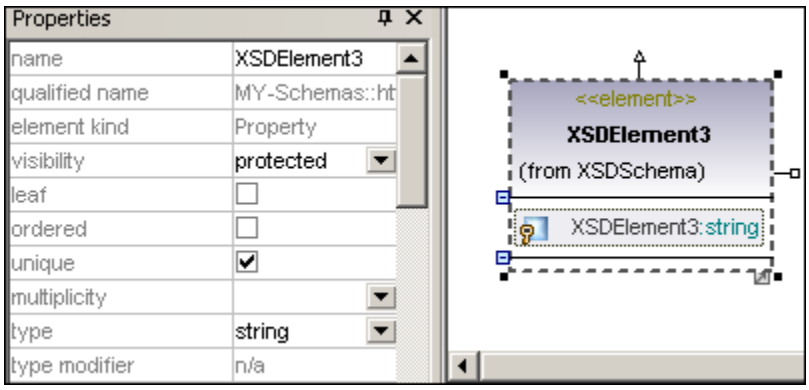

XSD Schema
Inserts/defines an XML schema. The XSD schema must belong to an XSD Target Namespace package.




Element (global)
Inserts a global element into the diagram. Note that a property is also automatically generated in the attributes compartment.

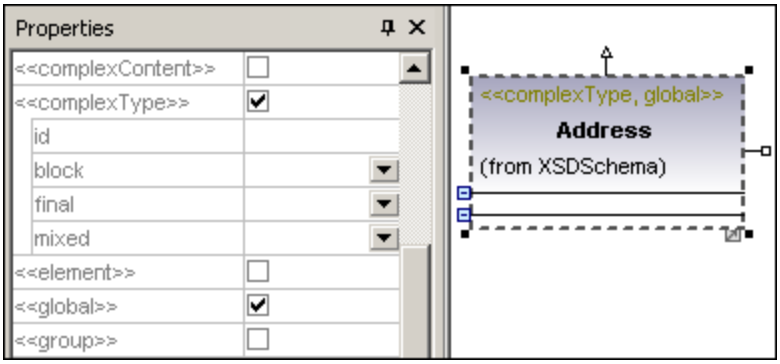


To define the property datatype:
1. Double click the property and place the cursor at the end of the line.
2. Enter a colon character ":", and select the datatype from the popup dialog box, e.g string.

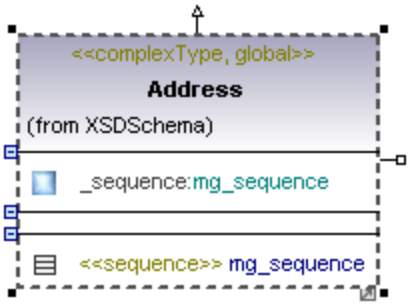


Creating a "content model" consisting of a complexType with mandatory elements:
This will entail inserting a complexType element, a sequence element/compositor, and three elements.

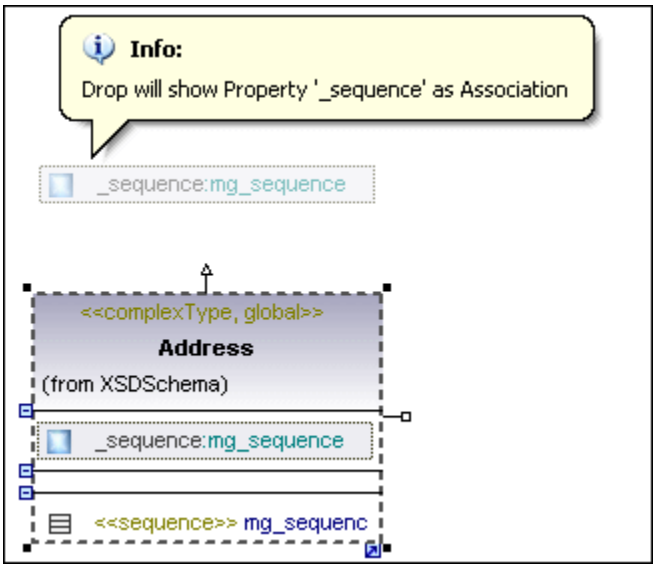
1. Click the XSD ComplexType icon  , then click in the diagram to insert it.
2. Double click the name and change it to Address.



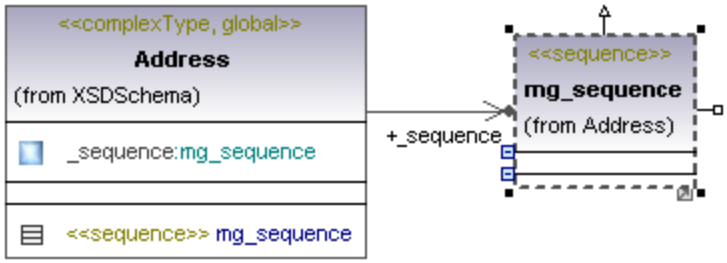
3. Right click Address and select **New | XSD Sequence**.



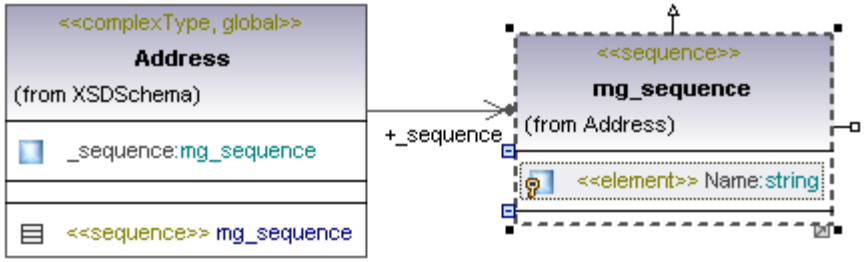
4. Click the **_sequence:mg_sequence** attribute in the attribute compartment, and drag it out into the diagram.



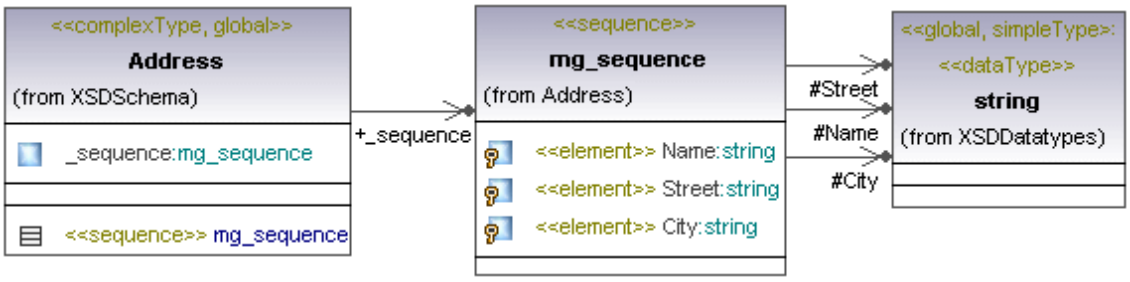
This creates a sequence class/compositor at the drop position.



5. Right click the sequence class and select **New | XSD Element (local)**.
   This adds a new property element.
6. Double click the property, enter the element name, e.g. Name, add a colon ":" and enter "string" as the datatype.



7. Do the same for the two more elements naming them Street and City for example.
8. Click the Name property and drag it into the diagram.

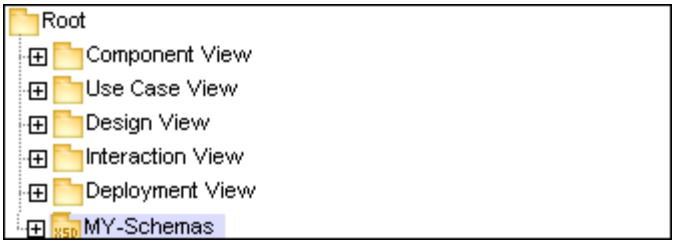


Creating and generating an XML Schema

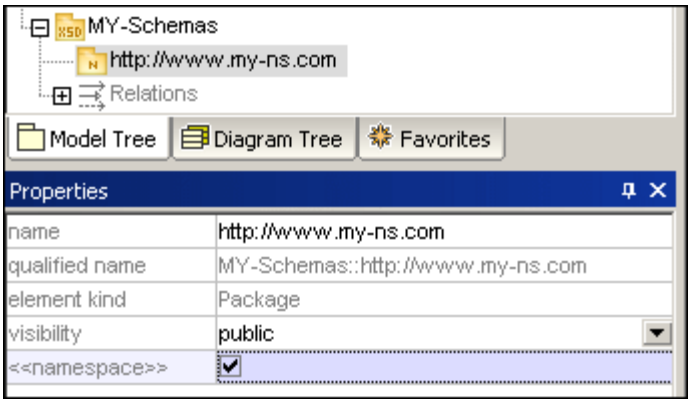
You would generally import a schema, edit it in UModel, and output the changes. It is however possible to generate a schema from scratch. This will only be described in broad detail however.

Creating a new schema in UModel:

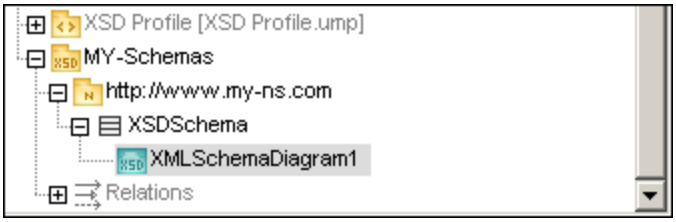
1. Create a new package in the Model Tree e.g. MY-Schemas.



2. Right click the new package and select the menu option **Code Engineering | Set as XSD namespace root**.
   You are asked if you want to assign the XSD profile if this is the first XSD Namespace root in the project.
3. Click OK to assign the profile.
4. Right click the new package and select **New Element | Package**.
5. Double click in the package name field and change it to the namespace you want to use, e.g. http://www.my-ns.com.
6. Click the <<namespace>> check box in the Properties tab, to define this as the target namespace.



7. Right click the namespace package and select **New diagram | XML Schema diagram**.
   You prompted if you want to add the Schema diagram to a new XSD Schema.
8. Click Yes to add the new diagram.

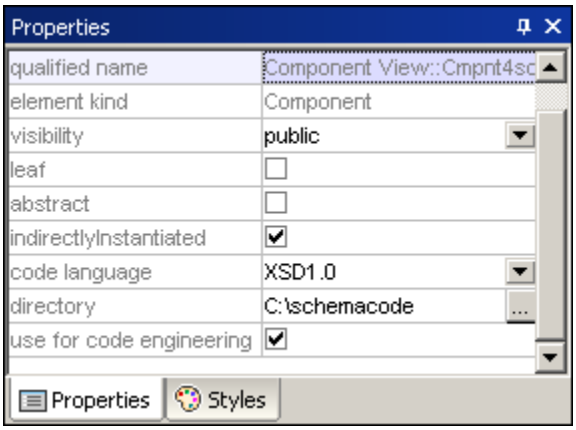


You can now create your schema using the icons in the XML Schema icon bar.

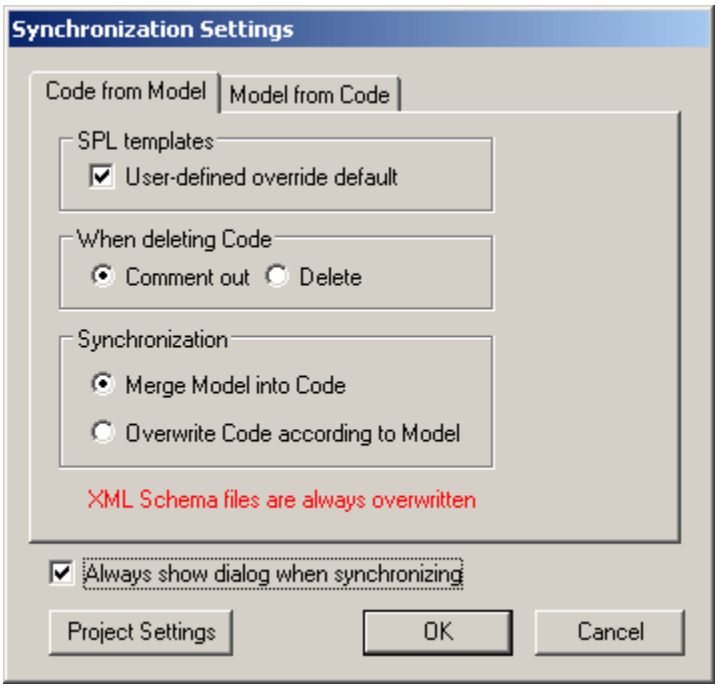
Generating the XML schema:

1. Drag the XSDSchema onto a component to create a Component Realization.

2. Make sure that you set the code language, of the component, to XSD1.0, and define a directory for the generated schema to be placed in.



