

# **Altova UModel 2022 Basic Edition**

## **User & Reference Manual**

## **Altova UModel 2022 Basic Edition User & Reference Manual**

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

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

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

Published: 2022

© 2016-2022 Altova GmbH

---

# Table of Contents

<b>1</b>	<b>Introduction</b>	<b>10</b>
1.1	Support Notes.....	11
<b>2</b>	<b>UModel Tutorial</b>	<b>14</b>
2.1	Getting Started.....	15
2.2	Use Cases.....	18
2.3	Class Diagrams.....	27
2.3.1	Creating Derived Classes.....	36
2.4	Object Diagrams.....	42
2.5	Component Diagrams.....	49
2.6	Deployment Diagrams.....	55
2.7	Forward Engineering (from Model to Code).....	60
2.8	Reverse Engineering (from Code to Model).....	69
<b>3</b>	<b>UModel Graphical User Interface</b>	<b>77</b>
3.1	Model Tree Window.....	79
3.2	Diagram Tree Window.....	83
3.3	Favorites Window.....	84
3.4	Properties Window.....	85
3.5	Styles Window.....	86
3.6	Hierarchy Window.....	87
3.7	Overview Window.....	89
3.8	Documentation Window.....	90
3.9	Messages Window.....	91
3.10	Diagram Window.....	93
3.11	Diagram Pane.....	94

---

<b>4</b>	<b>UModel Command Line Interface</b>	<b>96</b>
4.1	Creating, Loading, and Saving Projects in Batch Mode.....	101
<b>5</b>	<b>How to Model...</b>	<b>103</b>
5.1	Elements.....	104
5.1.1	Creating Elements.....	104
5.1.2	Inserting Elements from the Model into a Diagram.....	105
5.1.3	Renaming, Moving, and Copying Elements.....	107
5.1.4	Deleting Elements.....	108
5.1.5	Converting Elements.....	109
5.1.6	Finding and Replacing Text.....	109
5.1.7	Checking Where and If Elements Are Used.....	111
5.1.8	Constraining Elements.....	112
5.1.9	Hyperlinking Elements.....	113
5.1.10	Documenting Elements.....	116
5.1.11	Changing the Style of Elements.....	117
5.2	Diagrams.....	119
5.2.1	Creating Diagrams.....	119
5.2.2	Generating Diagrams.....	120
5.2.3	Opening Diagrams.....	122
5.2.4	Deleting Diagrams.....	123
5.2.5	Changing the Style of Diagrams.....	123
5.2.6	Aligning and Resizing Modeling Elements.....	125
5.2.7	Type Autocompletion in Classes.....	127
5.2.8	Zooming into/out of Diagrams.....	129
5.3	Relationships.....	130
5.3.1	Creating Relationships.....	130
5.3.2	Changing the Style of Lines and Relationships.....	131
5.3.3	Viewing Element Relationships.....	133
5.3.4	Associations.....	133
5.3.5	Collection Associations.....	136
5.3.6	Containment.....	139

---

5.4	Stereotypes and Tagged Values.....	140
5.4.1	Tagged Values.....	141
5.4.2	Applying Stereotypes.....	142
5.4.3	Showing or Hiding Tagged Values.....	144

## 6 Projects and Code Engineering 147

6.1	Managing UModel Projects.....	148
6.1.1	Creating, Opening, and Saving Projects.....	148
6.1.2	Opening Projects from a URL.....	149
6.1.3	Moving Projects to a New Directory.....	153
6.1.4	Applying UModel Profiles.....	154
6.1.5	Splitting UModel Projects.....	155
6.1.6	Including Subprojects.....	158
6.1.7	Sharing Packages and Diagrams.....	160
6.1.8	Tips for Enhancing Performance.....	163
6.2	Generating Program Code.....	164
6.2.1	Setting a Package as Namespace Root.....	164
6.2.2	Adding a Code Engineering Component.....	165
6.2.3	Checking Project Syntax.....	167
6.2.4	Code Generation Options.....	169
6.2.5	Example: Generate C# Code.....	171
6.2.6	Example: Generate Java Code.....	176
6.2.7	SPL Templates.....	185
6.3	Importing Source Code.....	187
6.3.1	Code Import Options.....	189
6.3.2	Example: Import a C# Project.....	191
6.4	Importing Java, C# and VB.NET Binaries.....	199
6.4.1	Adding Custom Java Runtimes.....	200
6.4.2	Import Binary Options.....	200
6.4.3	Example: Import .NET Assemblies.....	204
6.4.4	Example: Import Java .class Files.....	206
6.5	Synchronizing the Model and Source Code.....	212
6.5.1	Synchronization Tips.....	213
6.5.2	Refactoring Code and Synchronization.....	215

---

6.5.3	Code Synchronization Settings.....	216
6.6	UModel Element Mappings.....	219
6.6.1	C# Mappings.....	219
6.6.2	VB.NET Mappings.....	239
6.6.3	Java Mappings.....	253
6.6.4	XML Schema Mappings.....	259
6.7	Merging UModel Projects.....	269
6.7.1	3-Way Project Merge.....	269
6.7.2	Example: Manual 3-Way Project Merge.....	271
6.8	UML Templates.....	274
6.8.1	Template Signatures.....	275
6.8.2	Template Binding.....	276
6.8.3	Template Usage in Operations and Properties.....	276

## **7 Generating UML Documentation 278**

7.1	Documentation Generation Options.....	282
7.2	Customizing Output with StyleVision.....	287

## **8 UML Diagrams 289**

8.1	Behavioral Diagrams.....	290
8.1.1	Activity Diagram.....	290
8.1.2	State Machine Diagram.....	307
8.1.3	Protocol State Machine.....	330
8.1.4	Use Case Diagram.....	335
8.1.5	Communication Diagram.....	335
8.1.6	Interaction Overview Diagram.....	339
8.1.7	Sequence Diagram.....	344
8.1.8	Timing Diagram.....	371
8.2	Structural Diagrams.....	380
8.2.1	Class Diagram.....	380
8.2.2	Composite Structure Diagram.....	394
8.2.3	Component Diagram.....	397
8.2.4	Deployment Diagram.....	397

---

8.2.5	Object Diagram.....	398
8.2.6	Package Diagram.....	398
8.2.7	Profile Diagram.....	404
8.3	Additional Diagrams.....	417
8.3.1	XML Schema Diagrams.....	417

## 9 XMI - XML Metadata Interchange 435

## 10 Source Control 437

10.1	Setting Up Source Control.....	439
10.2	Supported Source Control Systems.....	440
10.3	Source Control Commands.....	442
10.3.1	Open from Source Control.....	442
10.3.2	Enable Source Control.....	445
10.3.3	Get Latest Version.....	446
10.3.4	Get .....	446
10.3.5	Get Folder(s).....	447
10.3.6	Check Out.....	448
10.3.7	Check In.....	450
10.3.8	Undo Check Out.....	450
10.3.9	Add to Source Control.....	452
10.3.10	Remove from Source Control.....	454
10.3.11	Share from Source Control.....	455
10.3.12	Show History.....	456
10.3.13	Show Differences.....	458
10.3.14	Show Properties.....	459
10.3.15	Refresh Status.....	460
10.3.16	Source Control Manager.....	460
10.3.17	Change Source Control.....	460
10.4	Source Control with Git.....	462
10.4.1	Enabling Git Source Control with GIT SCC Plug-in.....	463
10.4.2	Adding a Project to Git Source Control.....	463
10.4.3	Cloning a Project from Git Source Control.....	465

---

## **11 UModel Diagram icons** **467**

11.1	Activity Diagram.....	468
11.2	Class Diagram.....	470
11.3	Communication diagram.....	471
11.4	Composite Structure Diagram.....	472
11.5	Component Diagram.....	473
11.6	Deployment Diagram.....	474
11.7	Interaction Overview diagram.....	475
11.8	Object Diagram.....	476
11.9	Package diagram.....	477
11.10	Profile Diagram.....	478
11.11	Protocol State Machine.....	479
11.12	Sequence Diagram.....	480
11.13	State Machine Diagram.....	481
11.14	Timing Diagram.....	482
11.15	Use Case diagram.....	483
11.16	XML Schema diagram.....	484

## **12 Menu Reference** **485**

12.1	File .....	486
12.2	Edit .....	488
12.3	Project.....	490
12.4	Layout.....	493
12.5	View.....	494
12.6	Tools.....	495
12.6.1	User-defined Tools.....	495
12.6.2	Customize.....	495
12.6.3	Restore Toolbars and Windows.....	504
12.6.4	Options.....	505
12.7	Window.....	514
12.8	Help .....	516

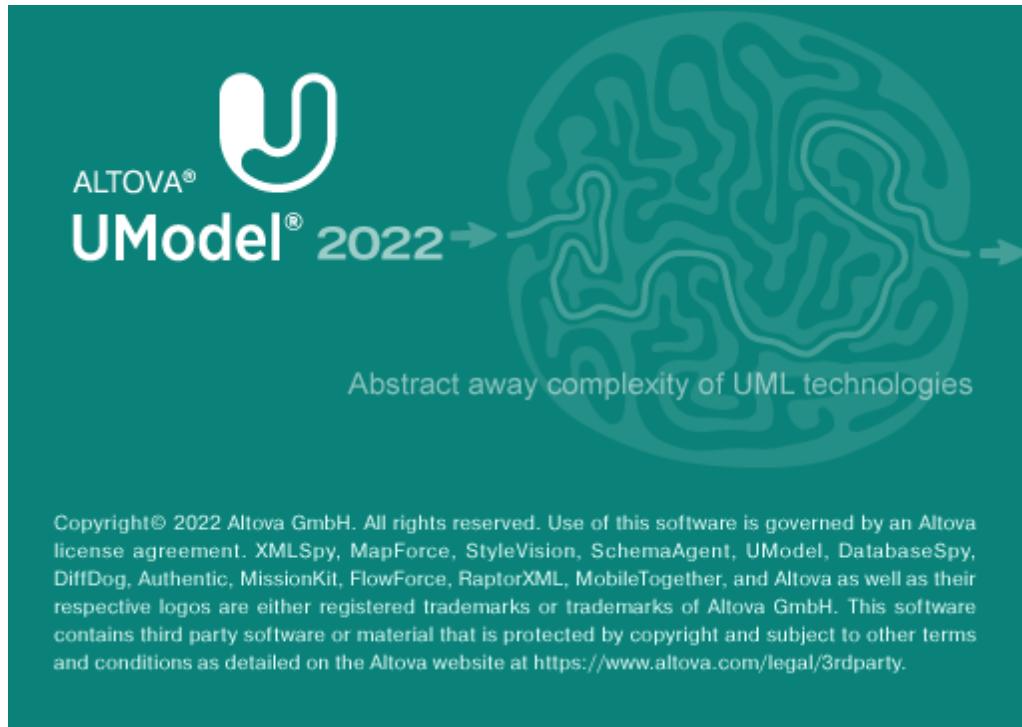
---

<b>13 SPL Reference</b>	<b>521</b>
13.1 Basic SPL structure.....	522
13.2 Variables.....	523
13.3 Operators.....	532
13.4 Conditions.....	533
13.5 Collections and foreach.....	534
13.6 Subroutines.....	536
13.6.1 Subroutine declaration.....	536
13.6.2 Subroutine invocation.....	537
<b>14 License Information</b>	<b>538</b>
14.1 Electronic Software Distribution.....	539
14.2 Software Activation and License Metering.....	540
14.3 Altova End-User License Agreement.....	542
<b>Index</b>	<b>543</b>

## 1 Introduction

Altova website:  [UML tool](#)

**Altova UModel 2022 Basic Edition** is a UML modeling application with a rich visual interface and superior usability features to help level the UML learning curve. UModel includes many high-end functions to empower users with the most practical aspects of the UML 2.5 specification. UModel is a 32/64-bit Windows application that runs on Windows 7 SP1 with Platform Update, Windows 8, Windows 10, Windows 11, and Windows Server 2008 R2 SP1 with Platform Update or newer. 64-bit support is available for the Enterprise and Professional editions. For an overview of UModel capabilities, see [Support Notes](#) (11).



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

*Last updated: 28 February 2022*

## 1.1 Support Notes

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

- Windows Server 2008 R2 SP1 with Platform Update or newer
- Windows 7 SP1 with Platform Update, Windows 8, Windows 10, Windows 11

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

### UML diagrams

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

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

\* Available in UModel Enterprise and Professional editions.

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

- UModel diagrams can be created in any order, and at any time; there is no need to follow a prescribed sequence during modeling.
- The syntax coloring in diagrams is customizable. For example, you can customize modeling elements and their properties (font, color, borders, etc.) in a hierarchical fashion at the project, node/line, element family and element level, see [Changing the Style of Elements](#)<sup>117</sup>.
- The unlimited levels of Undo/Redo track not only content changes, but also all style changes made to any model element.
- Modeling elements support hyperlinks, see [Hyperlinking Elements](#)<sup>113</sup>.

### Code engineering and import of binaries

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

Language	Code engineering	Import of binaries
C#	1.2, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 7.1, 7.2, 7.3, 8.0, 9.0 <sup>1</sup> , 10	Same language versions as for code engineering <sup>2</sup>
Java	1.4, 5.0 (1.5), 6 (1.6), 7 (1.7), 8 (1.8), 9 (1.9), 10, 11, 12, 13, 14, 15, 16	Same language versions as for code engineering <sup>3</sup>
Visual Basic .NET	7.1, 8.0, 9.0	Same language versions as for code engineering
XML Schemas <sup>4</sup>	1.0	Not applicable

Table footnotes:

1. If you import binary files compiled from C# 9.0 code, note that any *records* will be imported as *classes*. This limitation is due to the fact that records are marked as classes in the assembly, which makes it impossible to distinguish them from classes.
2. C# code engineering and import of binaries include support for .NET Framework, .NET Core, .NET 5, and .NET 6. Note that .NET Framework, .NET Core, .NET 5 or .NET 6 must be installed, as applicable. Binaries of other .NET implementations which are not mentioned are likely to be imported as well. See also [Importing Java, C# and VB.NET Binaries](#)<sup>199</sup>.
3. It is also possible to import binaries targeting Java Virtual Machines other than Oracle JDK, such as OpenJDK, SapMachine, Liberica JDK, and others, see [Adding Custom Java Runtimes](#)<sup>200</sup>.
4. In the case of XML Schemas, code engineering means that you can import a schema (or multiple schemas from a directory) into UModel, view or modify the model, and write the changes back to the schema file. When you synchronize data from the model to a schema file, the schema file is always overwritten by the model. See also [XML Schema Diagrams](#)<sup>417</sup>.

General notes:

- You can synchronize the code and model at the project, package, or even class level. UModel does not require that pseudo-code, or comments in the generated code be present, in order to accomplish round-trip engineering.
- A single project can support Java, C#, or VB.NET code simultaneously.
- UModel supports the use of UML templates and their mapping to or from Java, C# and Visual Basic generics.
- While importing source code, you can optionally generate [Class](#)<sup>392</sup> and [Package](#)<sup>401</sup> diagrams. Once the source code is imported into the model, you can also generate [Sequence](#)<sup>359</sup> diagrams.
- You can generate program code from [Sequence diagrams](#)<sup>365</sup> and from [State Machine diagrams](#)<sup>319</sup>.
- UModel projects can be split up into multiple sub-projects allowing several developers to simultaneously edit different parts of a single project. You can then reintegrate the changes back into a common model. You can also merge UModel projects, as a 2-way or as a 3-way merge, see [Merging UModel Projects](#)<sup>269</sup>.
- Code generation in UModel is based on Spy Programming Language (SPL) templates and is customizable.

## UML documentation generation

You can generate documentation from UModel projects in HTML, RTF, Microsoft Word 2000 or later formats. Various options are available that let you configure the level of detail of generated documentation, the look and feel, and other preferences. Generating documentation in PDF format and deep customization of document

generation templates is possible with Altova StyleVision (<https://www.altova.com/stylevision>). For more information, see [Generating UML Documentation](#)<sup>278</sup>.

## Interoperability

UModel also provides support for importing or exporting projects to or from XML Metadata Interchange (XMI) format, see [XMI - XML Metadata Interchange](#)<sup>435</sup>.

## 2 UModel Tutorial

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

The tutorial is organized into sections as shown below. In the initial sections of this tutorial you will be working with a sample project pre-installed with UModel. If you would like to quickly create a new modelling project from scratch with UModel, you can skip directly to [Forward Engineering \(from Model to Code\)](#) 60.

- [Getting Started](#) 15
- [Use Cases](#) 18
- [Class Diagrams](#) 27
- [Creating Derived Classes](#) 36
- [Object Diagrams](#) 42
- [Component Diagrams](#) 49
- [Deployment Diagrams](#) 55
- [Forward Engineering \(from Model to Code\)](#) 60
- [Reverse Engineering \(from Code to Model\)](#) 69

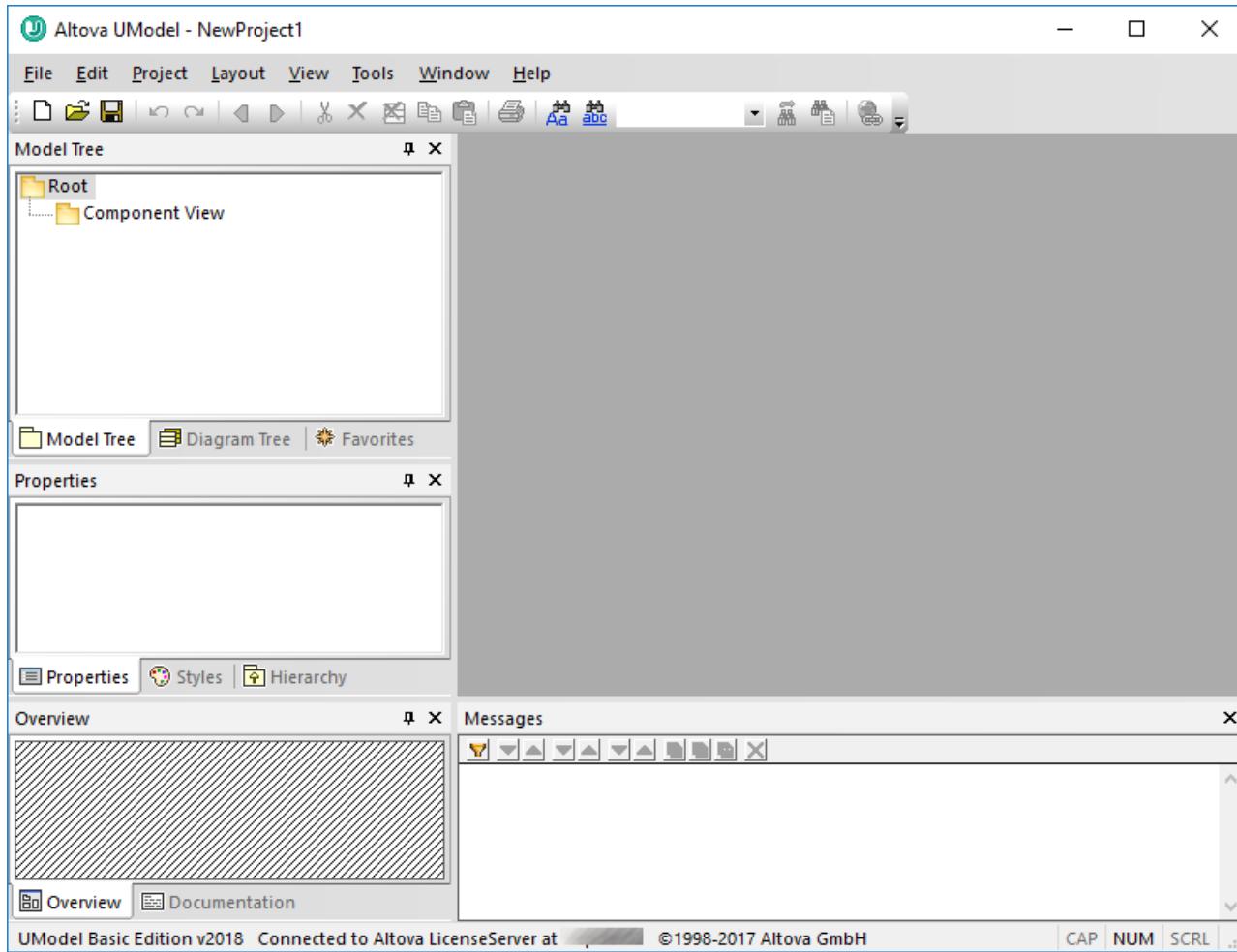
This tutorial makes use of the following sample UModel project files available in the directory **C:\Users\<username>\Documents\Altova\UModel2022\UModel\Examples\Tutorial**:

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

**Note:** All UModel example files are initially available in the directory **C:\ProgramData\Altova\UModel2022**. When any user starts the application for the first time, the example files are copied to **C:\Users\<username>\Documents\Altova\UModel2022\UModel\Examples**. Therefore, do not move, edit, or delete the example files in the initial directory.

## 2.1 Getting Started

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



### UModel Graphical User Interface

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

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

- The **Model Tree** window contains and displays all modeling elements of your UModel project. Elements can be directly manipulated in this window using the standard editing keys as well as drag and drop.
- The **Diagram Tree** window allows your quick access to the modeling diagrams of your project wherever they may be in the project structure. Diagrams are grouped according to their diagram type.

- The **Favorites** window is a user-definable repository of modeling elements. Any type of modeling element can be placed in this window using the "Add to Favorites" command of the context menu.

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

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

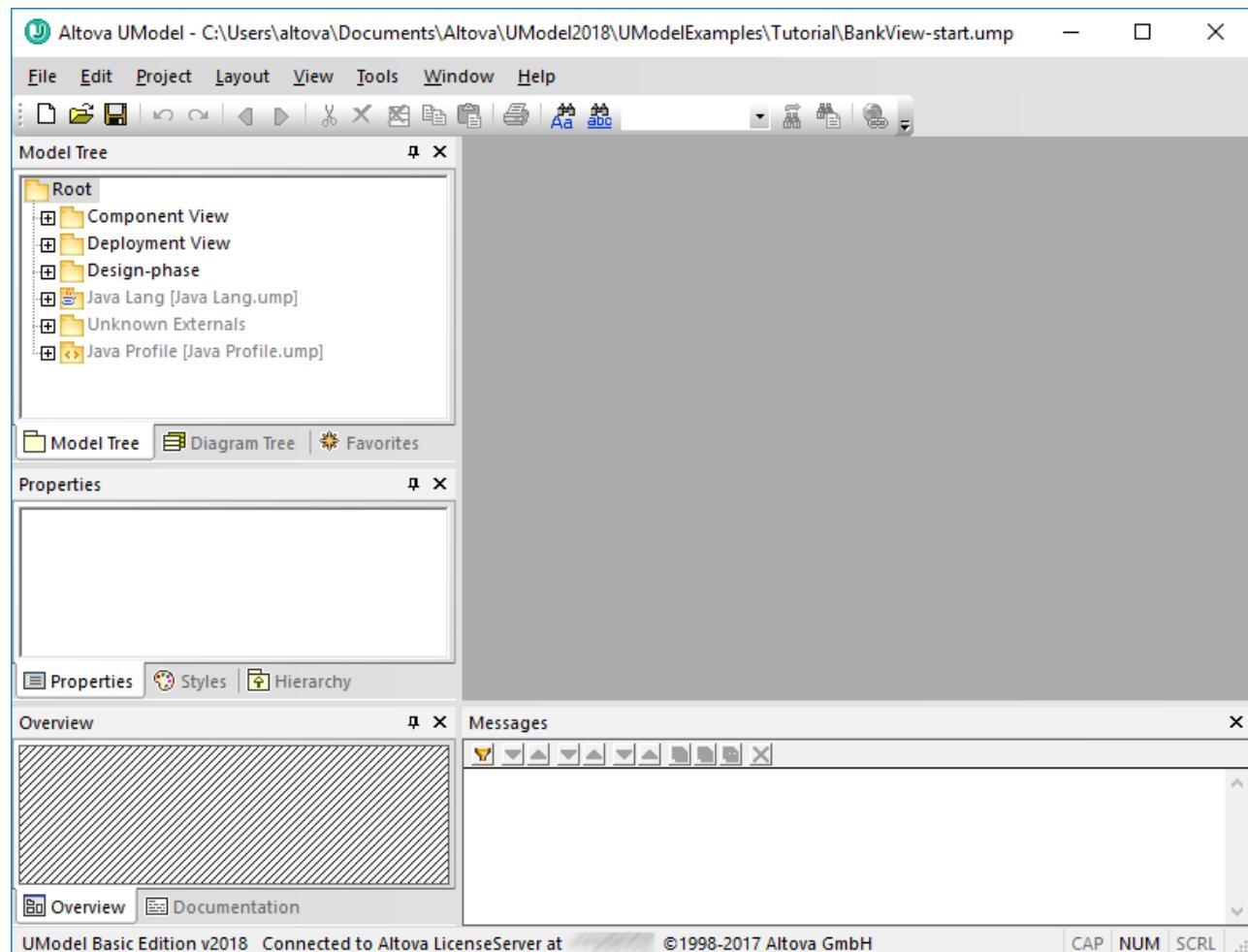
The helper windows in the lower-left area are as follows:

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

In this tutorial, you will be working mostly within the **Model Tree** and **Diagram Tree** windows, as well as the main diagram window. For further information about the graphical user interface elements, see [UModel User Interface](#) 77.

#### To open the tutorial project:

1. Select the menu option **File | Open** and navigate to the ...\\UModel\\Examples\\Tutorial folder of UModel. Note that you can also open a \*.ump file through a URL, please see [Switch to URL](#) 486 for more information.
2. Open the **BankView-start.ump** project file. The project file is now loaded into UModel. Several predefined packages are now visible under the Root package. Note that the main window is empty at the moment.



Bank View-start.ump project

## 2.2 Use Cases

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

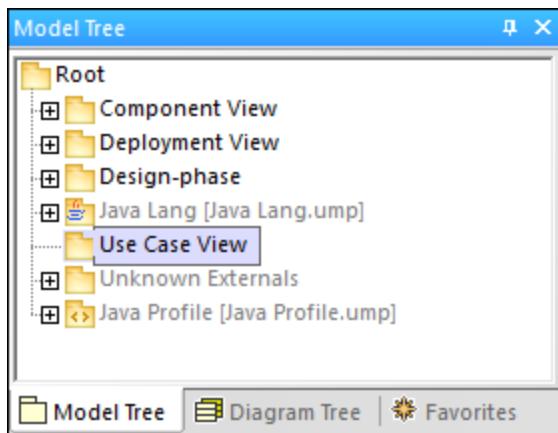
- Add a new package to the project
- Add a new use case diagram to the project
- Add use case elements to the diagram, and define the dependencies amongst them
- Align and adjust the size of elements in the diagram
- Change the style of all diagrams in a UModel project.