3. Select the menu option **Project | Merge Program Code from UModel project,** and click OK to generate the schema.



Chapter 10

XMI - XML Metadata Interchange

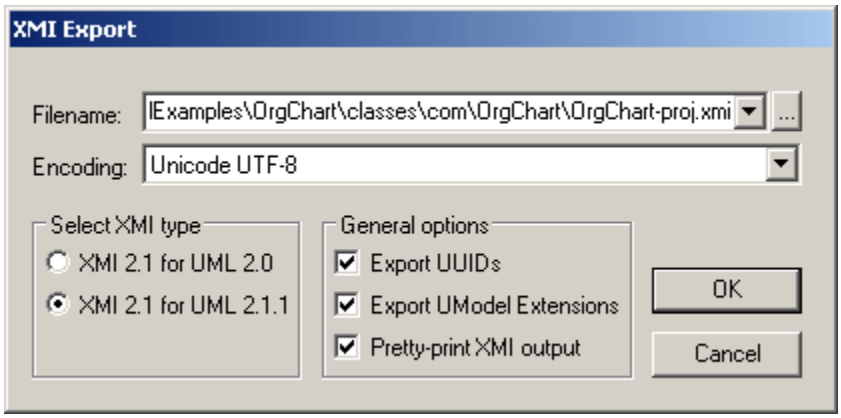
10 XMI - XML Metadata Interchange

UModel supports the export and import of XMI 2.1 for UML 2.0 / 2.1 and 2.1.1.

Select the menu item **File | Export to XMI File** to generate an XMI file from the UModel project, and File | Import from XMI File, to import a previously generated XMI file.

The XMI Export dialog box allows you to select the specific XMI format you want to output, XMI for UML 2.0/2.1.1. During the export process included files, even those defined as "include by reference" are also exported.

Please note:
If you intend to **reimport** generated XMI code into UModel, please make sure that you activate the "Export UModel Extensions" check box.



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" checkbox.
- 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.

Please note:
The XMI import process automatically supports both types of IDs.

XMI extensions
XMI defines an "extension mechanism" which allows each application to export its tool-specific extensions to the UML specification. If you select this option, other UML tools will 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.

Pretty-print XMI output
This option outputs the XMI file with XML appropriate tag indentation and carriage returns/line feeds.

Chapter 11

UModel Diagram icons

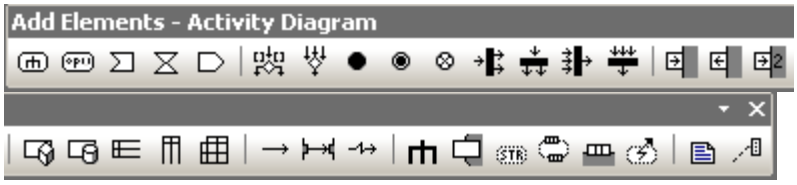
11 UModel Diagram icons

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.

11.1 Activity Diagram



Add
Action (CallBehaviorActiion)
Action (CallOperationAction)
AcceptEventAction
AcceptEventAction (TimeEvent)
SendSignalAction

DecisionNode (Branch)
MergeNode
InitialNode
ActivityFinalNode
FlowFinalNode
ForkNode (vertical)
ForkNode (horizontal)
JoinNode
JoinNode (horizontal)

InputPin
OutputPin
ValuePin

CentralBufferNode
DataStoreNode
ActivityPartition (horizontal)
ActivityPartition (vertical)
ActivityPartition 2-Dimensional

ControlFlow
ObjectFlow
ExceptionHandler

Activity
ActivityParameterNode
StructuredActivityNode
ExpansionRegion
ExpansionNode
InterruptibleActivityRegion

Note
Note Link

11.2 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

11.3 Communication diagram

Add
Lifeline
Message (Call)
Message (Reply)
Message (Creation)
Message (Destruction)

Note
Note Link

11.4 Composite Structure Diagram

Add
Collaboration
CollaborationUse
Part (Property)
Class
Interface
Port

Relationship
Connector
Dependency (Role Binding)
InterfaceRealization
Usage

Note
Note Link

11.5 Component Diagram

Add:
Package
Interface
Class
Component
Artifact

Relationship:
Realization
InterfaceRealization
Usage
Dependency

Note
Note Link

11.6 Deployment Diagram

Add:
Package
Component
Artifact
Node
Device
ExecutionEnvironment

Relationship:
Manifestation
Deployment
Association
Generalization
Dependency

Note
Note Link

11.7 Interaction Overview diagram

Add
CallBehaviorAction (Interaction)
CallBehaviorAction (InteractionUse)
DecisionNode
MergeNode
InitialNode
ActivityFinalNode
ForkNode
ForkNode (Horizontal)
JoinNode
JoinNode (Horizontal)
DurationConstraint

Relationship
ControlFlow

Note
Note Link

11.8 Object Diagram

Relationship:
Association
AssociationClass
Dependency
Usage
InterfaceRealization
Generalization

Add:
Package
Class
Interface
Enumeration
Datatype
PrimitiveType
InstanceSpecification

Note
Note Link

11.9 Package diagram

Add
Package
Profile

Relationship
Dependency
PackageImport
PackageMerge
ProfileApplication

Note
Note Link

11.10 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

11.11 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

11.12 Timing Diagram

Add
Lifeline (State/Condition)
Lifeline (General value)
TickMark
Event/Stimulus
DurationConstraint
TimeConstraint

Message (Call)
Message (Reply)
Asynchronous Message (Call)

Note
Note Link

11.13 Use Case diagram

Add:
Package
Actor
UseCase

Relationship:
Association
Generalization
Include
Extend

Note
Note Link

11.14 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

Chapter 12

UModel Reference

12 UModel Reference

The following section lists all the menus and menu options in UModel, and supplies a short description of each.

12.1 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.

Reload
Allows you to reload the current project and save, or discard, the changes made since you opened the project file.

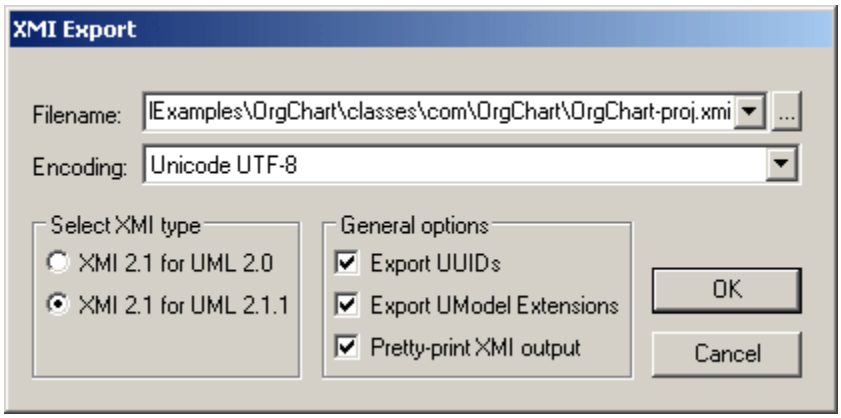
Save
Saves the currently active modeling project using the currently active file name.

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 Diagram as Image
Opens the "Save as..." dialog box and allows you to save the currently active diagram as a .PNG, or .EMF (enhanced metafile) file.

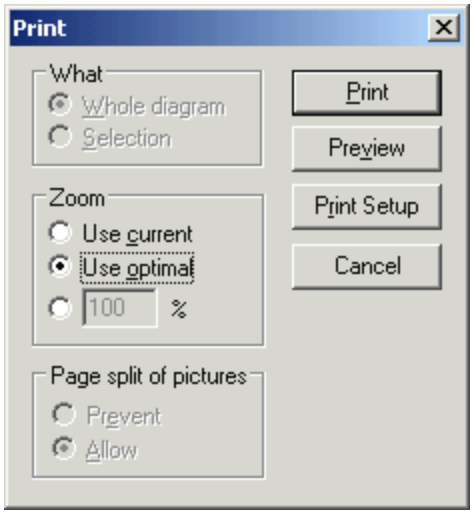
Import from XMI file
Imports a previously exported XMI file. If the file was produced with UModel, then all extensions etc. will be retained.

Export to XMI file
Export the model as an XMI file. You can select the UML version, as well as the specific IDs that you want to export please see XMI - XML Metadata Interchange for more information.



Send by Mail
Opens your default mail application and inserts the current UModel project as an attachment.

Print
Opens the Print dialog box, from where you can print out your modeling project as hardcopy.



"Use current", retains the currently defined zoom factor of the modeling project. Selecting this option enables the "Page split of pictures" group.

The Prevent option prevents modeling elements from being split over a page, and keeps them as one unit.

"Use optimal" scales the modeling project to fit the page size. You can also specify the zoom factor numerically.

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.

12.2 Edit

Undo 
UModel has an unlimited number of "Undo" steps that you can use to retrace you 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//Delete
The standard windows Edit commands, allow you to cut, copy, etc., modeling elements, please see "Cut, copy and paste in UModel Diagrams" for more information.

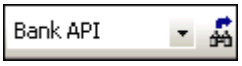
Paste
using the keyboard shortcut CTRL+V, or "Paste" from the context menu, as well as Paste from the Edit menu, always adds a **new** modeling element to the diagram and to the Model Tree, please see "Cut, copy and paste in UModel Diagrams".

Paste in Diagram only
using the context menu, i.e. right clicking on the diagram background, only adds a "link/view" of the existing element, to the current diagram and not to the Model Tree, please see "Cut, copy and paste in UModel Diagrams".

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
There are several options you can use to search for modeling elements:

- Use the text box in the Main **title bar** 

- Use the menu option Edit | Find
- Press the shortcut **CTRL+F** to open the find dialog box.

Allows you to search for specific text in:

- Any of the three Model Tree panes: Model Tree, Diagram Tree and Favorites tab.
- The Documentation tab of the Overview pane.

- Any currently active diagram.
- The Messages pane.

Find Next  **F3**
Searches for the next occurrence of the same search string in the currently active tab or diagram.

Find Previous SHIFT+F3
Searches for the previous occurrence of the same search string in the currently active tab or diagram.

Copy as bitmap
Copies the currently active diagram into the clipboard from where you can paste it into the application of your choice.

Please note:
Diagrams are copied into the system clipboard, you have to insert them into another application to see, or get access to them.

Copy selection as bitmap
Copies the currently **selected diagram elements** into the clipboard from where you can paste them into the application of your choice.

12.3 Project

Check Project Syntax...
Checks the UModel project syntax. The project file is checked on multiple levels detailed in the tables below:

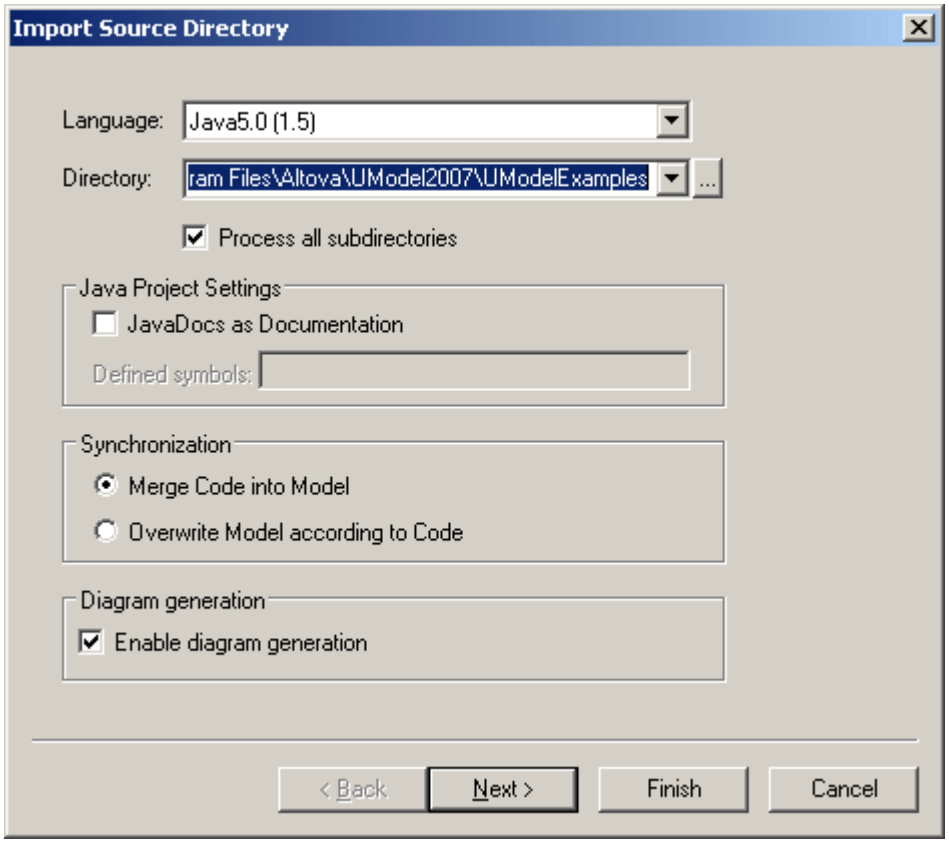
Level	Checks if...	Message...
Project level	at least one Java Namespace Root exists	Error
Components	Project file / Directory is set	Error
	If Realization exists	Error
	"Use for code engineering" check box unchecked: no check is performed and syntax check is disabled.	None
Class	Code file name is set.	**Error** if the local option "Generate missing code file names" is not set. **Warning** if the option is set.
	If class is nested then no check performed.	
	If contained in a code language namespace	Error
	Type for operation parameter is set	Error
	Type for properties is set	Error
	Operation return type is set	Error
	Duplicate operations (names + parameter types)	Error
	If classes are involved in Realization, only if the class is not nested.	Warning
Interface	Code file name is set.	**Error** if the local option "Generate missing code file names" is not set. **Warning** if the option is set.
	Contained in a code language namespace	Error
	Type for properties are set	Error
	Type for operation param. are set	Error
	Operation return type is set	Error
	Duplicate operations (names + parameter types)	Error
	If interfaces are involved in a ComponentRealization	Warning
Enumeration	Belongs to Java Namespace Root: gives a warning to say that no code will be generated.	Warning
	Does not belong to Java Namespace Root: no check is performed and syntax check is disabled for the enumeration. No check is performed on contained package	None