To proceed, run UModel and open the **BankView-start.ump** project (see also [Opening the Tutorial Project](#)<sup>15</sup>).

### Adding a new package to a project

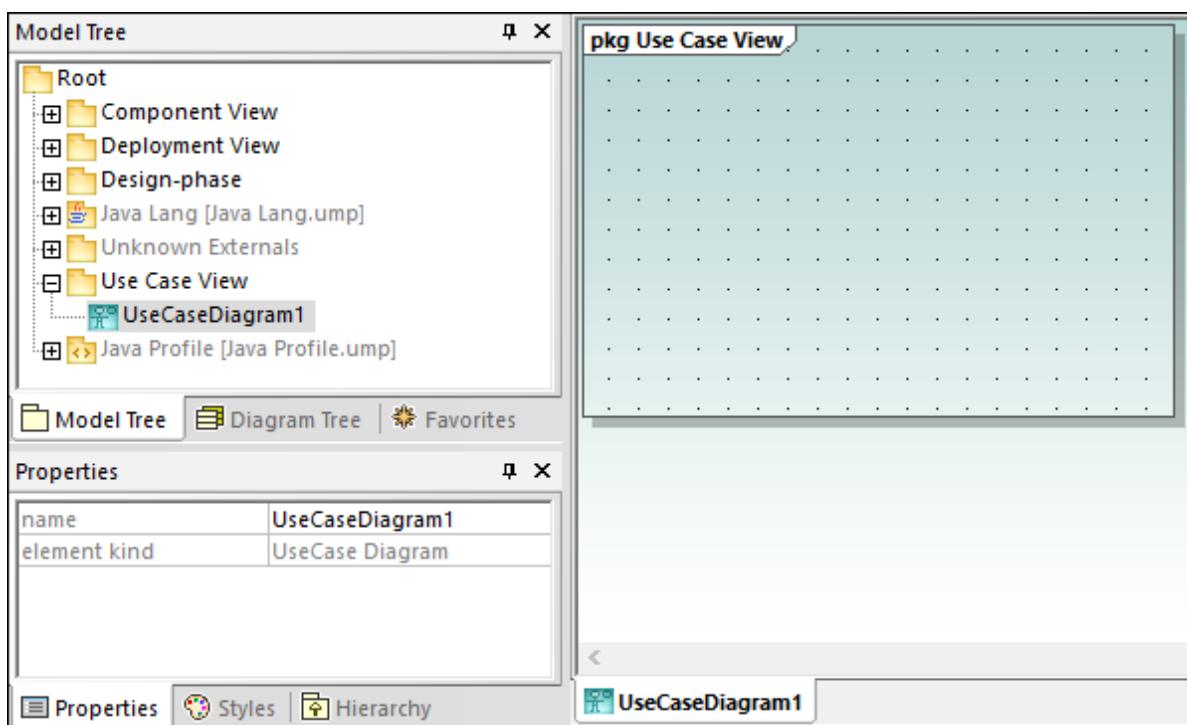
As you already know from UML, a package is a container for organizing classes and other UML elements, including use cases. Let's begin by creating a package that will store a new use case diagram. Note that UModel does not require that a specific diagram must reside in a specific package; however, you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

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



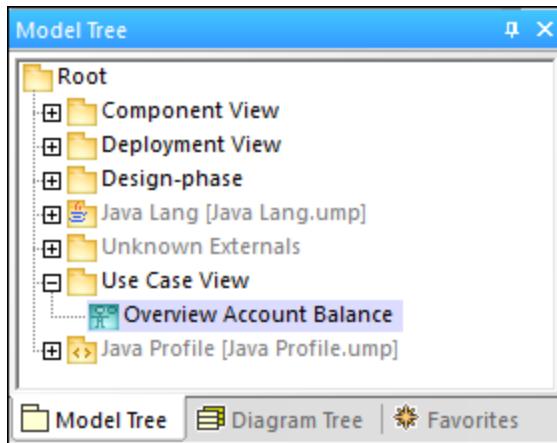
### Adding a Use Case diagram to a package

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



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

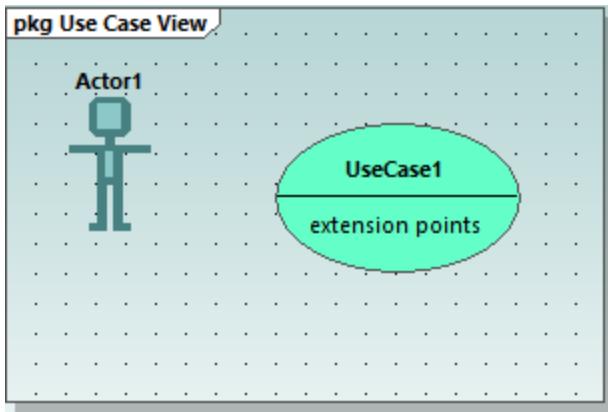
3. Double-click the diagram name in the **Model Tree** window, change it to "Overview Account Balance", and press **Enter** to confirm.



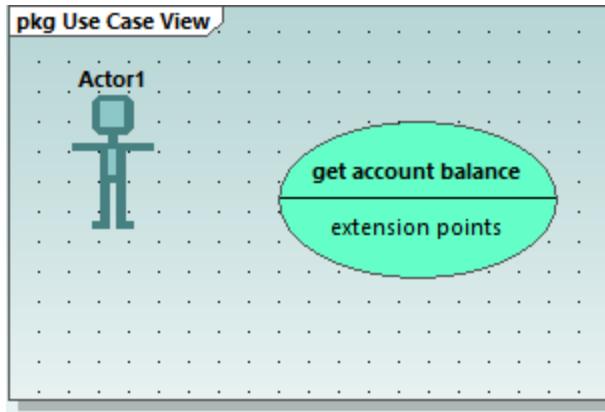
## Adding Use Case elements to the Use Case diagram

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

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



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

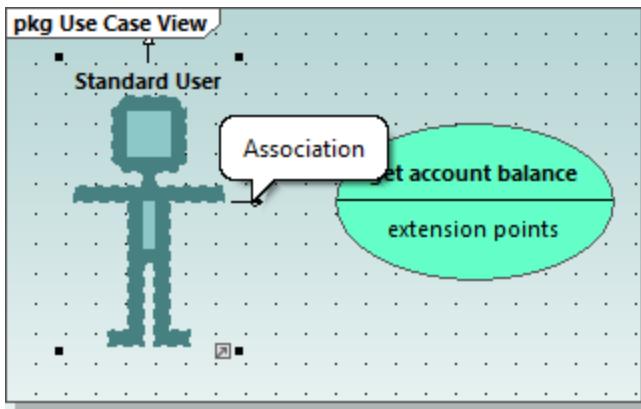


**Note:** To create a multi-line use case name, press **Enter** while holding the **Ctrl** key pressed.

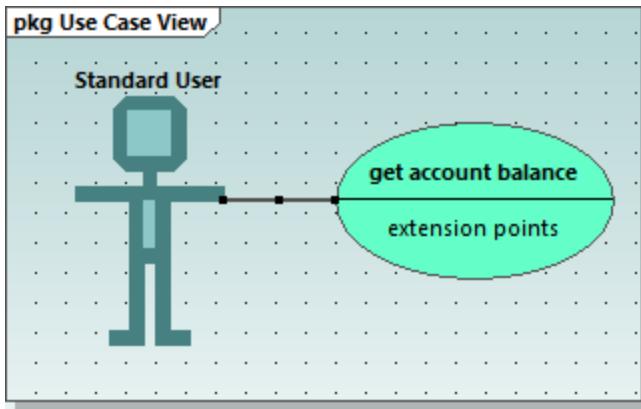
### Manipulating UModel elements: handles and compartments

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

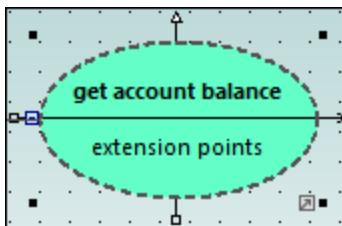
1. Double-click the "Actor1" text of the Actor element, change the name to "Standard User" and press **Enter** to confirm.
2. Place the mouse cursor over the handle to the right of the actor. A tooltip containing "Association" appears.



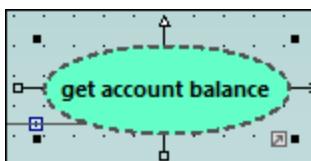
3. Click the handle, drag the Association line to the right, and drop it on the "get account balance" use case. An association has now been created between the actor and the use case. The association properties are also visible in the Properties window. The new association has been added to Model Tree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



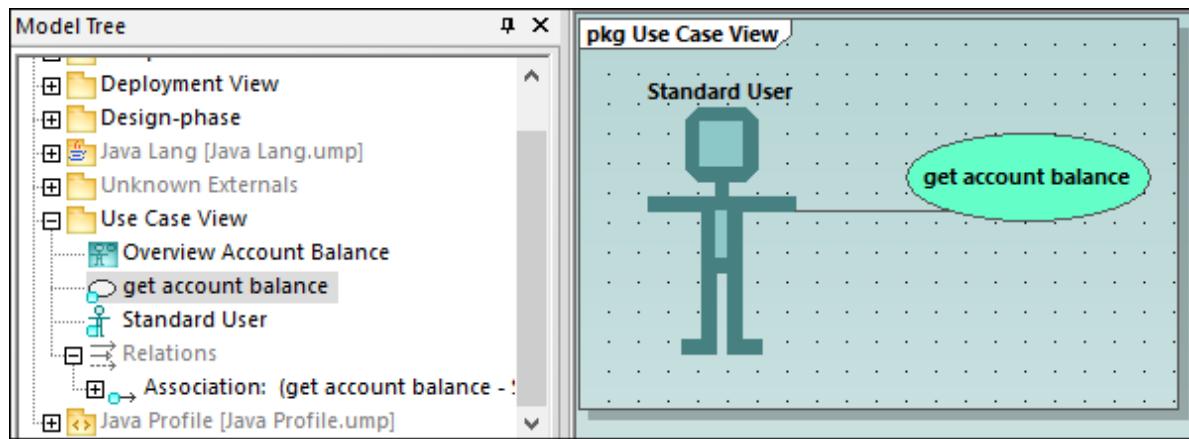
4. Click the use case and drag it to the right to reposition it. The association properties are visible on the association object.
5. Click the use case to select it, then click the **collapse icon** on the left edge of the ellipse.



The "extension points" compartment is now hidden.



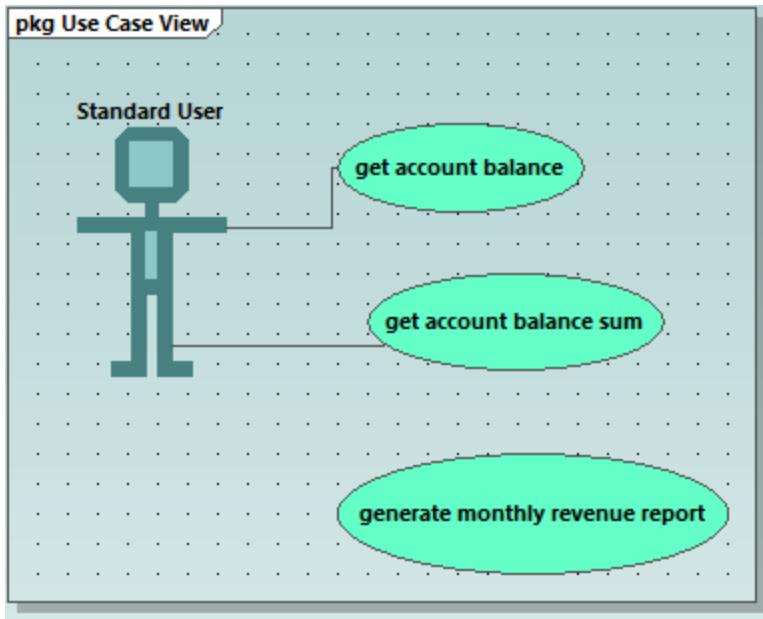
A blue dot next to an element in the Model Tree window signifies that the element is visible in the current diagram. For example, in the image below, three elements are currently visible in the diagram and thus have a blue dot in the Model Tree:



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

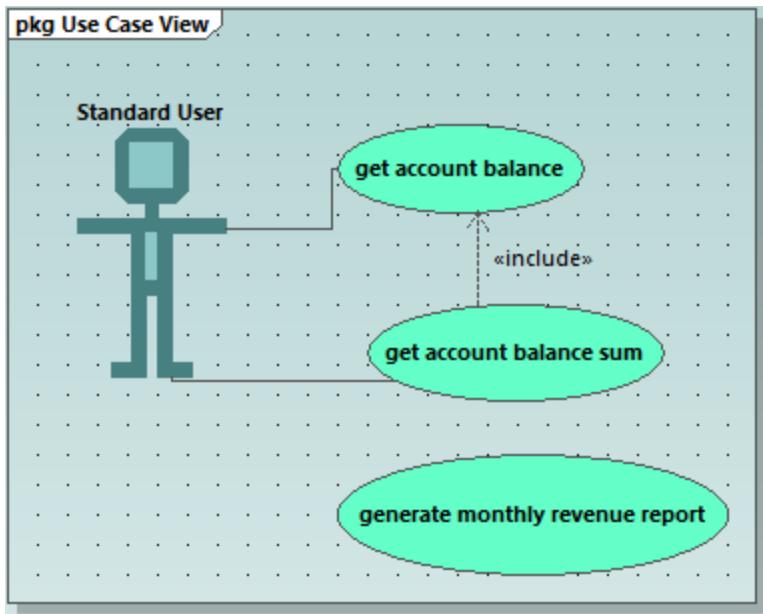
#### To finish up the Use Case diagram:

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



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

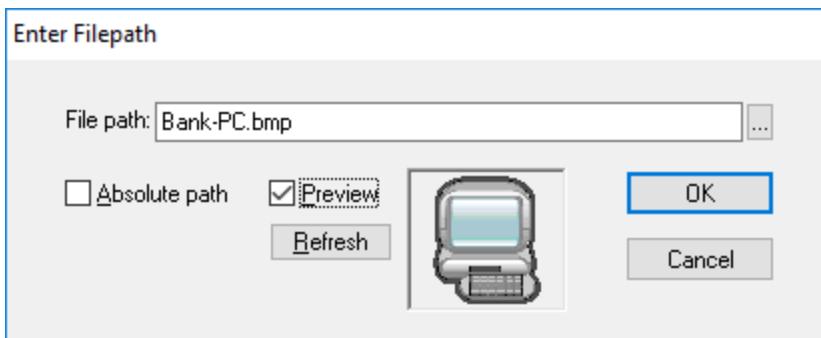
- Click the **Include** handle of the "get account balance sum" use case, at the bottom of the ellipse, and drop the dependency on "get account balance". An "include" dependency is created, and the include stereotype is displayed on the dotted arrow.