Syntax check for all UML elements involved in code generation

class	Checks name is a valid Java name (no forbidden characters, name is not a keyword)	Error
class property	Checks name is a valid Java name (no forbidden characters, name is not a keyword)	Error
class operation	Checks name is a valid Java name (no forbidden characters, name is not a keyword) Checks for existence of return parameter	Error
class operation parameter	Checks name is a valid Java name (no forbidden characters, name is not a keyword) Checks type has a valid Java type name	Error
interface	Checks name is a valid Java name (no forbidden characters, name is not a keyword)	Error
interface operation	Checks name is a valid Java name (no forbidden characters, name is not a keyword)	Error
interface operation parameter	Checks name is a valid Java name (no forbidden characters, name is not a keyword)	Error
interface properties	Checks name is a valid Java name (no forbidden characters, name is not a keyword)	Error
package with stereotype namespace	Checks name is a valid Java name (no forbidden characters, name is not a keyword)	Error
package without stereotype namespace	no element to check	None
class	multiple inheritance	Error

Please note:
Constraints on model elements are not checked, as they are not part of the Java code generation process. Please see "constraining model elements" for more information.

Import Source Directory...
Opens the Import Source Directory wizard shown below. Please see "Round-trip engineering (code - model - code)" for a specific example.

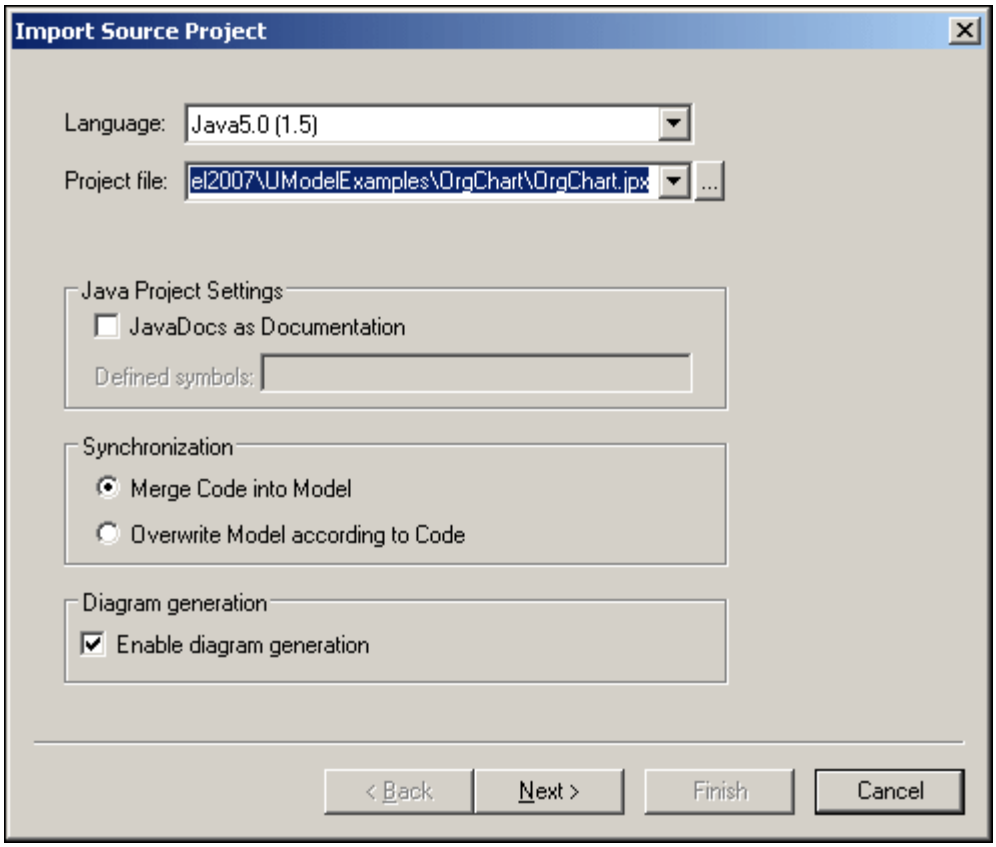


Import Source Project...

Opens the Import Source Project wizard shown below. Clicking the browse button allows you to select the project file and the specific project type. Please see "Importing source code into projects" for a specific example.

Java projects:

- JBuilder **.jpx,** Eclipse **.project** project files, as well as NetBeans (project.**xml**) are currently supported.



C# projects:

- MS Visual studio.Net projects, **csproj**, **csdprj**..., as well as
- Borland **.bdsproj** project files

Import Binary Types
Opens the Import Binary Types dialog box allowing you to import Java and C# binary files. Please see "Importing C# and Java binaries" for more information.

Import XML Schema File
Opens the Import XML Schema File dialog box allowing you to import schema files. Please see "XML Schema Diagrams" for more information.

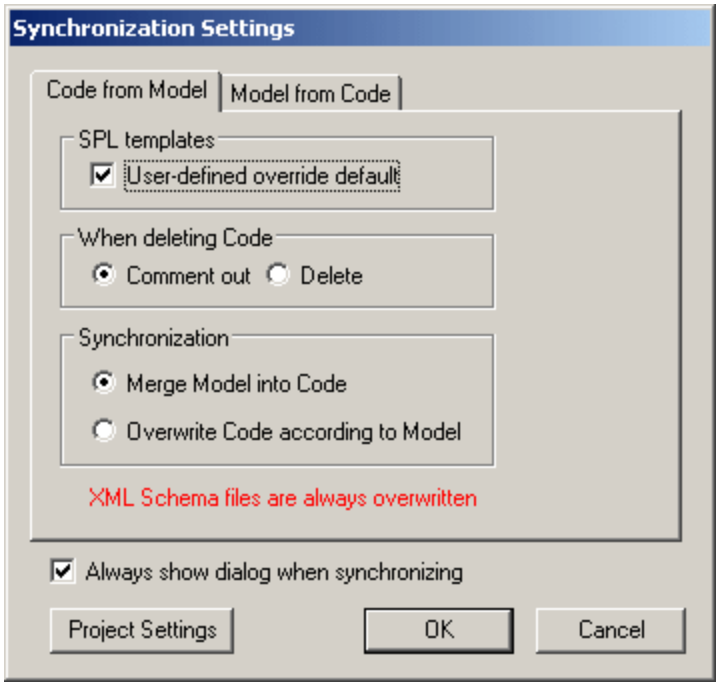
Merge Program Code from UModel Project
Opens the Synchronization Settings dialog box with the "**Code from Model**" tab active. Clicking the Project Settings button allows you to select the specific programming language settings.

Merging or overwriting code
Assuming that code has been generated once from a model, and changes have since been made to both model and code e.g.:

- Model elements have been added in UModel e.g. a new class X
- A new class has been added to the external code e.g. class Y

Merging (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.

Overwriting (code according to model) means that:
- the newly added class Y in the external code is **deleted**
- the newly added class X, from UModel, is added to the code.



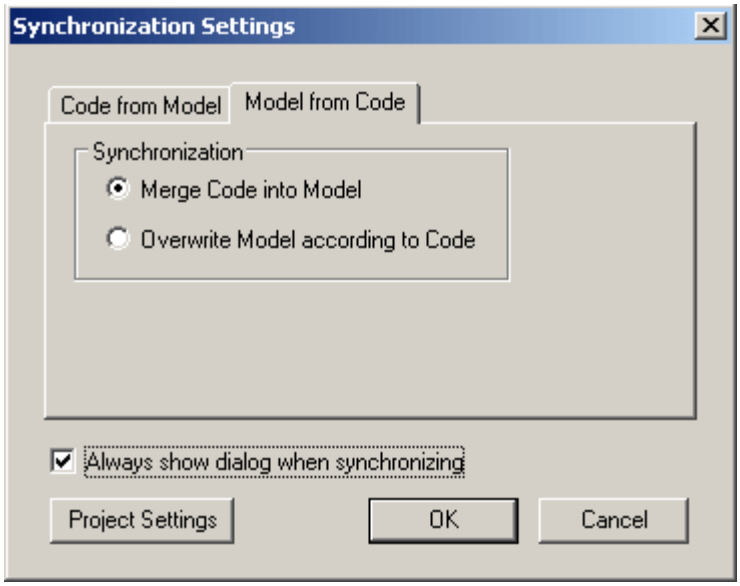
Merge UModel Project from Program Code
Opens the Synchronization Settings dialog box with the "**Model from Code**" tab active. Clicking the Project Settings button allows you to select the specific programming language settings.

Merging or overwriting code
Assuming that code has been generated once from a model, and changes have since been made to both model and code e.g.:

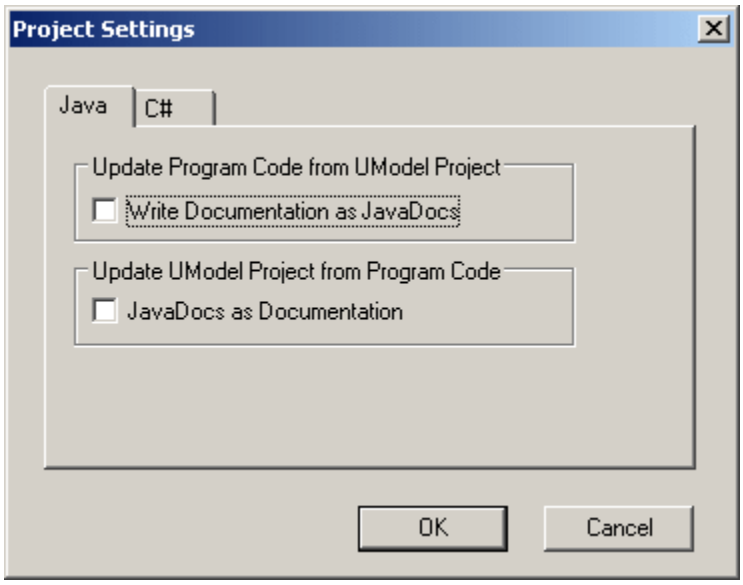
- Model elements have been added in UModel e.g. a new class X
- A new class has been added to the external code e.g. class Y

Merging (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

Overwriting (Model according to code) means that:
the newly added class X in UModel is **deleted**
the newly added class Y, from the external code, is added to the model



Project settings
Allows you to define the specific languages settings for your project.



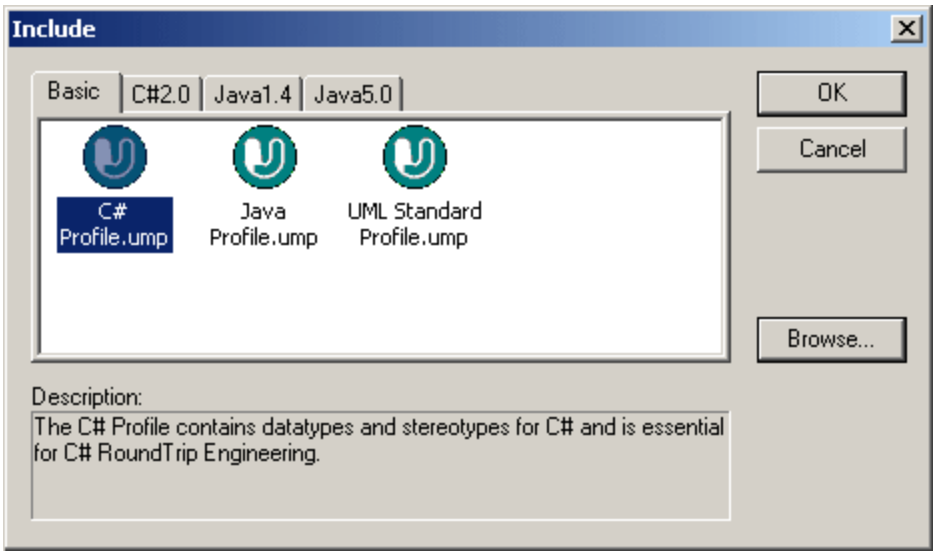
Synchronization Settings...
Opens the Synchronization Settings dialog box as shown in the screenshots above.