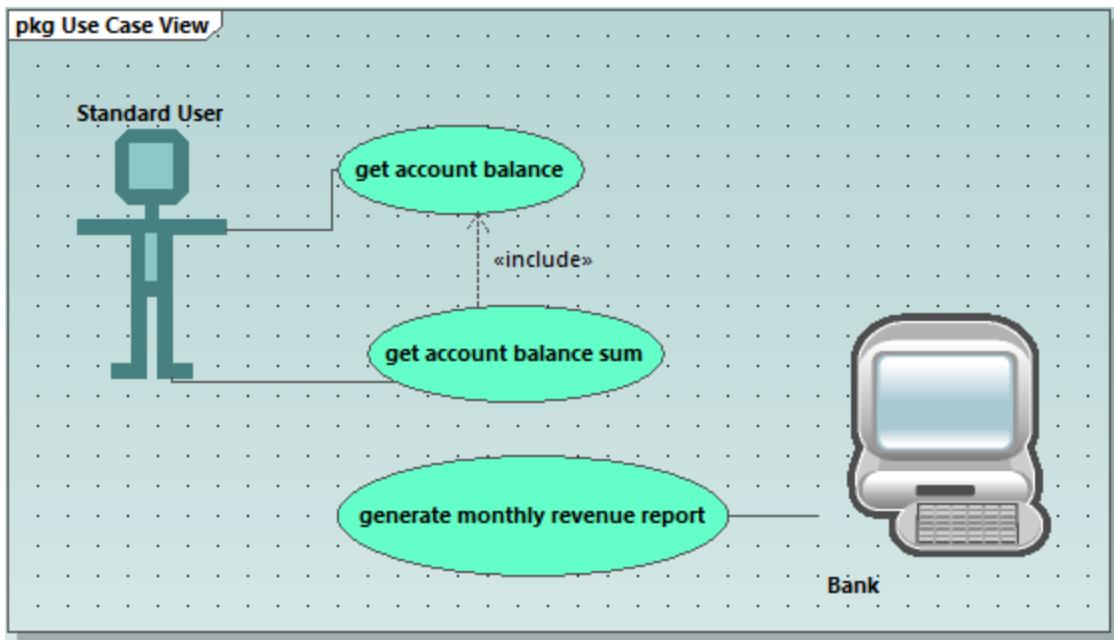
## Inserting user-defined (customized) actors

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

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



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



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

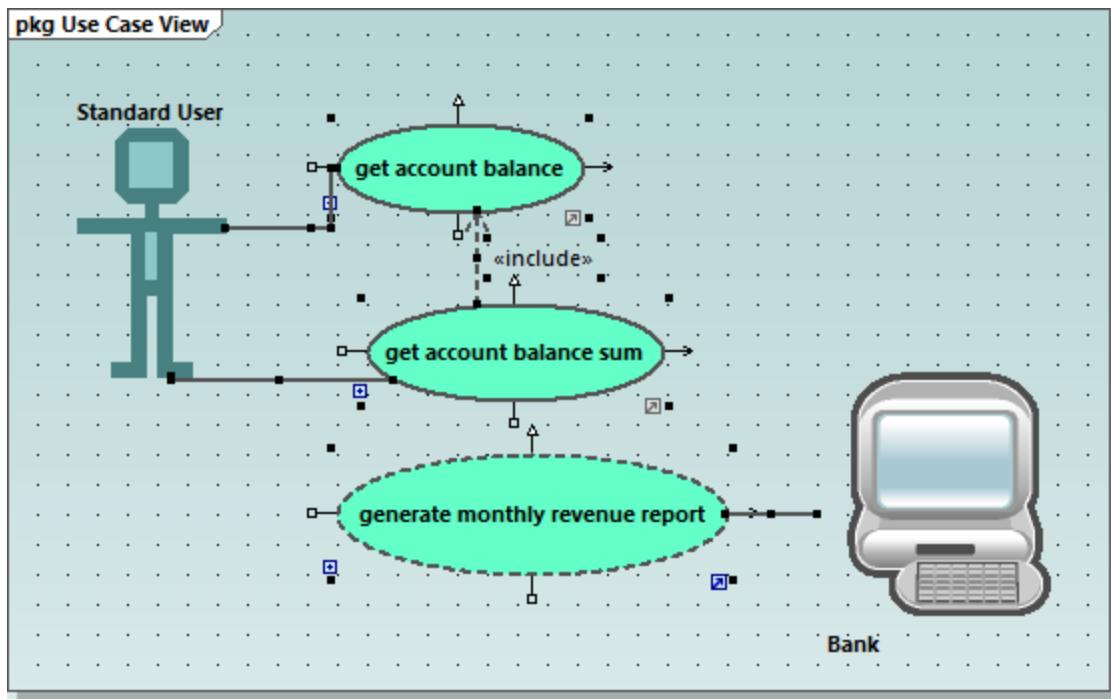
## Aligning and adjusting the size of diagram elements

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

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

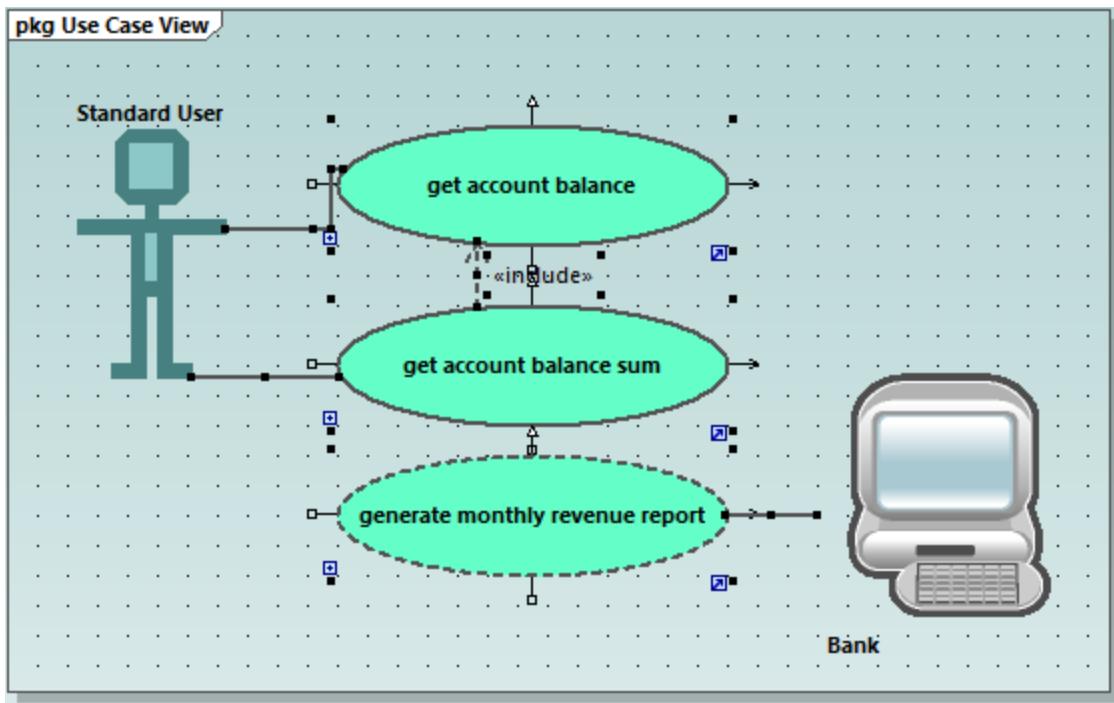
You can also align and adjust the size of multiple elements, as follows:

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



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

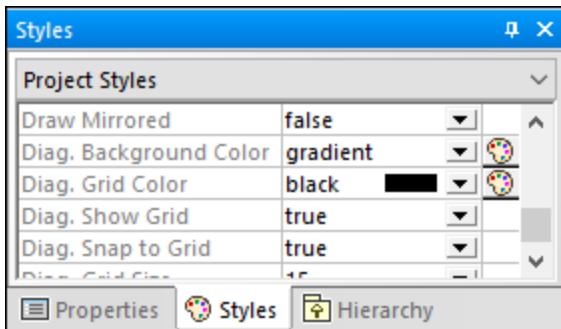
2. Click the **Make same size** toolbar button.
3. To line up all the ovals, click the **Center Horizontally** toolbar button.



## Change the style of diagrams in a project

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

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



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

### To enable or disable the diagram background grid:

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

## 2.3 Class Diagrams

This tutorial section illustrates the following tasks:

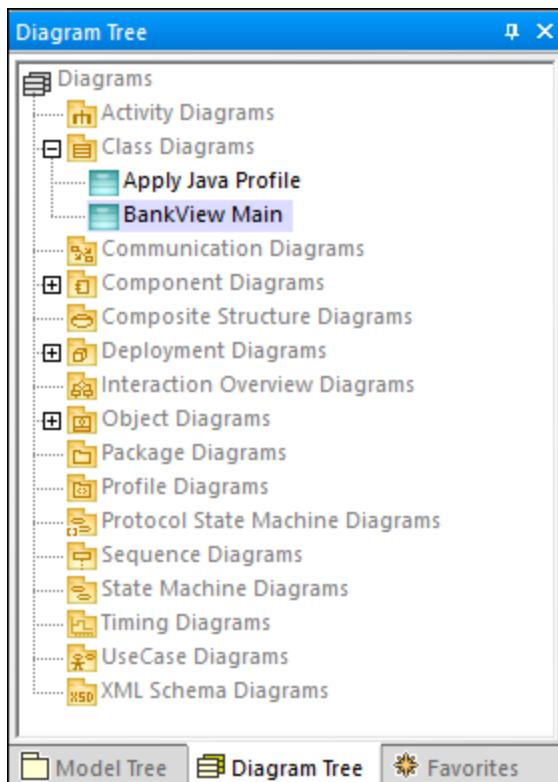
- Add an abstract class to an existing class diagram
- Add class properties and operations, and define parameters as well as their direction and type
- Add a return type to an operation
- Change icons to UML conformant symbols
- Delete and hide class properties and operations
- Create a composite association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

To proceed, run UModel and open the **BankView-start.ump** project (see also [Opening the Tutorial Project](#)<sup>15</sup>).

### Adding an abstract class

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

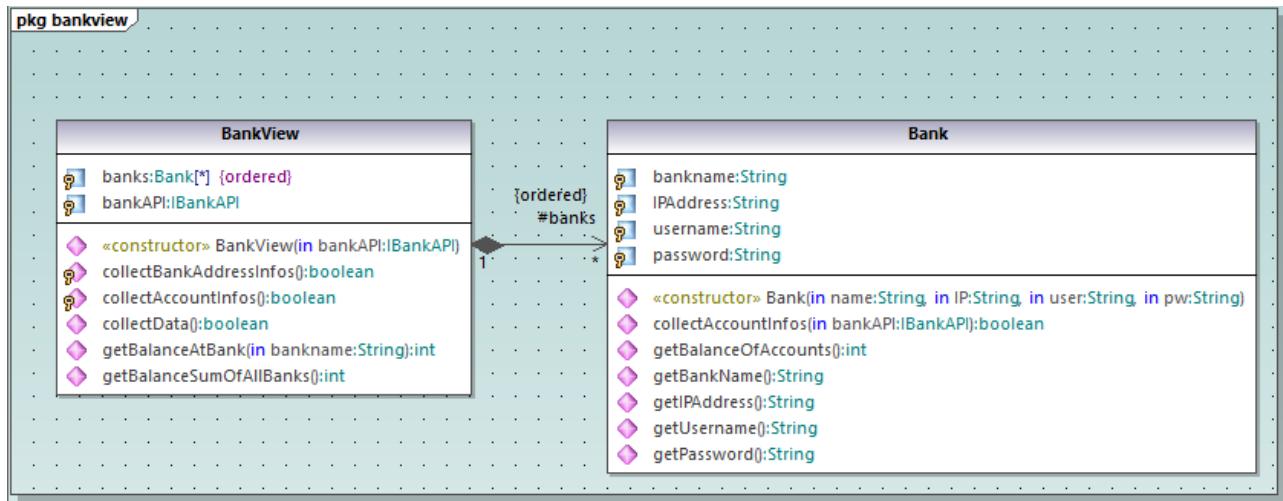
1. In the **Diagram Tree** window, expand the "Class Diagrams" package to display all class diagrams contained in the project.