Include Subproject
UModel is supplied with several files that can be included in a UModel project. Clicking the Java tab allows you to include Java lang classes, interfaces and packages in your project, by selecting one of the supplied files.

1. Select **Project | Include** to open the "Include" dialog box.
2. Click the UModel project file you want to include and press OK.

UModel projects can be included within other UModel projects. To include projects place the respective *.ump files in:

- ...\UModel2007**UModelInclude** to appear in the Basic tab, or
- ...\UModel2007**UModelInclude\Java** to appear in the Java tab.



Please note:
An include file, which contains all types of the Microsoft .NET Framework 2.0, is available in the C# 2.0 tab.

To create a user-defined tab/folder:

1. Navigate to the ...\UModel2007**UModelInclude** and create/add your folder below ...\UModelInclude, i.e. ...**UModelInclude\myfolder.**

To create descriptive text for each UModel project file:

1. Create a text file using the same name as the *.ump file and place in the same folder. Eg. the **MyModel.ump** file requires a descriptive file called **MyModel.txt.**

To remove an included project:

1. Click the included package in the Model Tree view and press the Del. key.
2. You are prompted if you want to continue the deletion process.
3. Click OK to delete the included file from the project.

Please note:

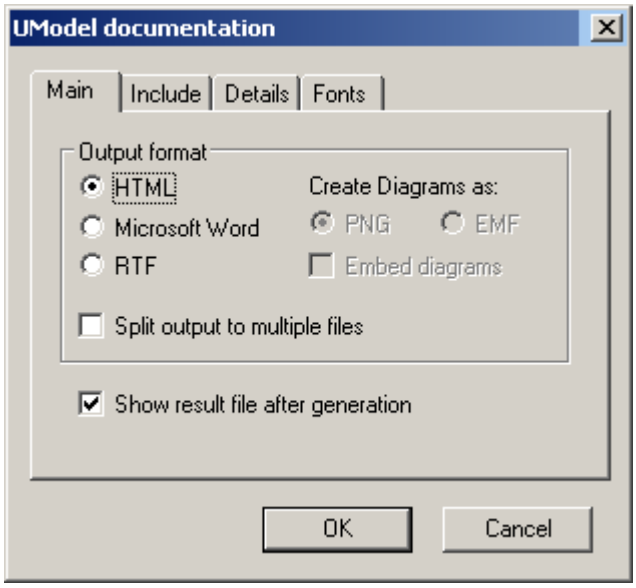
- To delete or remove a project from the "Include" dialog box, delete or remove the (MyModel).ump file from the respective folder.

Open Subproject as project
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.

Please 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
Allows you to generate documenation for the currently open project in HTML, Microsoft Word, and RTF formats. please see Generating UML documentation for more information.



List Elements not used in any Diagram
Creates a list of all elements not used in any diagram in the project.

List shared Packages
Lists all shared packages of the current project.

List included Packages
Lists all include packages in the current project. Java Profile (Java Profile.ump) and Java Lang (Java Lang.ump) are automatically supplied in the Bankview example supplied with UModel.

12.4 Layout

The commands of the Layout menu allow you to line up and align the elements of your modeling diagrams.

When using the marquee (drag on the diagram background) to mark several elements, the element with the dashed outline becomes the "active" element, i.e. the last marked element. All alignment commands use this element as the origin, or basis for the following alignment commands.

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 allows you to choose the type of presentation of the modeling elements in the UML diagram tab. "Force directed", displays the modeling elements from a centric viewpoint. "Hierarchic", displays elements according to their relationships, superclass - derived class etc.

Reposition text labels:
Repositions modeling element names (of the selected elements) to their default positions.

12.5 View

The commands available in this menu allow you to:

- **Switch**/activate tabs of the various panes
- Define the modeling element **sort criteria** of the Model Tree and Favorites tab
- Define the **grouping criteria** of the diagrams in the Diagram Tree tab
- Show or hide specific UML elements in the Favorites and Model Tree tab
- Define the **zoom** factor of the current diagram.

12.6 Tools

The tools menu allows you to:

- Customize your version: define your own toolbars, keyboard shortcuts, menus, and macros
- Define the global program settings

12.6.1 Customize...

The customize command lets you customize UModel to suit your personal needs.

Commands

The Commands tab allows you customize your menus or toolbars.

To add a command to a toolbar or menu:

1. Open this dialog box using **Tools | Customize.**
2. Select the command category in the Categories list box. The commands available appear in the Commands list box.
3. Click on a command in the commands list box and drag "it" to an to an existing menu or toolbar.
4. An I-beam appears when you place the cursor over a valid position to drop the command.
5. 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 tool bar or menu).
- 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.

Please note:

You can also edit the commands in the **context menus** (right click anywhere opens 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.

To delete a command or menu:

1. Open this dialog box using **Tools | Customize.**
2. Click on 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.

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:

1. 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 tab of the Customize dialog box.

To reset the Menu Bar

- Click the Menu Bar entry and
- Click the **Reset** button, to reset the menu commands to the state they were when installed.

To reset all toolbar and menu commands

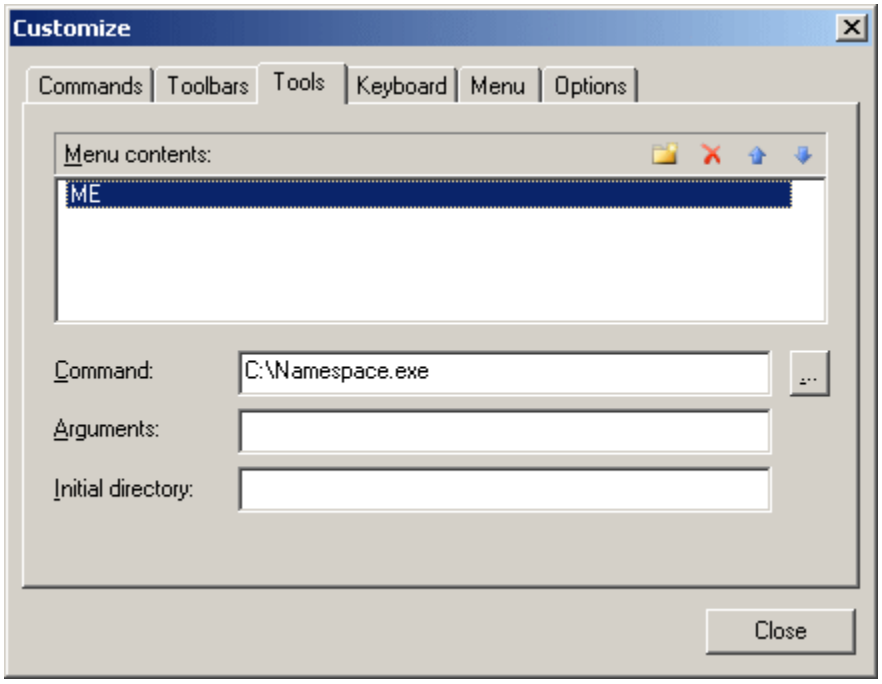
- Click the **Reset All** button, to reset all the toolbar commands to the state they were when the program was installed. A prompt appears stating that all toolbars and menus will be reset.
- Click Yes to confirm the reset.

Show text labels:
This option places explanatory text below toolbar icons when activated.

Tools

The Tools tab allows you to create your own menu entries in the Tools menu.

Click the folder icon to add a new menu entry and use the Command field to associate it to an application.



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 the commands category using the **Category** combo box.
2. Select the **command** you want to assign a new shortcut to, in the Commands list box
3. Click in 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 the **Assign** button 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 (or 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 the Close button to confirm all the changes made in the Customize dialog box.

Menu

The Menu tab allows you to customize the main menu bars as well as the (popup - right click) context menus.

You can customize both the Default and **UModel Project** menu bars.
The **Default** menu is the one visible when no XML documents of any type are open.
The **UModel Project** menu is the menu bar visible when a *.ump file has been opened.

To customize a menu:

1. Select the menu bar you want to customize from the "Show Menus for:" combo box
2. Click the **Commands** tab, and drag the commands to the menu bar of your choice.

To delete commands from a menu:

1. Click right on the command, or icon representing the command, and
2. Select the **Delete** option from the popup menu,

or,

1. Select **Tools | Customize** to open the Customize dialog box, and
2. Drag the command away from the menu, and drop it as soon as the check mark icon appears below the mouse pointer.

To reset either of the menu bars:

1. Select either the Default or UModel Project entry in the combo box, and
2. Click the **Reset** button just below the menu name.
   A prompt appears asking if you are sure you want to reset the menu bar.

To customize any of the Context menus (right click menus):

1. Select the context menu from the "Select context menus" combo box.
2. Click the **Commands** tab, and drag the specific commands to context menu that is now open.

To delete commands from a context menu:

1. Click right on the command, or icon representing the command, and
2. Select the **Delete** option from the popup menu

or,

1. Select **Tools | Customize** to open the Customize dialog box, and
2. Drag the command away from the context menu, and drop it as soon as the check mark icon appears below the mouse pointer.

To reset any of the context menus:

1. Select the context menu from the combo box, and
2. Click the **Reset** button just below the context menu name.
   A prompt appears asking if you are sure you want to reset the context menu.

To close an context menu window:

1. Click on the **Close icon** at the top right of the title bar, or
2. Click the Close button of the Customize dialog box.

Menu shadows

- Click the Menu shadows check box, if you want all your menus to have shadows.

Options

The Options tab allows you to set general environment settings.

Toolbar

When active, the **Show ToolTips on toolbars** check box displays a popup when the mouse pointer is placed over an icon in any of the icon bars. The popup contains a short description of the icon function, as well as the associated keyboard shortcut, if one has been assigned.

The **Show shortcut keys in ToolTips** check box, allows you to decide if you want to have the shortcut displayed in the tooltip.

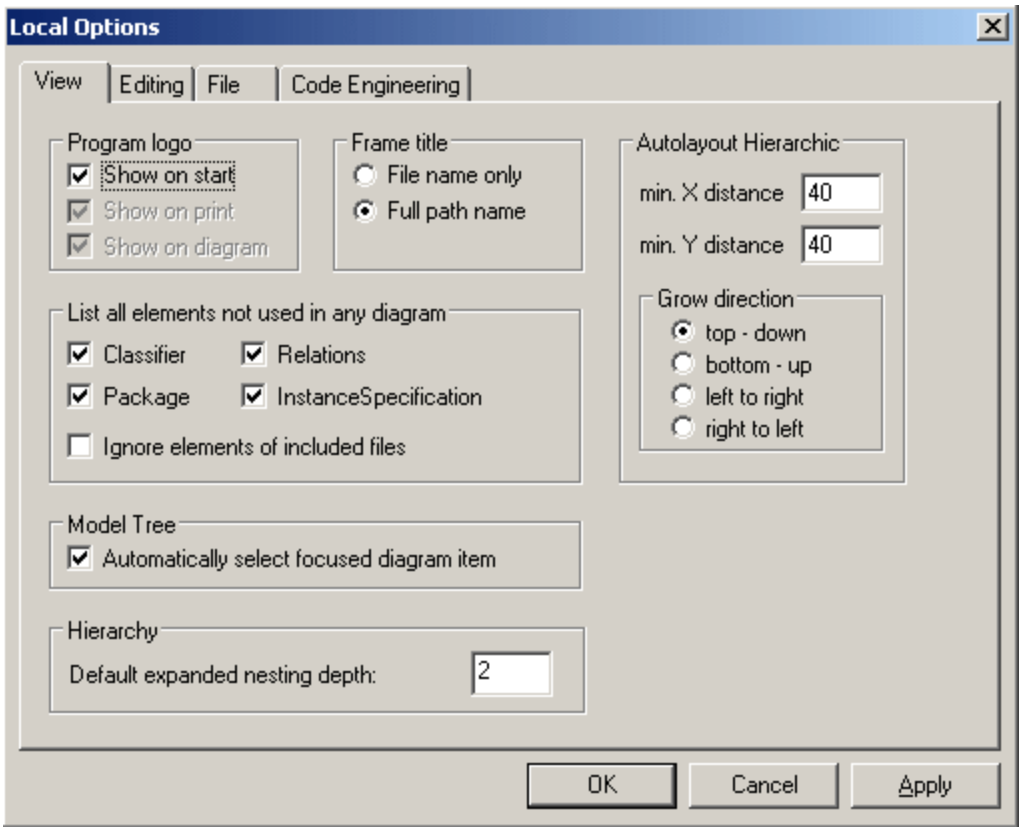
When active, the **Large icons** check box switches between the standard size icons, and larger versions of the icons.