2. Do one of the following:
  - Double-click the "BankView Main" diagram icon.
  - Right-click the diagram, and select **Open diagram** from the context menu.

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

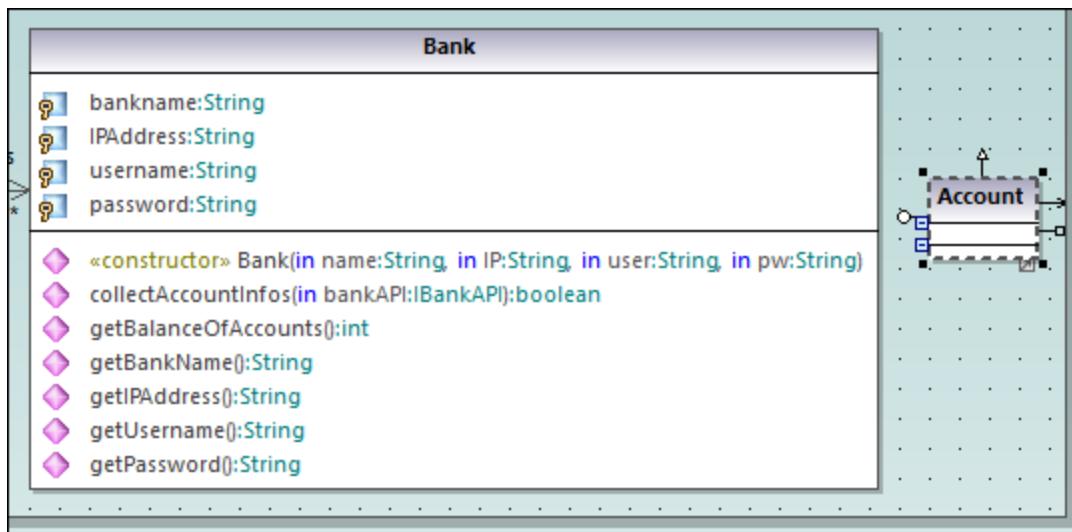
Two concrete classes with a composite association between them are visible in the class diagram.



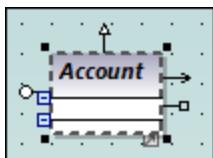
"Bank View Main" diagram

The new abstract class can be added as follows:

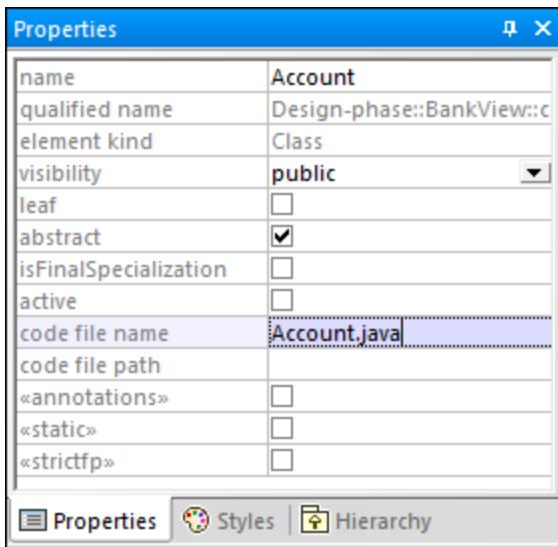
1. Click the **Class** toolbar button, and then click to the right of the **Bank** class to insert the new class.
2. Double-click the name of the new class and change it to **Account**.



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

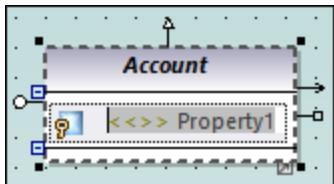


4. In the **code file name** text box, enter "Account.java" to define the Java class.

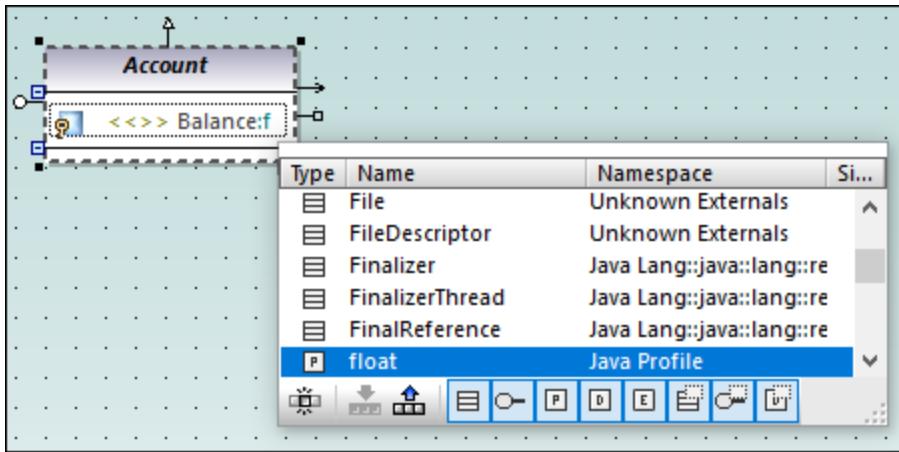


## Adding properties to a class

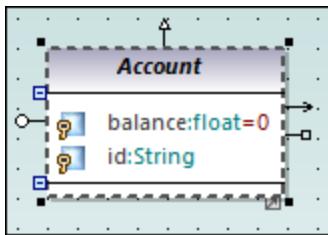
1. Right-click the "Account" class and select **New | Property**, or press **F7**. A default property `Property1` is inserted with stereotype identifiers `<>`.



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

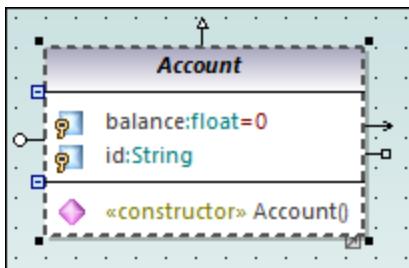


4. Continue on the same line by appending "=0" to define the default value.
5. Using the same method as above, create a new property `id` of type `String`.

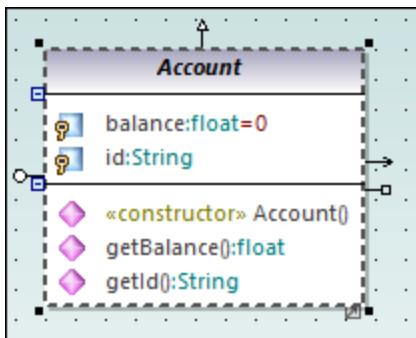


## Adding operations to a class

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

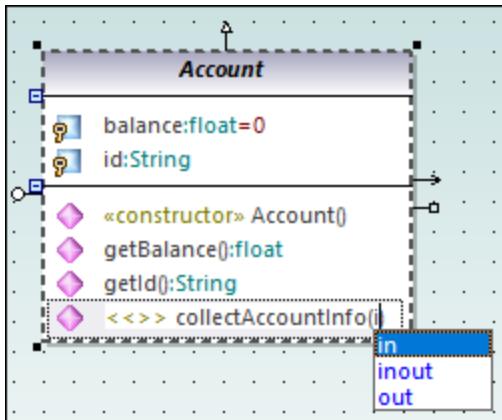


3. Using the same method as above, add two more operations, namely, `getBalance() : float` and `getId() : String`.

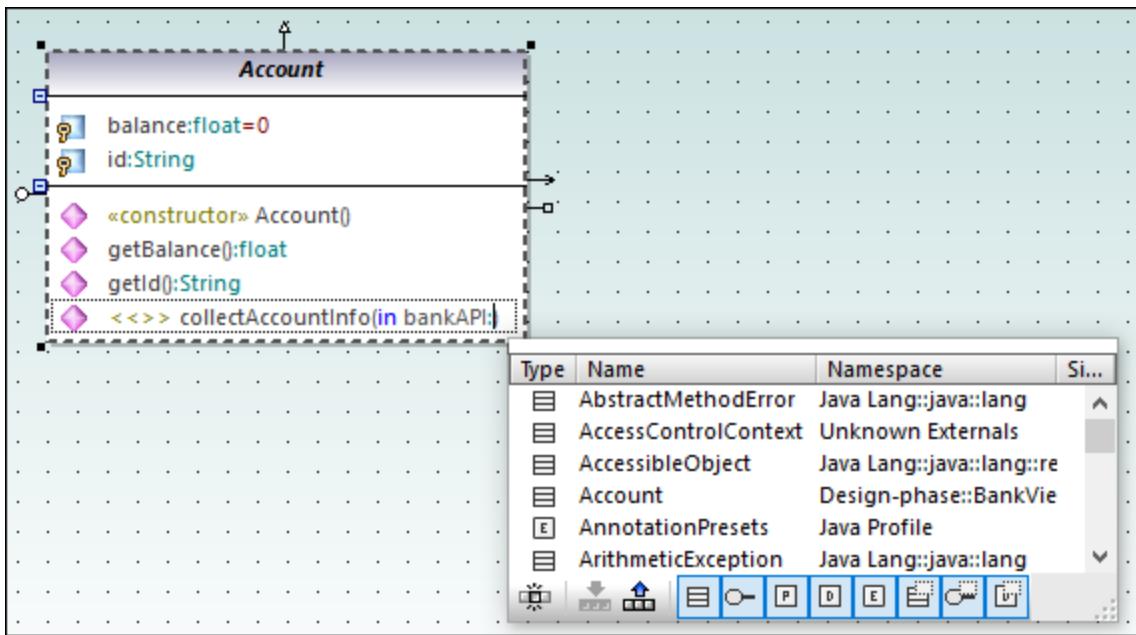


Let's now add a new operation which takes a parameter. We will also specify the parameter direction and type.

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



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

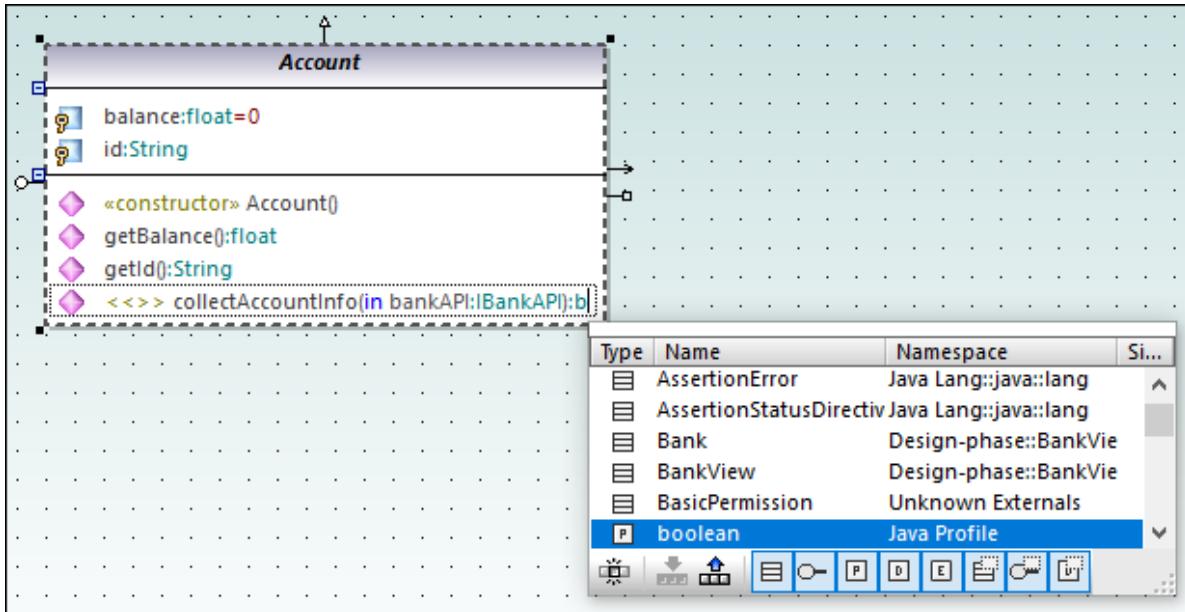


5. Select **IBankAPI** from the drop-down list.

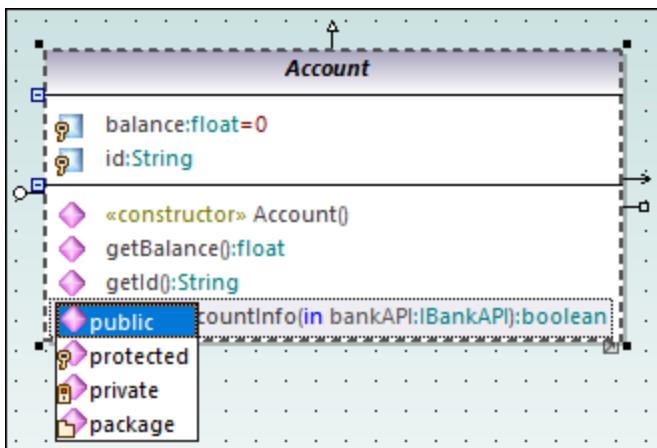
### Adding a return type to an operation

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

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



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



The visibility "package" is applicable for Java. In C#, use "package" to specify visibility as "internal". For information about how UModel elements map to constructs in each language, see [UModel Element Mappings](#)<sup>219</sup>.