12.6.2 Options

Select the menu item **Tools | Options** to define your project options.

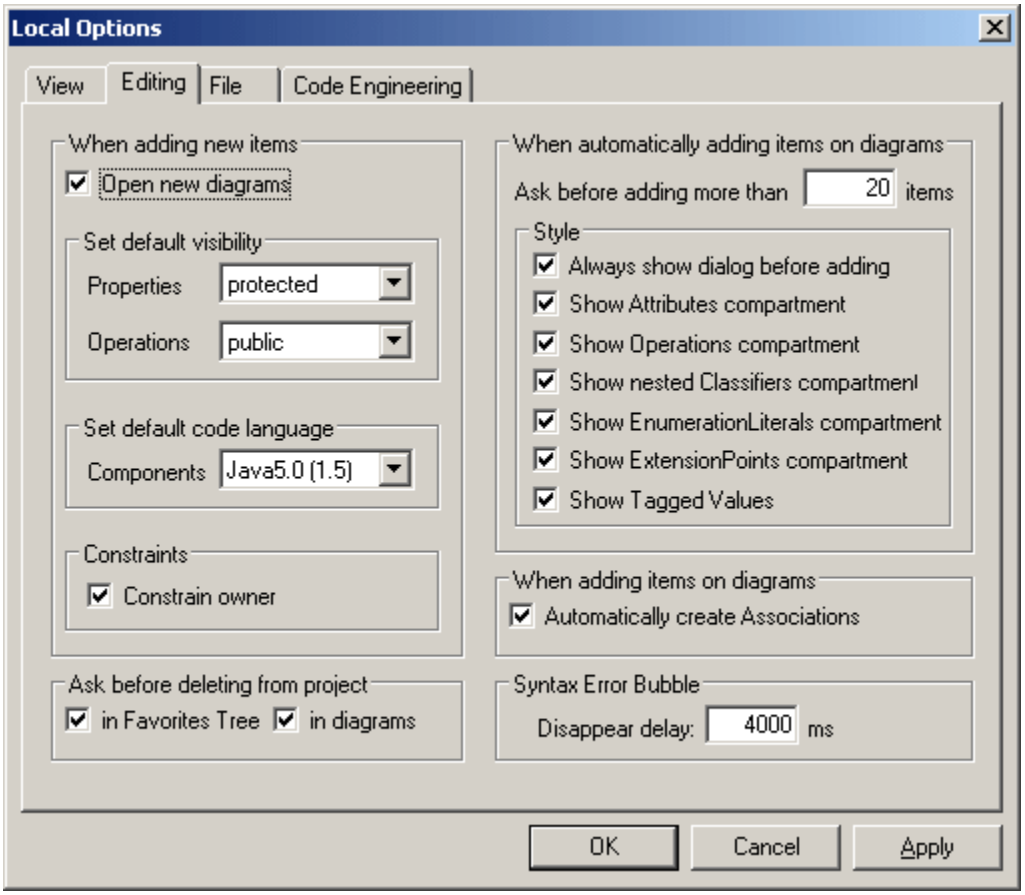
The **View** tab allows you to define:

- 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.
- Autolayout settings.
- 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.



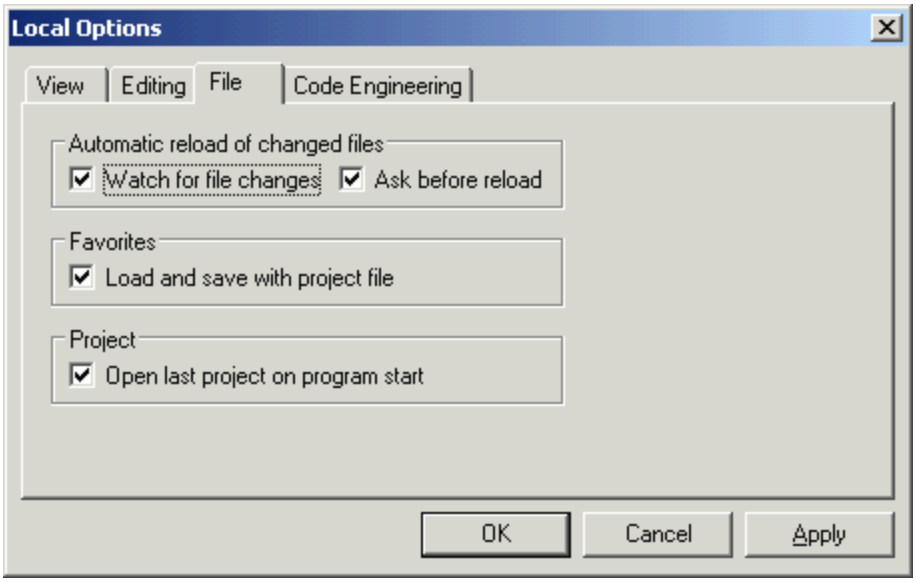
The **Editing** tab allows you to define:

- 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.
- 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 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.



The **File** tab allows you to define:

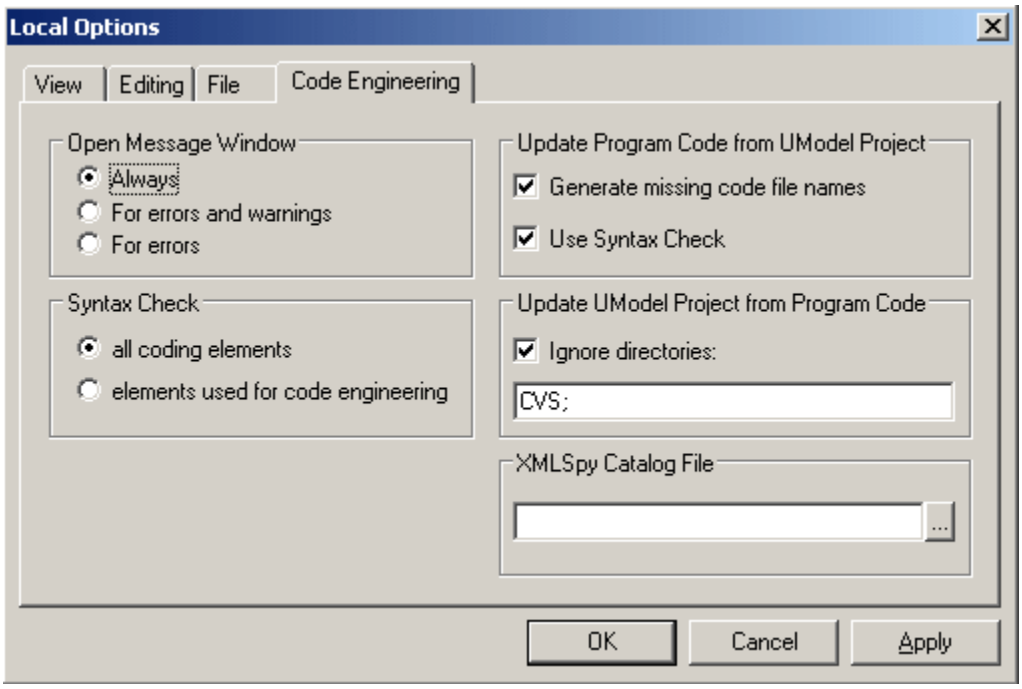
- The actions performed when files are changed.
- If the contents of the Favorites tab are to be loaded and saved with the current project.
- If the previously opened project is to automatically be opened when starting the application.



The **Code Engineering** tab allows you to define:

- The circumstances under which the Message window will open.
- If **all coding elements** i.e. those contained in a Java / C# namespace root, as well as those assigned to a Java / C# 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.
- If missing code file names in the merged code are to be generated.
- If a syntax check is to be performed when updating program code.
- 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.



12.7 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.

Next:
Switches to the next modeling diagram in the tab sequence, or the next hyperlinked element.

Previous:
Switches to the previous modeling diagram in the tab sequence, or the previous hyperlinked element.

Window list:
This list shows all currently open 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.

12.8 Help

Allows access to the Table of Contents and Index of the UModel documentation, as well as Altova web site links. The Registration option opens the Altova Licensing Manager, which contains the licensing information for all of Altova products.

Chapter 13

Code Generator

13 Code Generator

UModel includes a built-in code generator which can automatically generate Java, C#, or XML Schema files from UML models.

13.1 The way to SPL (Spy Programming Language)

This section gives an overview of Spy Programming Language, the 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 **...\UModelspl** folder. You can use these files as an aid to help you in developing your own templates.

How code generator works
Inputs to the code generator are the template files (.spl) and the object model provided by UModel. The template files contain SPL instructions for creating files, reading information from the object model and performing calculations, interspersed with literal code fragments in the target programming language.

The template file is interpreted by the code generator and outputs **.java, .cs** source code files, , or any other type of file depending on the template.

13.1.1 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$ñ=Z=QO=
$ñ=Z=$ñ=+=Nz
```

or

```
x$ñ=Z=QOW=$ñ=Z=$ñ=+=Nz
```

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 **]**.

13.1.2 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 statement

Variable types are declared by first assignment:

```
x$ñ=Z=Mz=
```
x is now an integer.

```
x$ñ=Z=?íÉëíëíêáåÖ?z=
```
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:

```
x$_~ëÉm~íÜ=Z=$çìíéìíé~íÜ=&=?L?=&=$g~î~m~Åâ~ÖÉaáêz
```

Objects
Objects represent the information contained in the UModelproject. 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:

```
Åä~ëë=xZ$Åä~ëëKN~ãÉz
```

This example outputs the word "class", followed by a space and the value of the **Name** property of the **$class** object.

The following table show the relationship between UML elements their SPL equivalents along with a short description.

Predefined variables

UML element	SPL property	Multi-plicity	UML	UModel	Description
			Attribute / Association	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	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
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		

UML element	SPL property	Multi-plicity	UML	UModel	Description
			Attribute / Association	Attribute / Association	Description
Interface	nestedClassifier	*	nestedClassifier: Classifier		
InterfaceRealization	contract	1	contract:Interface		
MultiplicityElement	lowerValue	0..1	lowerValue:Value Specification		
MultiplicityElement	upperValue	0..1	upperValue:Value Specification		
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>
NamedElement	isPackage			isPackage:Boolean	visibility <package>
NamedElement	namespacePrefix			namespacePrefix: String	XSD only - namespace prefix when exists
Namespace	elementImport	*	elementImport:Element Import		
Operation	ownedReturn Parameter	0..1		ownedReturnParameter: Parameter	parameter with direction return set
Operation	type	0..1		type	type of parameter with direction return set
Operation	ownedOperation Parameter	*		ownedOperation Parameter:Parameter	all parameters excluding parameter with direction return set
Package	namespace	*		namespace:Package	packages with code language <<namespace>> set
PackageableElement	owningPackage	0..1		owningPackage	set if owner is a package
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

UML element	SPL property	Multi-plicity	UML	UModel	Description
			Attribute / Association	Attribute / Association	Description
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:TemplateSignature		
TemplateBinding	parameterSubstitution	*	parameterSubstitution: TemplateParameterSubstitution		
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:
If you look into UModelSPL\C#[Java]**Default\Attribute.spl**, you can change the way how the name is written, e.g. replace

write $Property.name

by

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!

Please 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 Setting dialog box.

13.1.3 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

13.1.4 Conditions

SPL allows you to use standard "if" statements. The syntax is as follows:

```
áf=condition
    statements
ÉäëÉ
    statements
ÉåÇáf
```

or, without else:

```
áf=condition
    statements
ÉåÇáf
```

Please note that there are no round brackets enclosing the condition!
As in any other programming language, conditions are constructed with logical and comparison operators.

Example:

```
xáÑ=$å~ãÉëé~ÅÉK` çåí~áåëmìÄäáÅ` ä~ëëÉë=~åÇ=$å~ãÉëé~ÅÉKmêÉÑáñ=<>=??z
    ïÜ~íÉîÉê=óçì=ï~åí=x' áåëÉêîíë=ïÜ~íÉîÉê=óçì=ï~åí,=áå=íÜÉ=êÉëìäíáåÖ=ÑáäÉz
xÉåÇáÑz
```

Switch
SPL also contains a multiple choice statement.

Syntax:

```
ëwáíÅÜ=$variable
    Å~ëÉ=uW
        statements
    Å~ëÉ=YW
    Å~ëÉ=ZW
        statements
    ÇÉf~ìäíW
        statements
ÉåÇëwáíÅÜ
```

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.

13.1.5 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:

```
fçêÉ~ÅÜ=iterator=áå=collection
    statements
åÉñí
```

Example:

```
xfçêÉ~ÅÜ=$Åä~ëë=áå=$Åä~ëëÉë
    áf=åçí=$Åä~ëëKfëfåíÉêå~ä
        z   Åä~ëë=xZ$Åä~ëëKN~ãÉz;
x   ÉåÇáf
åÉñíz
```

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
Current	The current object (this is implicit if not specified and can be left out)

Example:

```
xfçêÉ~ÅÜ=$Éåìã=áå=$Ñ~ÅÉíKbåìãÉê~íáçå
    áf=åçí=$ÉåìãKfëcáêëí
        z,=x
    ÉåÇáf
    z?xZ$ÉåìãKV~äìÉz?x
åÉñíz
```

13.1.6 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)

Subroutine declaration

Subroutines

Syntax example:

```
pìÄ=páãéäÉpìÄEF
KKK=äáåÉë=çÑ=ÅçÇÉ
båÇpìÄ
```

- **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.

Please 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:

```
' =ÇÉÑáåÉ=ëìÄ=`çãéäÉíÉpìÄEF
xpìÄ=`çãéäÉíÉpìÄE=$é~ê~ã,=_óV~ä=$é~ê~ã_óV~äìÉ,=_óoÉÑ=$é~ê~ã_óoÉÑ=F
z=KKK
```

- **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:

```
' =ÇÉÑáåÉ=~=ÑìåÅíáçå
xpìÄ=j~âÉQì~äáÑáÉÇN~ãÉE=_óV~ä=$å~ãÉëé~ÅÉmêÉÑáñ,=_óV~ä=$äçÅ~äN~ãÉ=F
áÑ=$å~ãÉëé~ÅÉmêÉÑáñ=Z=??
==êÉíìêå=$äçÅ~äN~ãÉ
ÉäëÉ
==êÉíìêå=$å~ãÉëé~ÅÉmêÉÑáñ=&=?W?=&=$äçÅ~äN~ãÉ
ÉåÇáÑ
båÇpìÄ
z
```

Subroutine invocation

Use **call** to invoke a subroutine, followed by the procedure name and parameters, if any.

```
`~ää=páãéäÉpìÄEF
```

or,

```
`~ää=`çãéäÉíÉpìÄE=?cáêëím~ê~ãÉíÉê?,=$m~ê~ã_óV~äìÉ,=$m~ê~ã_óoÉÑ=F
```

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:

```
$QN~ãÉ=Z=j~âÉQì~äáÑáÉÇN~ãÉE$å~ãÉëé~ÅÉ,=?Éåíêó?F
```

13.2 Error Codes

Operating System Error Codes

201 File not found: '%s'
202 Cannot create file '%s'
203 Cannot open file '%s'
204 Cannot copy file '%s' to '%s'

Schema Error Codes

302 Validator: %s
303 Validator cannot load schema '%s'

Syntax Error Codes

401 Keyword expected
402 '%s' expected
403 No output file specified
404 Unexpected end of file
405 Keyword not allowed

Runtime Error Codes

501 Unknown variable '%s'
502 Redefinition of variable '%s'
503 Variable '%s' is not a container
504 Unknown property '%s'
505 Cannot convert from %s to %s
507 Unknown function
508 Function already defined
509 Invalid parameter
510 Division by zero
511 Unknown method
512 Incorrect number of parameters
513 Stack overflow

Chapter 14

Appendices

14 Appendices

These appendices contain technical information about UModel and important licensing information.

License Information

- Electronic software distribution
- Copyrights
- End User License Agreement

14.1 License Information

This section contains:

- Information about the distribution of this software product
- Information about the copyrights related to this software product
- The End User License Agreement governing the use of this software product

Please read this information carefully. It is binding upon you since you agreed to these terms when you installed this software product.

14.1.1 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 before making a purchasing decision.
- Once you decide to buy the software, you can place your order online at the Altova website and immediately 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 a comprehensive integrated onscreen help system. The latest version of the user manual is available at www.altova.com (i) in HTML format for online browsing, and (ii) in 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 this 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 have to purchase an End User License Agreement, which is delivered in the form of a key-code that you enter into the Software Activation dialog to unlock the product. You can purchase your license at the online shop at the Altova website.

Distributing the product
If you wish to share the product with others, please make sure that you distribute only the installation program, which is a convenient package that will install the application together with all sample files and the onscreen help. Any person that receives the product from you is also automatically entitled to a 30-day evaluation period. After the expiration of this period, any other user must also purchase a license in order to be able to continue using the product.

For further details, please refer to the End User License Agreement at the end of this section.

14.1.2 License Metering

Your Altova product has a built-in license metering module that 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, it 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. Other than that, it will not attempt to communicate over a network. If you are not connected to a LAN, or are using dial-up connections to connect to the Internet, the application will not generate any network traffic at all.

Multi 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 ensure 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.

Please note that your Altova product at no time attempts to send any information out of your LAN or over the Internet. 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* http://www.isi.edu/in-notes/iana/assignments/port-numbers *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.

You will also notice that, if you are online, your Altova product contains many useful functions; these are unrelated to the license-metering technology.

14.1.3 Copyright

All title and copyrights in this software product (including but not limited to images, photographs, animations, video, audio, music, text, and applets incorporated in the product), in the accompanying printed materials, and in any copies of these printed materials are owned by Altova GmbH or the respective supplier. This software product is protected by copyright laws and international treaty provisions.

- This software product ©1998-2007 Altova GmbH. All rights reserved.
- The Sentry Spelling-Checker Engine © 2000 Wintertree Software Inc.
- STLport © 1999, 2000 Boris Fomitchev, © 1994 Hewlett-Packard Company, © 1996, 1997 Silicon Graphics Computer Systems, Inc, © 1997 Moscow Center for SPARC Technology.
- Scintilla © 1998–2002 Neil Hodgson <åÉáäÜ@ëÅáåíáää~KçêÖ>.
- "ANTLR Copyright © 1989-2005 by Terence Parr (www.antlr.org)"

All other names or trademarks are the property of their respective owners.

14.1.4 Altova End User License Agreement

THIS IS A LEGAL DOCUMENT -- RETAIN FOR YOUR RECORDS

ALTOVA® END USER LICENSE AGREEMENT

Licensor:

Altova GmbH
Rudolfsplatz 13a/9
A-1010 Wien
Austria

Important - Read Carefully. Notice to User:

This End User License Agreement ("Software License Agreement")=is a legal document between you and Altova GmbH ("Altova"). It is important that you read this document before using the Altova-provided software ("Software") and any accompanying documentation, including, without limitation printed materials, 'online' files, or electronic documentation ("Documentation"). By clicking the "I accept" and "Next" buttons below, or by installing, or otherwise using the Software, you agree to be bound by the terms of this Software License Agreement as well as the Altova Privacy Policy ("Privacy Policy") including, without limitation, the warranty disclaimers, limitation of liability, data use and termination provisions below, whether or not you decide to purchase the Software. You agree that this agreement is enforceable like any written agreement negotiated and signed by you. If you do not agree, you are not licensed to use the Software, and you must destroy any downloaded copies of the Software in your possession or control. Please go to our Web site at http://www.altova.com/eula to download and print a copy of this Software License Agreement for your files and http://www.altova.com/privacy to review the privacy policy.

1. SOFTWARE LICENSE

(a) **License Grant.** Upon your acceptance of this Software License Agreement=Altova grants you a non-exclusive, non-transferable (except as provided below), limited license to install and use a copy of the Software on your compatible computer, up to the Permitted Number of computers. The Permitted Number of computers shall be delineated at such time as you elect to purchase the Software. During the evaluation period, hereinafter defined, only a single user may install and use the software on one computer. If you have licensed the Software as part of a suite of Altova software products (collectively, the "Suite") and have not installed each product individually, then the Software License Agreement governs your use of all of the software included in the Suite. If you have licensed SchemaAgent, then the terms and conditions of this Software License Agreement apply to your use of the SchemaAgent server software ("SchemaAgent Server") included therein, as applicable and you are licensed to use SchemaAgent Server solely in connection with your use of Altova Software and solely for the purposes described in the accompanying documentation. In addition, if you have licensed XMLSpy Enterprise Edition or MapForce Enterprise Edition, or UModel, your license to install and use a copy of the Software as provided herein permits you to generate source code based on (i) Altova Library modules that are included in the Software (such generated code hereinafter referred to as the "Restricted Source Code") and (ii) schemas or mappings that you create or provide (such code as may be generated from your schema or mapping source materials hereinafter referred to as the "Unrestricted Source Code"). In addition to the rights granted herein, Altova grants you a non-exclusive, non-transferable, limited license to compile into executable form the complete generated code comprised of the combination of the Restricted Source Code and the Unrestricted Source Code, and to use, copy, distribute or license that executable. You may not distribute or redistribute, sublicense, sell, or transfer to a third party the Restricted Source Code, unless said third party already has a license to the Restricted Source Code through their separate license agreement with Altova or other agreement with Altova. Altova reserves all other rights in and to the Software. With respect to the feature(s) of

UModel that permit reverse-engineering of your own source code or other source code that you have lawfully obtained, such use by you does not constitute a violation of this Agreement. Except as otherwise permitted in Section 1(h) reverse engineering of the Software is strictly prohibited as further detailed therein.

(b) **Server Use.** You may install one copy of the Software on your computer file server for the purpose of downloading and installing the Software onto other computers within your internal network up to the Permitted Number of computers. If you have licensed SchemaAgent, then you may install SchemaAgent Server on any server computer or workstation and use it in connection with your Software. No other network use is permitted, including without limitation using the Software either directly or through commands, data or instructions from or to a computer not part of your internal network, for Internet or Web-hosting services or by any user not licensed to use this copy of the Software through a valid license from Altova. If you have purchased Concurrent User Licenses as defined in Section 1(c) you may install a copy of the Software on a terminal server within your internal network for the sole and exclusive purpose of permitting individual users within your organization to access and use the Software through a terminal server session from another computer on the network provided that the total number of user that access or use the Software on such network or terminal server does not exceed the Permitted Number. Altova makes no warranties or representations about the performance of Altova software in a terminal server environment and the foregoing are expressly excluded from the limited warranty in Section 5 hereof and technical support is not available with respect to issues arising from use in such an environment.

(c) **Concurrent Use**. If you have licensed a "Concurrent-User" version of the Software, you may install the Software on any compatible computers, up to ten (10) times the Permitted Number of users, provided that only the Permitted Number of users actually use the Software at the same time. The Permitted Number of concurrent users shall be delineated at such time as you elect to purchase the Software licenses.

(d) **Backup and Archival Copies.** You may make one backup and one archival copy of the Software, provided your backup and archival copies are not installed or used on any computer and further provided that all such copies shall bear the original and unmodified copyright, patent and other intellectual property markings that appear on or in the Software. You may not transfer the rights to a backup or archival copy unless you transfer all rights in the Software as provided under Section 3.

(e) **Home Use.** You, as the primary user of the computer on which the Software is installed, may also install the Software on one of your home computers for your use. However, the Software may not be used on your home computer at the same time as the Software is being used on the primary computer.

(f) **Key Codes, Upgrades and Updates.** Prior to your purchase and as part of the registration for the thirty (30) -day evaluation period, as applicable, you will receive an evaluation key code. You will receive a purchase key code when you elect to purchase the Software from either Altova GMBH or an authorized reseller. The purchase key code will enable you to activate the Software beyond the initial evaluation period. You may not re-license, reproduce or distribute any key code except with the express written permission of Altova. If the Software that you have licensed is an upgrade or an update, then the update replaces all or part of the Software previously licensed. The update or upgrade and the associated license keys does not constitute the granting of a second license to the Software in that you may not use the upgrade or update in addition to the Software that it is replacing. You agree that use of the upgrade of update terminates your license to use the Software or portion thereof replaced.

(g) **Title.** Title to the Software is not transferred to you. Ownership of all copies of the Software and of copies made by you is vested in Altova, subject to the rights of use granted to you in this Software License Agreement. As between you and Altova, documents, files, stylesheets, generated program code (including the Unrestricted Source Code) and schemas that are authored or created by you via your utilization of the Software, in accordance with its Documentation and the terms of this Software License Agreement, are your property.

(h) **Reverse Engineering.** Except and to the limited extent as may be otherwise specifically provided by applicable law in the European Union, you may not reverse engineer, decompile, disassemble or otherwise attempt to discover the source code, underlying ideas,

underlying user interface techniques or algorithms of the Software by any means whatsoever, directly or indirectly, or disclose any of the foregoing, except to the extent you may be expressly permitted to decompile under applicable law in the European Union, if it is essential to do so in order to achieve operability of the Software with another software program, and you have first requested Altova to provide the information necessary to achieve such operability and Altova has not made such information available. Altova has the right to impose reasonable conditions and to request a reasonable fee before providing such information. Any information supplied by Altova or obtained by you, as permitted hereunder, may only be used by you for the purpose described herein and may not be disclosed to any third party or used to create any software which is substantially similar to the expression of the Software. Requests for information from users in the European Union with respect to the above should be directed to the Altova Customer Support Department.

(i) **Other Restrictions.** You may not loan, rent, lease, sublicense, distribute or otherwise transfer all or any portion of the Software to third parties except to the limited extent set forth in Section 3 or otherwise expressly provided. You may not copy the Software except as expressly set forth above, and any copies that you are permitted to make pursuant to this Software License Agreement must contain the same copyright, patent and other intellectual property markings that appear on or in the Software. You may not modify, adapt or translate the Software. You may not, directly or indirectly, encumber or suffer to exist any lien or security interest on the Software; knowingly take any action that would cause the Software to be placed in the public domain; or use the Software in any computer environment not specified in this Software License Agreement. You will comply with applicable law and Altova's instructions regarding the use of the Software. You agree to notify your employees and agents who may have access to the Software of the restrictions contained in this Software License Agreement and to ensure their compliance with these restrictions. YOU AGREE THAT YOU ARE SOLELY RESPONSIBLE FOR THE ACCURACY AND ADEQUACY OF THE SOFTWARE FOR YOUR INTENDED USE AND YOU WILL INDEMNIFY AND HOLD HARMLESS ALTOVA FROM ANY 3RD PARTY SUIT TO THE EXTENT BASED UPON THE ACCURACY AND ADEQUACY OF THE SOFTWARE IN YOUR USE. WITHOUT LIMITATION, THE SOFTWARE IS NOT INTENDED FOR USE IN THE OPERATION OF NUCLEAR FACILITIES, AIRCRAFT NAVIGATION, COMMUNICATION SYSTEMS OR AIR TRAFFIC CONTROL EQUIPMENT, WHERE THE FAILURE OF THE SOFTWARE COULD LEAD TO DEATH, PERSONAL INJURY OR SEVERE PHYSICAL OR ENVIRONMENTAL DAMAGE.

2. INTELLECTUAL PROPERTY RIGHTS

Acknowledgement of Altova's Rights. You acknowledge that the Software and any copies that you are authorized by Altova to make are the intellectual property of and are owned by Altova and its suppliers. The structure, organization and code of the Software are the valuable trade secrets and confidential information of Altova and its suppliers. The Software is protected by copyright, including without limitation by United States Copyright Law, international treaty provisions and applicable laws in the country in which it is being used. You acknowledge that Altova retains the ownership of all patents, copyrights, trade secrets, trademarks and other intellectual property rights pertaining to the Software, and that Altova's ownership rights extend to any images, photographs, animations, videos, audio, music, text and "applets" incorporated into the Software and all accompanying printed materials. You will take no actions which adversely affect Altova's intellectual property rights in the Software. Trademarks shall be used in accordance with accepted trademark practice, including identification of trademark owners' names. Trademarks may only be used to identify printed output produced by the Software, and such use of any trademark does not give you any right of ownership in that trademark. XMLSpy, Authentic, StyleVision, MapForce, Markup Your Mind, Axad, Nanonull, and Altova are trademarks of Altova GmbH (registered in numerous countries). Unicode and the Unicode Logo are trademarks of Unicode, Inc. Windows, Windows 95, Windows 98, Windows NT, Windows 2000 and Windows XP are trademarks of Microsoft. W3C, CSS, DOM, MathML, RDF, XHTML, XML and XSL are trademarks (registered in numerous countries) of the World Wide Web Consortium (W3C); marks of the W3C are registered and held by its host institutions, MIT, INRIA and Keio. Except as expressly stated above, this Software License Agreement does not

grant you any intellectual property rights in the Software. Notifications of claimed copyright infringement should be sent to Altova's copyright agent as further provided on the Altova Web Site.

3. LIMITED TRANSFER RIGHTS

Notwithstanding the foregoing, you may transfer all your rights to use the Software to another person or legal entity provided that: (a) you also transfer each of this Software License Agreement, the Software and all other software or hardware bundled or pre-installed with the Software, including all copies, updates and prior versions, and all copies of font software converted into other formats, to such person or entity; (b) you retain no copies, including backups and copies stored on a computer; (c) the receiving party secures a personalized key code from Altova; and (d) the receiving party accepts the terms and conditions of this Software License Agreement and any other terms and conditions upon which you legally purchased a license to the Software. Notwithstanding the foregoing, you may not transfer education, pre-release, or not-for-resale copies of the Software.

4. PRE-RELEASE AND EVALUATION PRODUCT ADDITIONAL TERMS

If the product you have received with this license is pre-commercial release or beta Software ("Pre-release Software"), then this Section applies. In addition, this section applies to all evaluation and/or demonstration copies of Altova software ("Evaluation Software") and continues in effect until you purchase a license. To the extent that any provision in this section is in conflict with any other term or condition in this Software License Agreement, this section shall supersede such other term(s) and condition(s) with respect to the Pre-release and/or Evaluation Software, but only to the extent necessary to resolve the conflict. You acknowledge that the Pre-release Software is a pre-release version, does not represent final product from Altova, and may contain bugs, errors and other problems that could cause system or other failures and data loss. CONSEQUENTLY, THE PRE-RELEASE AND/OR EVALUATION SOFTWARE IS PROVIDED TO YOU **"AS-IS" WITH NO WARRANTIES FOR USE OR PERFORMANCE,** AND ALTOVA DISCLAIMS ANY WARRANTY OR LIABILITY OBLIGATIONS TO YOU OF ANY KIND, WHETHER EXPRESS OR IMPLIED. WHERE LEGALLY LIABILITY CANNOT BE EXCLUDED FOR PRE-RELEASE AND/OR EVALUATION SOFTWARE, BUT IT MAY BE LIMITED, ALTOVA'S LIABILITY AND THAT OF ITS SUPPLIERS SHALL BE LIMITED TO THE SUM OF FIFTY DOLLARS (USD $50) IN TOTAL. If the Evaluation Software has a time-out feature, then the software will cease operation after the conclusion of the designated evaluation period. Upon such expiration date, your license will expire unless otherwise extended. Access to any files created with the Evaluation Software is entirely at your risk. You acknowledge that Altova has not promised or guaranteed to you that Pre-release Software will be announced or made available to anyone in the future, that Altova has no express or implied obligation to you to announce or introduce the Pre-release Software, and that Altova may not introduce a product similar to or compatible with the Pre-release Software. Accordingly, you acknowledge that any research or development that you perform regarding the Pre-release Software or any product associated with the Pre-release Software is done entirely at your own risk. During the term of this Software License Agreement, if requested by Altova, you will provide feedback to Altova regarding testing and use of the Pre-release Software, including error or bug reports. If you have been provided the Pre-release Software pursuant to a separate written agreement, your use of the Software is governed by such agreement. You may not sublicense, lease, loan, rent, distribute or otherwise transfer the Pre-release Software. Upon receipt of a later unreleased version of the Pre-release Software or release by Altova of a publicly released commercial version of the Software, whether as a stand-alone product or as part of a larger product, you agree to return or destroy all earlier Pre-release Software received from Altova and to abide by the terms of the license agreement for any such later versions of the Pre-release Software.