## Changing icons to UML conformant symbols

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

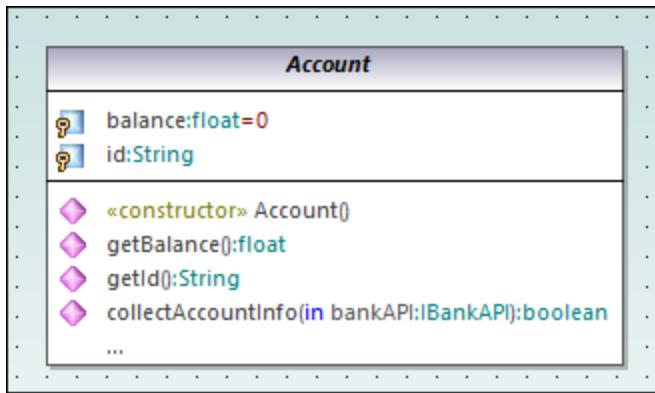
1. In the **Styles** window, select **Project Styles** from the top drop-down list.
2. Scroll down to the **Show Visibility** setting, and select **UML Style**.

## Deleting and hiding class properties and operations from a Class diagram

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

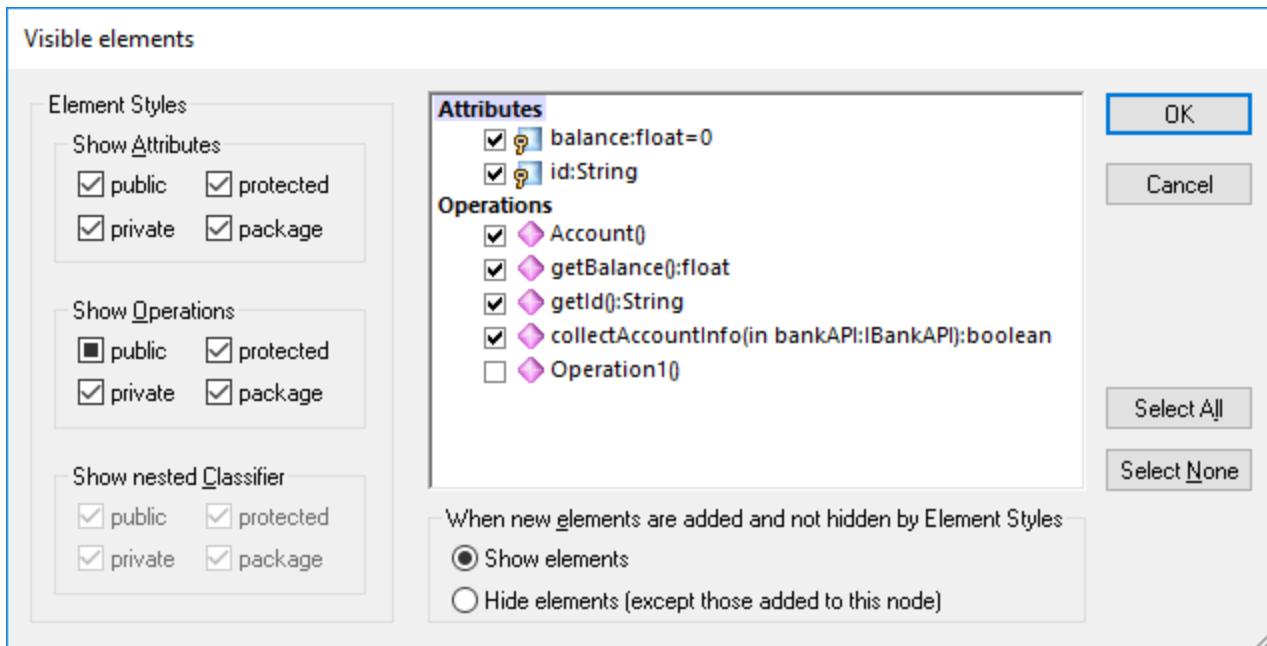
To delete the dummy operation, select it and then press **Delete**. (Alternatively, right-click it and select **Delete** from the context menu). A message box appears asking if you want to delete the element from the project. Click **Yes** to delete `Operation1` from the class diagram as well as from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

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



*A class with hidden operations*

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



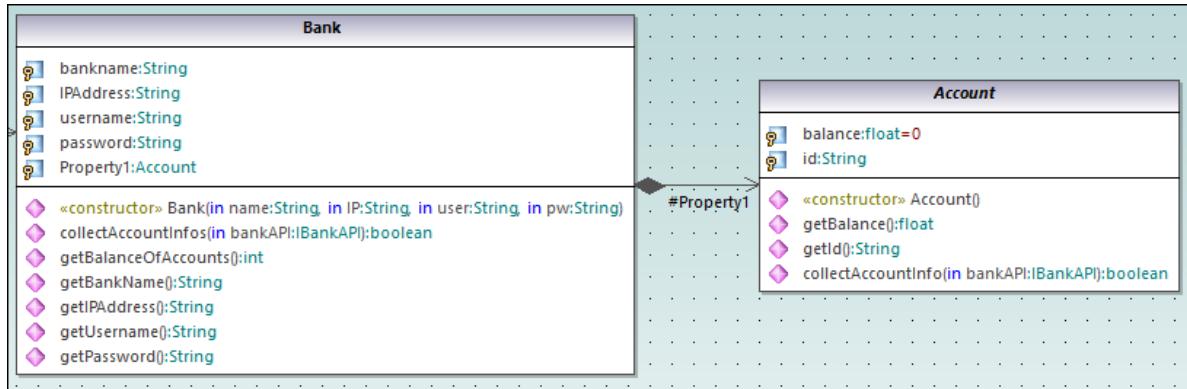
*"Visible elements" dialog box*

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

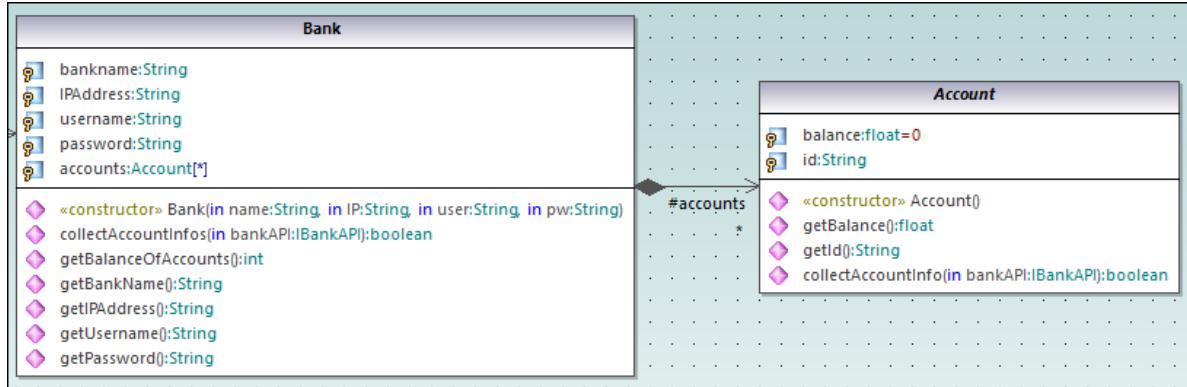
1. On the **Tools** menu, click **Options**.
2. Click the **Editing** tab.
3. Under **Ask before deleting from project**, clear the **in diagrams** check box.

## Creating a composition association between the Bank and Account classes

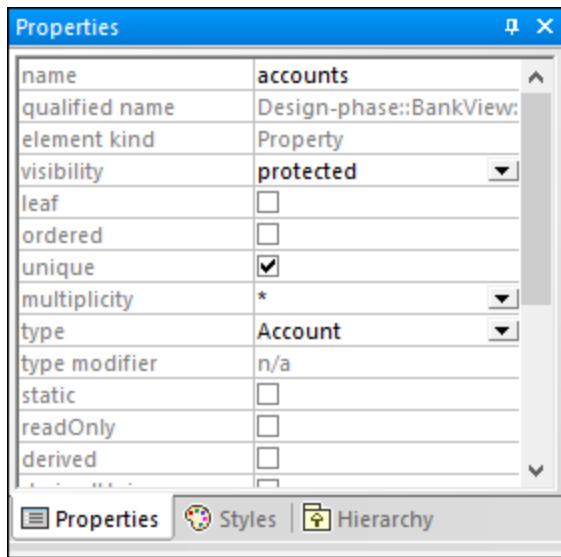
- Click the **Composition**  toolbar button, and then drag from the **Bank** class to the **Account** class. The class is highlighted when the association can be made. A new property (`Property1:Account`) is created in the **Bank** class, and a composite association arrow joins the two classes.



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



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



### 2.3.1 Creating Derived Classes

This tutorial section illustrates the following tasks:

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

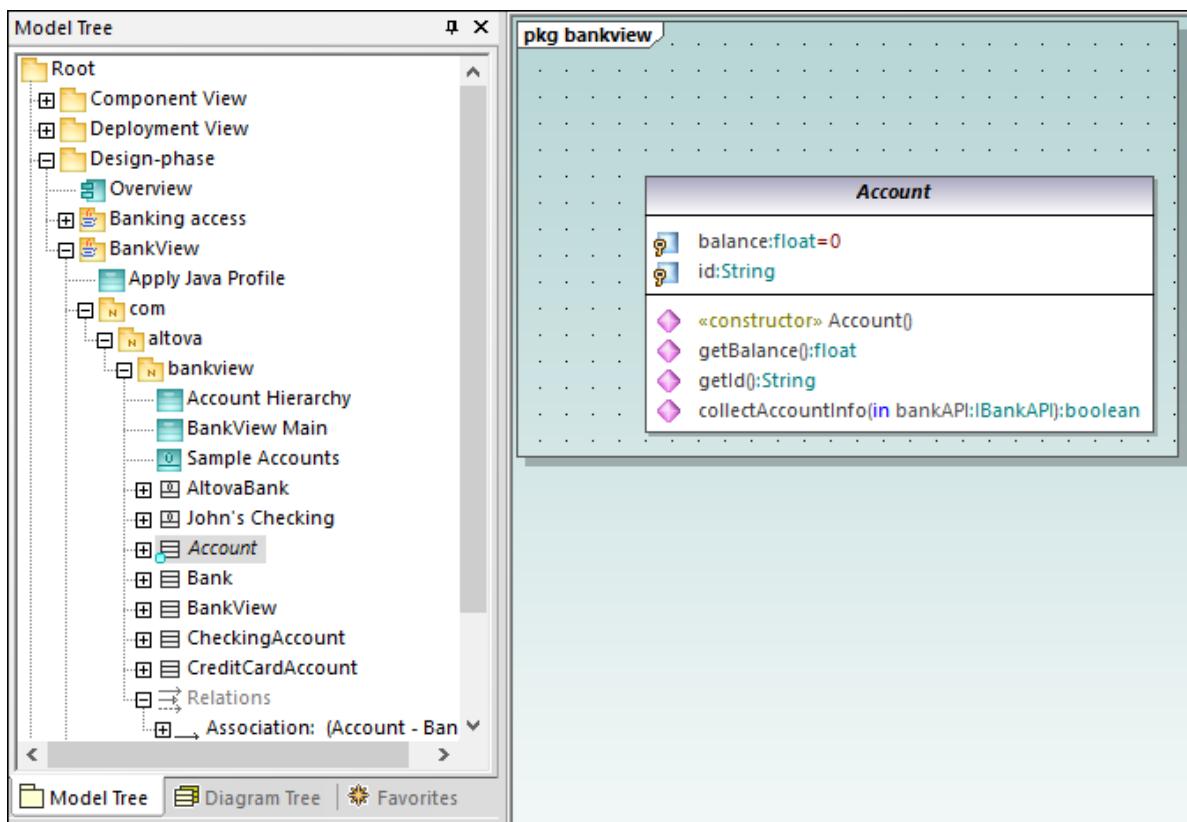
**Note:** It is assumed you have already followed the previous tutorial section, [Class Diagrams](#)<sup>27</sup>, to create the abstract class `Account`.

#### Creating a new Class Diagram

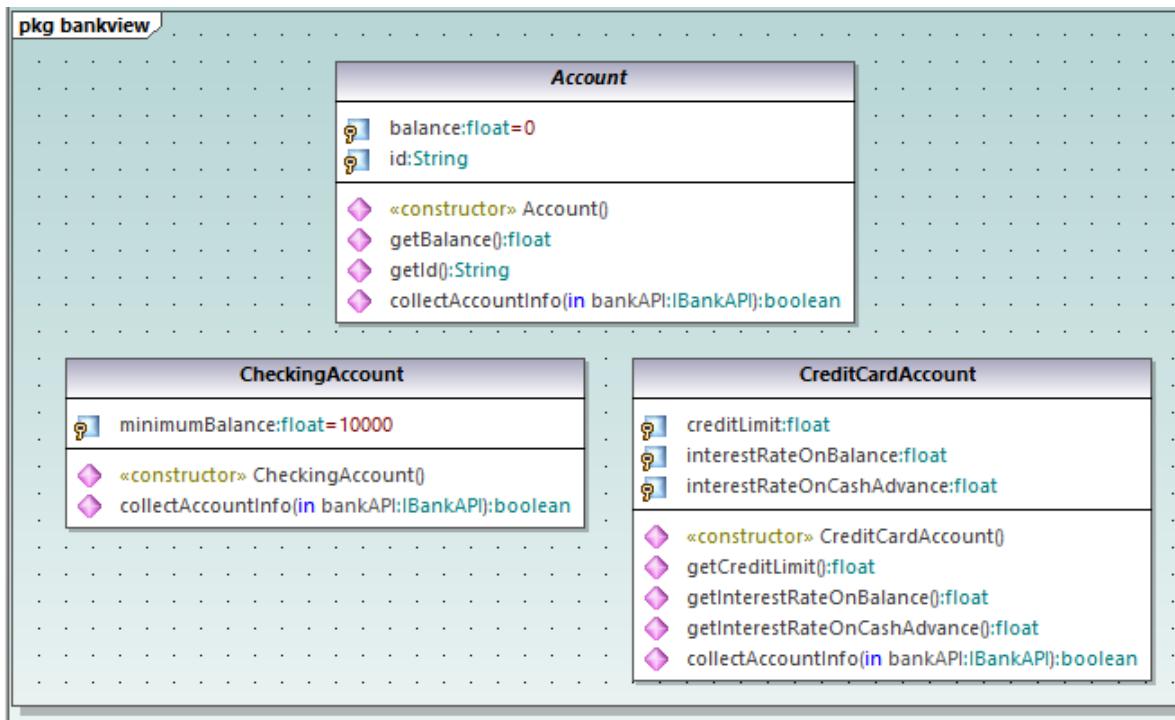
1. In the **Model Tree** window, right-click the `bankview` package (under **Root | Design-phase | BankView | com | altova**), and select **New Diagram | Class Diagram**.
2. Double-click the new "ClassDiagram1" entry, rename it to "Account Hierarchy", and press **Enter** to confirm. The new "Account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

#### Adding existing classes to a diagram

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



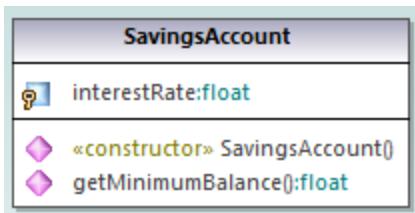
2. Click the `CheckingAccount` class (of the same package) and drag it into the diagram. Place the class below and to the left of the `Account` class.
3. Use the same method to insert the `CreditCardAccount` class. Place it to the right of the `CheckingAccount` class.



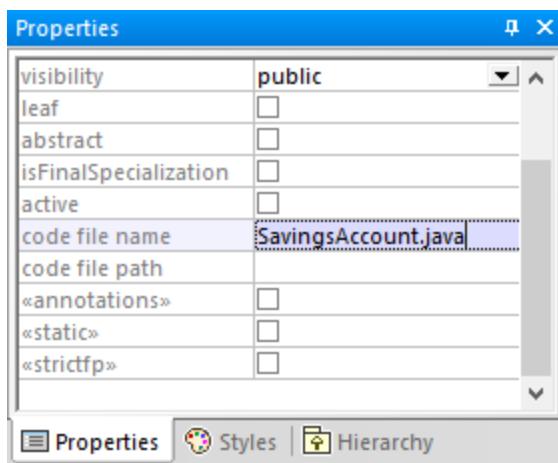
## Adding a new class

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

1. Right-click the diagram and select **New | Class**. A new class is automatically added to the correct package (`bankview`) which contains the current class diagram "Account Hierarchy".
2. Double-click the class name and change it to `SavingsAccount`.
3. Create the class structure as illustrated below. To add properties and operations, use the methods illustrated in the previous tutorial section, [Class Diagrams](#) 27.



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



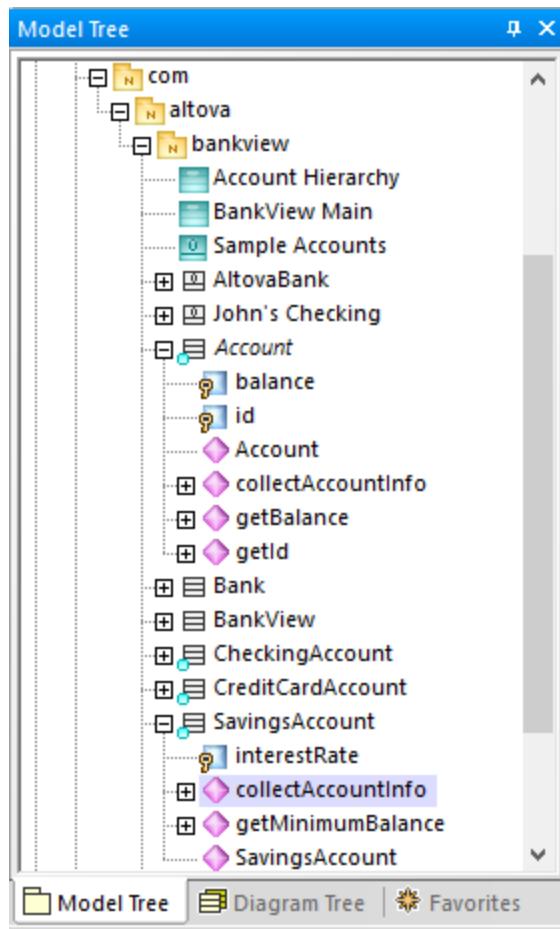
Properties and operations can be directly copied or moved from one class to another:

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

This can be achieved using drag and drop, as well as the standard **Copy/Paste** keyboard shortcuts (**Ctrl + C**, **Ctrl + V**), see also [Renaming, Moving, and Copying Elements](#)<sup>107</sup>. For the scope of this example, you can quickly copy the `collectAccountInfo()` operation from the `Account` class to the new `SavingsAccount` class, as follows:

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

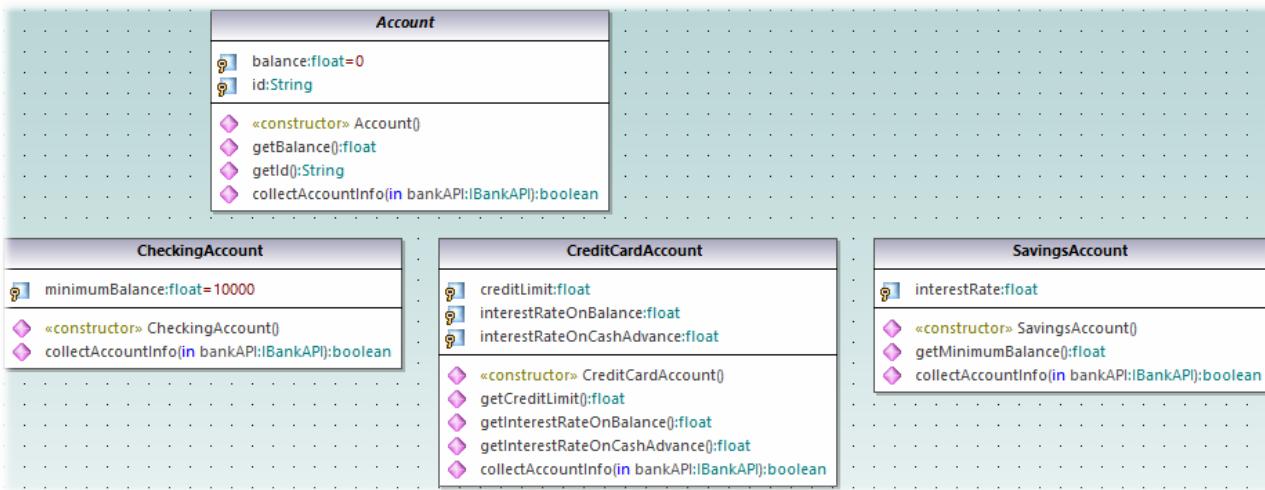
The operation is copied into the `SavingsAccount` class, which is automatically expanded to display the new operation.



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

### Creating derived classes using generalization/specialization

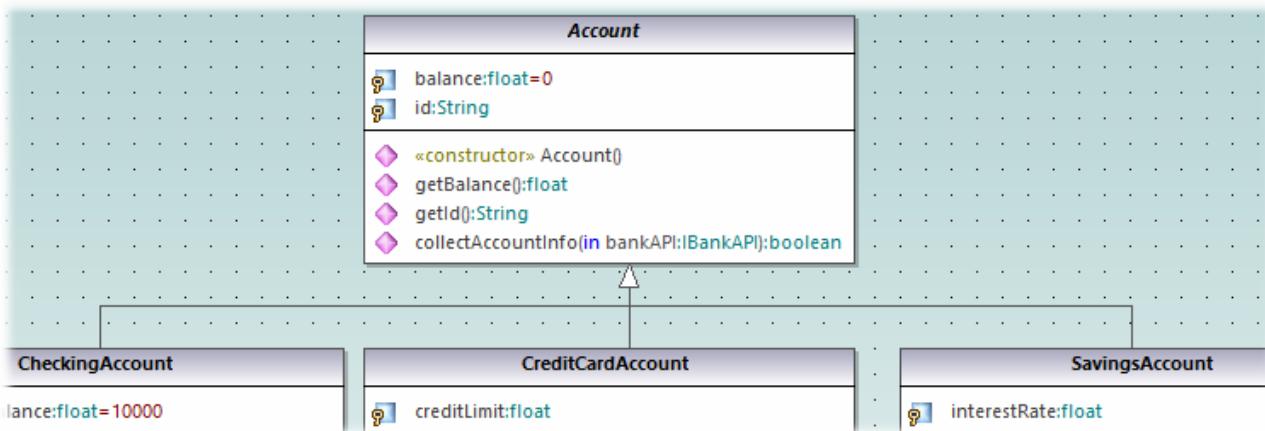
At this point, the class diagram contains the abstract class, `Account`, as well as three specific classes.



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

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

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



## 2.4 Object Diagrams

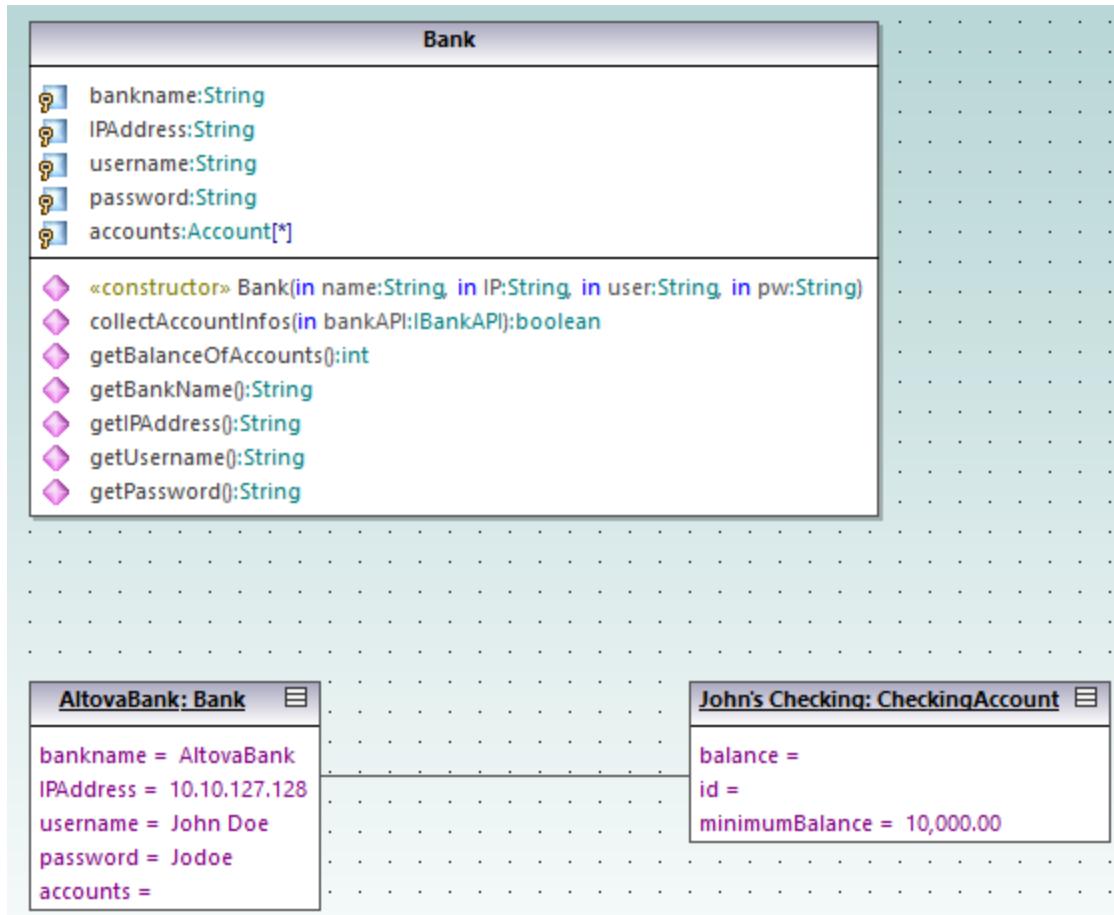
This tutorial section illustrates the following tasks:

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

To proceed, run UModel and open the **BankView-start.ump** project (see also [Opening the Tutorial Project](#)<sup>15</sup>). The project includes a predefined object diagram "Sample Accounts", which will be used to illustrate the tasks above.