5. LIMITED WARRANTY AND LIMITATION OF LIABILITY

(a) **Limited Warranty and Customer Remedies.** Altova warrants to the person or entity

that first purchases a license for use of the Software pursuant to the terms of this Software License Agreement = that (i) the Software will perform substantially in accordance with any accompanying Documentation for a period of ninety (90) days from the date of receipt, and (ii) any support services provided by Altova shall be substantially as described in Section 6 of this agreement. Some states and jurisdictions do not allow limitations on duration of an implied warranty, so the above limitation may not apply to you. To the extent allowed by applicable law, implied warranties on the Software, if any, are limited to ninety (90) days. Altova's and its suppliers' entire liability and your exclusive remedy shall be, at Altova's option, either (i) return of the price paid, if any, or (ii) repair or replacement of the Software that does not meet Altova's Limited Warranty and which is returned to Altova with a copy of your receipt. This Limited Warranty is void if failure of the Software has resulted from accident, abuse, misapplication, abnormal use, Trojan horse, virus, or any other malicious external code. Any replacement Software will be warranted for the remainder of the original warranty period or thirty (30) days, whichever is longer. This limited warranty does not apply to Evaluation and/or Pre-release Software.

(b) **No Other Warranties and Disclaimer.** THE FOREGOING LIMITED WARRANTY AND REMEDIES STATE THE SOLE AND EXCLUSIVE REMEDIES FOR ALTOVA OR ITS SUPPLIER'S BREACH OF WARRANTY. ALTOVA AND ITS SUPPLIERS DO NOT AND CANNOT WARRANT THE PERFORMANCE OR RESULTS YOU MAY OBTAIN BY USING THE SOFTWARE. EXCEPT FOR THE FOREGOING LIMITED WARRANTY, AND FOR ANY WARRANTY, CONDITION, REPRESENTATION OR TERM TO THE EXTENT WHICH THE SAME CANNOT OR MAY NOT BE EXCLUDED OR LIMITED BY LAW APPLICABLE TO YOU IN YOUR JURISDICTION, ALTOVA AND ITS SUPPLIERS MAKE NO WARRANTIES, CONDITIONS, REPRESENTATIONS OR TERMS, EXPRESS OR IMPLIED, WHETHER BY STATUTE, COMMON LAW, CUSTOM, USAGE OR OTHERWISE AS TO ANY OTHER MATTERS. TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW, ALTOVA AND ITS SUPPLIERS DISCLAIM ALL OTHER WARRANTIES AND CONDITIONS, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, SATISFACTORY QUALITY, INFORMATIONAL CONTENT OR ACCURACY, QUIET ENJOYMENT, TITLE AND NON-INFRINGEMENT, WITH REGARD TO THE SOFTWARE, AND THE PROVISION OF OR FAILURE TO PROVIDE SUPPORT SERVICES. THIS LIMITED WARRANTY GIVES YOU SPECIFIC LEGAL RIGHTS. YOU MAY HAVE OTHERS, WHICH VARY FROM STATE/JURISDICTION TO STATE/JURISDICTION.

(c) **Limitation Of Liability.** TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW EVEN IF A REMEDY FAILS ITS ESSENTIAL PURPOSE, IN NO EVENT SHALL ALTOVA OR ITS SUPPLIERS BE LIABLE FOR ANY SPECIAL, INCIDENTAL, DIRECT, INDIRECT OR CONSEQUENTIAL DAMAGES WHATSOEVER (INCLUDING, WITHOUT LIMITATION, DAMAGES FOR LOSS OF BUSINESS PROFITS, BUSINESS INTERRUPTION, LOSS OF BUSINESS INFORMATION, OR ANY OTHER PECUNIARY LOSS) ARISING OUT OF THE USE OF OR INABILITY TO USE THE SOFTWARE OR THE PROVISION OF OR FAILURE TO PROVIDE SUPPORT SERVICES, EVEN IF ALTOVA HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. IN ANY CASE, ALTOVA'S ENTIRE LIABILITY UNDER ANY PROVISION OF THIS SOFTWARE LICENSE AGREEMENT SHALL BE LIMITED TO THE AMOUNT ACTUALLY PAID BY YOU FOR THE SOFTWARE PRODUCT. Because some states and jurisdictions do not allow the exclusion or limitation of liability, the above limitation may not apply to you. In such states and jurisdictions, Altova's liability shall be limited to the greatest extent permitted by law and the limitations or exclusions of warranties and liability contained herein do not prejudice applicable statutory consumer rights of person acquiring goods otherwise than in the course of business. The disclaimer and limited liability above are fundamental to this Software License Agreement between Altova and you.

(d) **Infringement Claims.** Altova will indemnify and hold you harmless and will defend or settle any claim, suit or proceeding brought against you by a third party that is based upon a claim that the content contained in the Software infringes a copyright or violates an intellectual

or proprietary right protected by United States or European Union law ("Claim"), but only to the extent the Claim arises directly out of the use of the Software and subject to the limitations set forth in Section 5 of this Agreement except as otherwise expressly provided. You must notify Altova in writing of any Claim within ten (10) business days after you first receive notice of the Claim, and you shall provide to Altova at no cost with such assistance and cooperation as Altova may reasonably request from time to time in connection with the defense of the Claim. Altova shall have sole control over any Claim (including, without limitation, the selection of counsel and the right to settle on your behalf on any terms Altova deems desirable in the sole exercise of its discretion). You may, at your sole cost, retain separate counsel and participate in the defense or settlement negotiations. Altova shall pay actual damages, costs, and attorney fees awarded against you (or payable by you pursuant to a settlement agreement) in connection with a Claim to the extent such direct damages and costs are not reimbursed to you by insurance or a third party, to an aggregate maximum equal to the purchase price of the Software. If the Software or its use becomes the subject of a Claim or its use is enjoined, or if in the opinion of Altova's legal counsel the Software is likely to become the subject of a Claim, Altova shall attempt to resolve the Claim by using commercially reasonable efforts to modify the Software or obtain a license to continue using the Software. If in the opinion of Altova's legal counsel the Claim, the injunction or potential Claim cannot be resolved through reasonable modification or licensing, Altova, at its own election, may terminate this Software License Agreement without penalty, and will refund to you on a pro rata basis any fees paid in advance by you to Altova. THE FOREGOING CONSTITUTES ALTOVA'S SOLE AND EXCLUSIVE LIABILITY FOR INTELLECTUAL PROPERTY INFRINGEMENT. This indemnity does not apply to infringements that would not be such, except for customer-supplied elements.

6. SUPPORT AND MAINTENANCE

Altova offers multiple optional "Support & Maintenance Package(s)" ("SMP") for the version of Software product edition that you have licensed, which you may elect to purchase in addition to your Software license. The Support Period, hereinafter defined, covered by such SMP shall be delineated at such time as you elect to purchase a SMP. Your rights with respect to support and maintenance as well as your upgrade eligibility depend on your decision to purchase SMP and the level of SMP that you have purchased:

(a) If you have not purchased SMP, you will receive the Software AS IS and will not receive any maintenance releases or updates. However, Altova, at its option and in its sole discretion on a case by case basis, may decide to offer maintenance releases to you as a courtesy, but these maintenance releases will not include any new features in excess of the feature set at the time of your purchase of the Software. In addition, Altova will provide free technical support to you for thirty (30) days after the date of your purchase (the "Support Period" for the purposes of this paragraph a), and Altova, in its sole discretion on a case by case basis, may also provide free courtesy technical support during your thirty (30)-day evaluation period. Technical support is provided via a Web-based support form only, and there is no guaranteed response time.

(b) If you have purchased SMP, then solely for the duration of its delineated Support Period, **you are eligible to receive the version of the Software edition** that you have licensed and all maintenance releases and updates for that edition that are released during your Support Period. For the duration of your SMP's Support Period, you will also be eligible to receive upgrades to the comparable edition of the next version of the Software that succeeds the Software edition that you have licensed for applicable upgrades released during your Support Period. The specific upgrade edition that you are eligible to receive based on your Support Period is further detailed in the SMP that you have purchased. Software that is introduced as separate product is not included in SMP. Maintenance releases, updates and upgrades may or may not include additional features. In addition, Altova will provide Priority Technical Support to you for the duration of the Support Period. Priority Technical Support is provided via a Web-based support form only, and Altova will make commercially reasonable efforts to respond via e-mail to all requests within forty-eight (48) hours during Altova's business hours (MO-FR, 8am UTC – 10pm UTC, Austrian and US holidays excluded) and to make reasonable efforts to provide work-arounds to errors reported in the Software.

During the Support Period you may also report any Software problem or error to Altova. If Altova determines that a reported reproducible material error in the Software exists and significantly impairs the usability and utility of the Software, Altova agrees to use reasonable commercial efforts to correct or provide a usable work-around solution in an upcoming maintenance release or update, which is made available at certain times at Altova's sole discretion.

If Altova, in its discretion, requests written verification of an error or malfunction discovered by you or requests supporting example files that exhibit the Software problem, you shall promptly provide such verification or files, by email, telecopy, or overnight mail, setting forth in reasonable detail the respects in which the Software fails to perform. You shall use reasonable efforts to cooperate in diagnosis or study of errors. Altova may include error corrections in maintenance releases, updates, or new major releases of the Software. Altova is not obligated to fix errors that are immaterial. Immaterial errors are those that do not significantly impact use of the Software. Whether or not you have purchased the Support & Maintenance Package, technical support only covers issues or questions resulting directly out of the operation of the Software and Altova will not provide you with generic consultation, assistance, or advice under any circumstances.

Updating Software may require the updating of software not covered by this Software License Agreement before installation. Updates of the operating system and application software not specifically covered by this Software License Agreement are your responsibility and will not be provided by Altova under this Software License Agreement. Altova's obligations under this Section 6 are contingent upon your proper use of the Software and your compliance with the terms and conditions of this Software License Agreement at all times. Altova shall be under no obligation to provide the above technical support if, in Altova's opinion, the Software has failed due to the following conditions: (i) damage caused by the relocation of the software to another location or CPU; (ii) alterations, modifications or attempts to change the Software without Altova's written approval; (iii) causes external to the Software, such as natural disasters, the failure or fluctuation of electrical power, or computer equipment failure; (iv) your failure to maintain the Software at Altova's specified release level; or (v) use of the Software with other software without Altova's prior written approval. It will be your sole responsibility to: (i) comply with all Altova-specified operating and troubleshooting procedures and then notify Altova immediately of Software malfunction and provide Altova with complete information thereof; (ii) provide for the security of your confidential information; (iii) establish and maintain backup systems and procedures necessary to reconstruct lost or altered files, data or programs.

7. SOFTWARE ACTIVATION, UPDATES AND LICENSE METERING

(a) **License Metering**. Altova has a built-in license metering module that helps you to avoid any unintentional violation of this Software License Agreement. Altova may use your internal network for license metering between installed versions of the Software.

(b) **Software Activation**. **Altova's 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 between your computer and the Altova license server. You agree that Altova may use these measures and you agree to follow any applicable requirements.**

(c) **LiveUpdate**. Altova provides a new LiveUpdate notification service to you, which is free of charge. Altova may use your internal network and Internet connection for the purpose of transmitting license-related data to an Altova-operated LiveUpdate server to validate your license at appropriate intervals and determine if there is any update available for you.

(d) **Use of Data.** The terms and conditions of the Privacy Policy are set out in full at http://www.altova.com/privacy and are incorporated by reference into this Software License Agreement. By your acceptance of the terms of this Software License Agreement or use of the Software, you authorize the collection, use and disclosure of information collected by Altova for the purposes provided for in this Software License Agreement and/or the Privacy Policy as

revised from time to time. European users understand and consent to the processing of personal information in the United States for the purposes described herein. Altova has the right in its sole discretion to amend this provision of the Software License Agreement and/or Privacy Policy at any time. You are encouraged to review the terms of the Privacy Policy as posted on the Altova Web site from time to time.

8. TERM AND TERMINATION

This Software License Agreement may be terminated (a) by your giving Altova written notice of termination; or (b) by Altova, at its option, giving you written notice of termination if you commit a breach of this Software License Agreement and fail to cure such breach within ten (10) days after notice from Altova or (c) at the request of an authorized Altova reseller in the event that you fail to make your license payment or other monies due and payable.. In addition the Software License Agreement governing your use of a previous version that you have upgraded or updated of the Software is terminated upon your acceptance of the terms and conditions of the Software License Agreement accompanying such upgrade or update. Upon any termination of the Software License Agreement, you must cease all use of the Software that it governs, destroy all copies then in your possession or control and take such other actions as Altova may reasonably request to ensure that no copies of the Software remain in your possession or control. The terms and conditions set forth in Sections 1(g), (h), (i), 2, 5(b), (c), 9, 10 and 11 survive termination as applicable.

9. RESTRICTED RIGHTS NOTICE AND EXPORT RESTRICTIONS

The Software was developed entirely at private expense and is commercial computer software provided with **RESTRICTED RIGHTS**. Use, duplication or disclosure by the U.S. Government or a U.S. Government contractor or subcontractor is subject to the restrictions set forth in this Agreement and as provided in FAR 12.211 and 12.212 (48 C.F.R. §12.211 and 12.212) or DFARS 227. 7202 (48 C.F.R. §227-7202) as applicable. Consistent with the above as applicable, Commercial Computer Software and Commercial Computer Documentation licensed to U.S. government end users only as commercial items and only with those rights as are granted to all other end users under the terms and conditions set forth in this Software License Agreement. Manufacturer is Altova GmbH, Rudolfsplatz, 13a/9, A-1010 Vienna, Austria/EU. You may not use or otherwise export or re-export the Software or Documentation except as authorized by United States law and the laws of the jurisdiction in which the Software was obtained. In particular, but without limitation, the Software or Documentation may not be exported or re-exported (i) into (or to a national or resident of) any U.S. embargoed country or (ii) to anyone on the U.S. Treasury Department's list of Specially Designated Nationals or the U.S. Department of Commerce's Table of Denial Orders. By using the Software, you represent and warrant that you are not located in, under control of, or a national or resident of any such country or on any such list.

10. = THIRD PARTY SOFTWARE

The Software may contain third party software which requires notices and/or additional terms and conditions. Such required third party software notices and/or additional terms and conditions are located Our Website at http://www.altova.com/legal_3rdparty.html and are made a part of and incorporated by reference into this Agreement. By accepting this Agreement, you are also accepting the additional terms and conditions, if any, set forth therein.

11. GENERAL PROVISIONS

If you are located in the European Union and are using the Software in the European Union and not in the United States, then this Software License Agreement will be governed by and construed in accordance with the laws of the Republic of Austria (excluding its conflict of laws principles and the U.N. Convention on Contracts for the International Sale of Goods) and you expressly agree that exclusive jurisdiction for any claim or dispute with Altova or relating in any way to your use of the Software resides in the Handelsgericht, Wien (Commercial Court,

Vienna) and you further agree and expressly consent to the exercise of personal jurisdiction in the Handelsgericht, Wien (Commercial Court, Vienna) in connection with any such dispute or claim.

If you are located in the United States or are using the Software in the United States then this Software License Agreement will be governed by and construed in accordance with the laws of the Commonwealth of Massachusetts, USA (excluding its conflict of laws principles and the U.N. Convention on Contracts for the International Sale of Goods) and you expressly agree that exclusive jurisdiction for any claim or dispute with Altova or relating in any way to your use of the Software resides in the federal or state courts of Massachusetts and you further agree and expressly consent to the exercise of personal jurisdiction in the federal or state courts of Massachusetts in connection with any such dispute or claim.

If you are located outside of the European Union or the United States and are not using the Software in the United States, then this Software License Agreement will be governed by and construed in accordance with the laws of the Republic of Austria (excluding its conflict of laws principles and the U.N. Convention on Contracts for the International Sale of Goods) and you expressly agree that exclusive jurisdiction for any claim or dispute with Altova or relating in any way to your use of the Software resides in the Handelsgericht, Wien (Commercial Court, Vienna) and you further agree and expressly consent to the exercise of personal jurisdiction in the Handelsgericht Wien (Commercial Court, Vienna) in connection with any such dispute or claim. This Software License Agreement will not be governed by the conflict of law rules of any jurisdiction or the United Nations Convention on Contracts for the International Sale of Goods, the application of which is expressly excluded.

This Software License Agreement contains the entire agreement and understanding of the parties with respect to the subject matter hereof, and supersedes all prior written and oral understandings of the parties with respect to the subject matter hereof. Any notice or other communication given under this Software License Agreement shall be in writing and shall have been properly given by either of us to the other if sent by certified or registered mail, return receipt requested, or by overnight courier to the address shown on Altova's Web site for Altova and the address shown in Altova's records for you, or such other address as the parties may designate by notice given in the manner set forth above. This Software License Agreement will bind and inure to the benefit of the parties and our respective heirs, personal and legal representatives, affiliates, successors and permitted assigns. The failure of either of us at any time to require performance of any provision hereof shall in no manner affect such party's right at a later time to enforce the same or any other term of this Software License Agreement. This Software License Agreement may be amended only by a document in writing signed by both of us. In the event of a breach or threatened breach of this Software License Agreement by either party, the other shall have all applicable equitable as well as legal remedies. Each party is duly authorized and empowered to enter into and perform this Software License Agreement. If, for any reason, any provision of this Software License Agreement is held invalid or otherwise unenforceable, such invalidity or unenforceability shall not affect the remainder of this Software License Agreement, and this Software License Agreement shall continue in full force and effect to the fullest extent allowed by law. The parties knowingly and expressly consent to the foregoing terms and conditions.

Last updated: 2006-09-05

Index

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z