### Combining objects and classes into one diagram

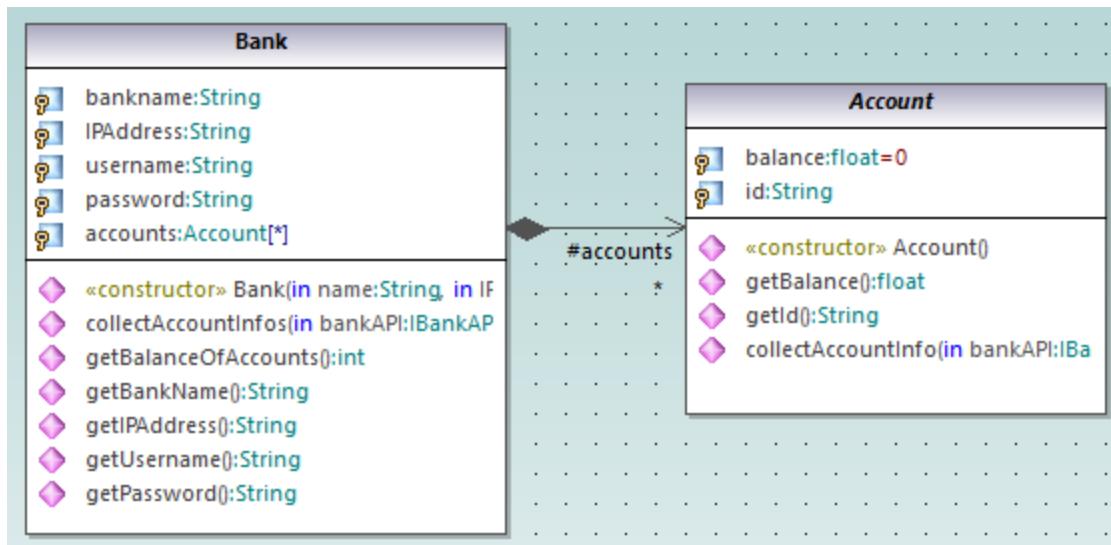
In the **Model Tree** window, navigate to the following path: **Root | Design-phase | BankView | com | altova | bankview**. Then double-click the icon next to the "Sample Accounts" diagram.



"Sample Accounts" diagram

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

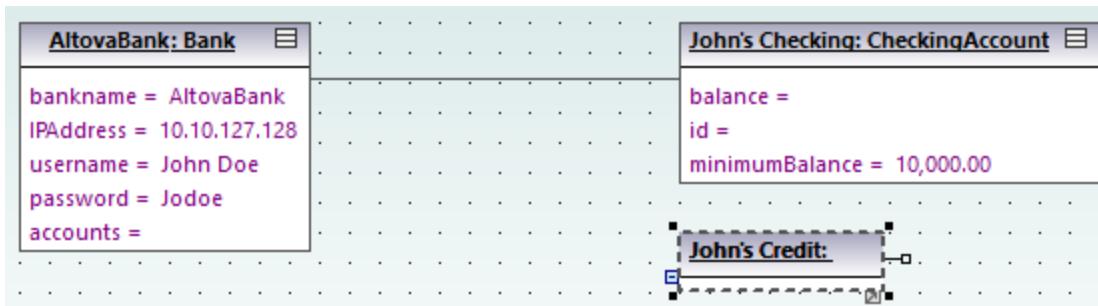
Let's now add the missing Account class to the diagram, by dragging it from the **Model Tree** into the diagram. Notice that the composite association between Bank and Account is displayed automatically (this association was defined in one of the previous tutorial sections, see [Class Diagrams](#) 27).



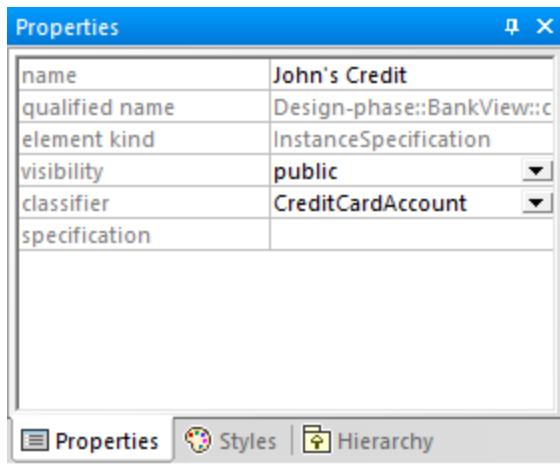
## Adding a new object-instance (Approach 1)

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

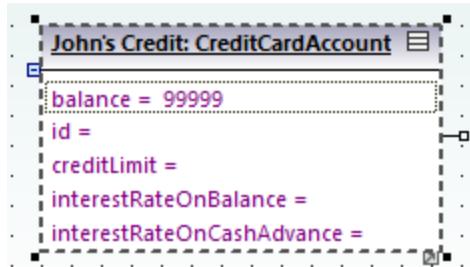
1. Click the **InstanceSpecification** toolbar button, and then click inside the diagram, below the object John's Checking: Checking Account.
2. Change the name of the new instance to John's Credit, and press **Enter**.



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



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

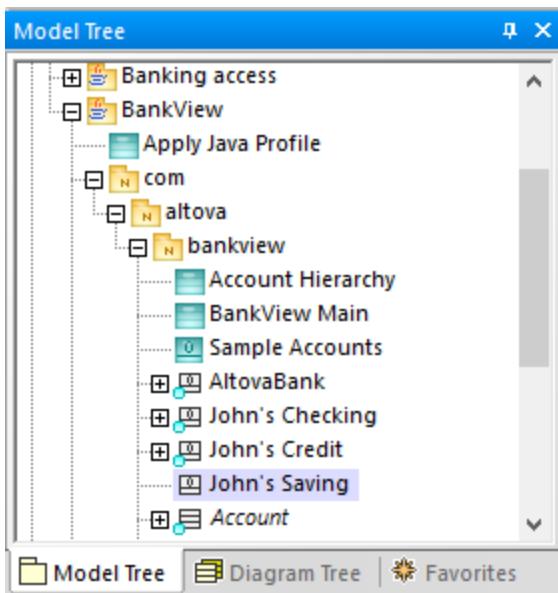


To show or hide specific nodes, right-click the instance and select **Show/hide node content (Ctrl+Shift+H)** from the context menu.

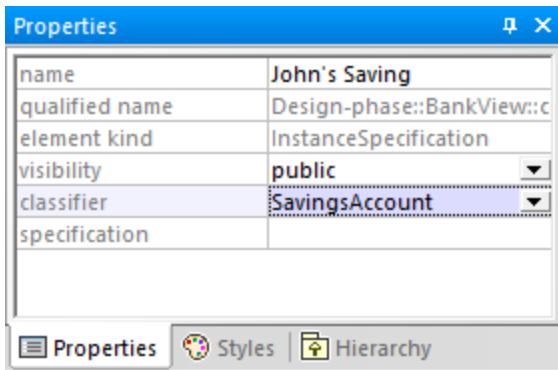
## Adding a new object-instance (Approach 2)

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

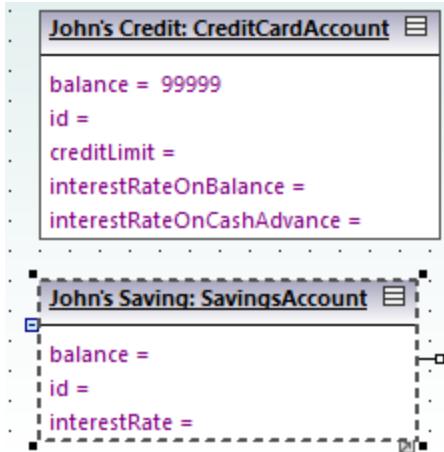
1. In the **Model Tree** window, right-click the `bankview` package, and select **New element | InstanceSpecification**.
2. Rename the new instance to `John's Saving`, and press **Enter** to confirm. The new object is added to the package and sorted accordingly.



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



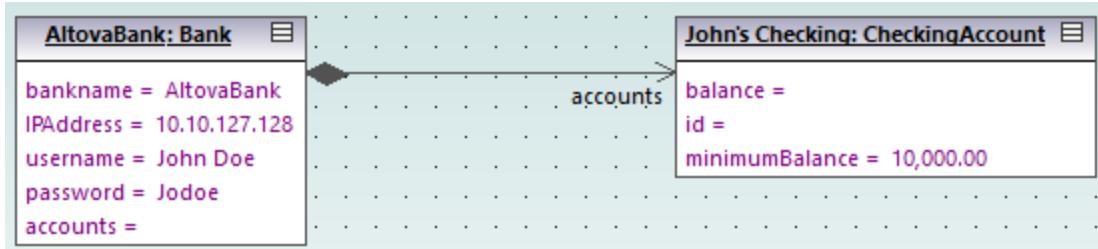
4. Drag the object John's Saving from the **Model Tree** window into the diagram, placing it below the object John's Credit.



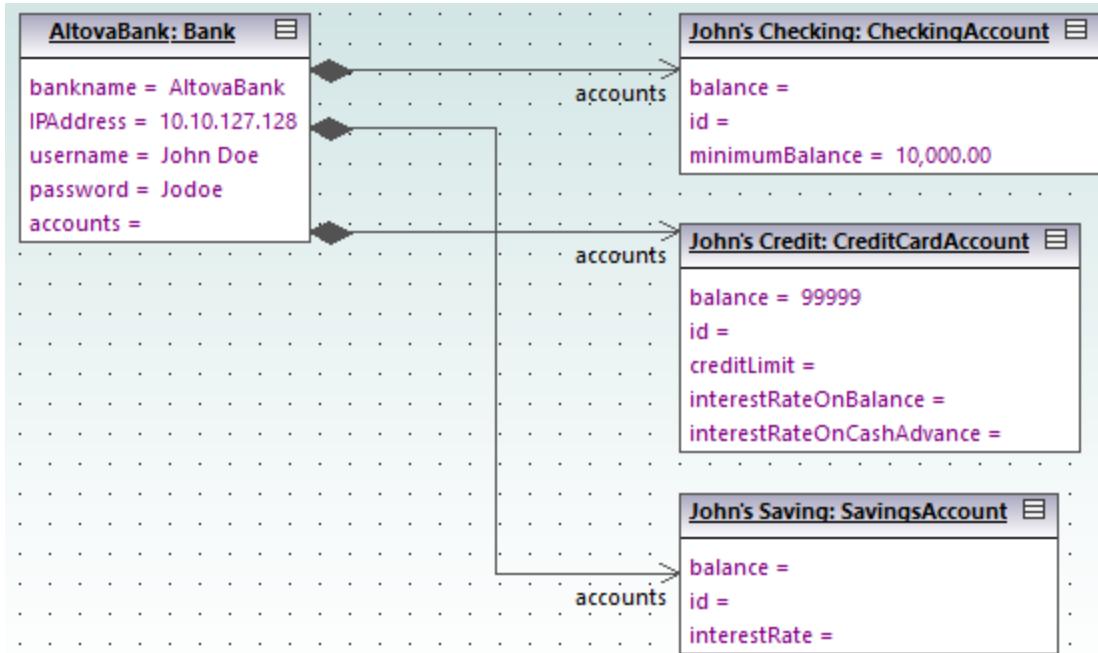
## Creating links between objects

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

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



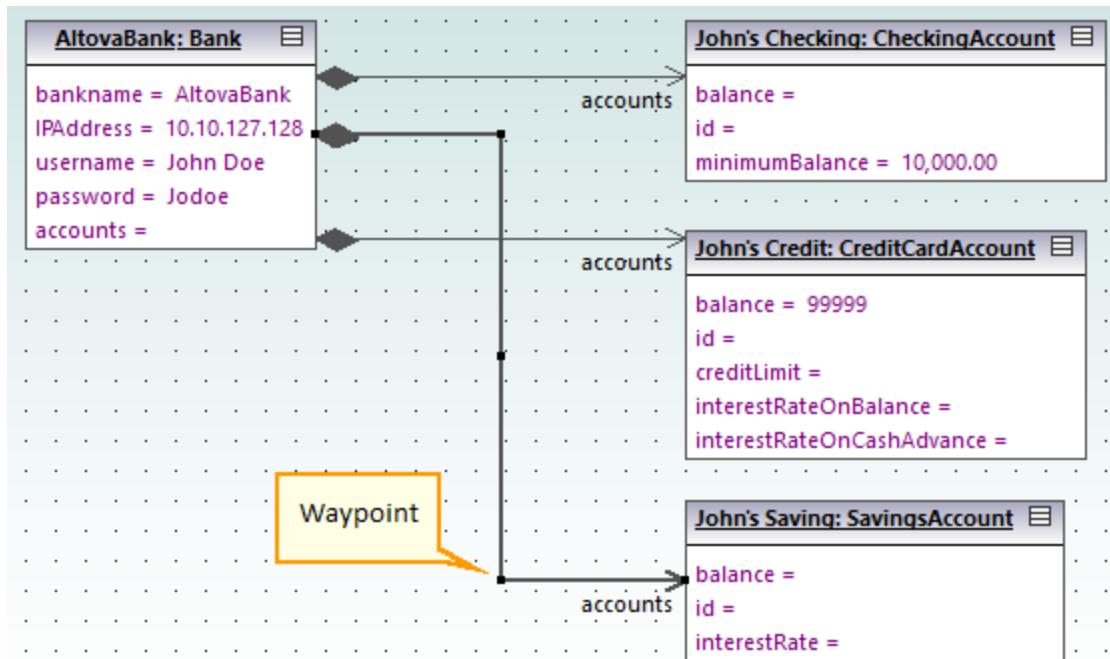
3. Click the **InstanceSpecification** toolbar button, and position the cursor over the object John's Credit: CreditAccount. The cursor now appears as a + sign.
4. Drag from the object John's Credit: CreditAccount to AltovaBank: Bank to create a link between the two.
5. In the **Properties** window,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



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

## Formatting association/link lines in a diagram

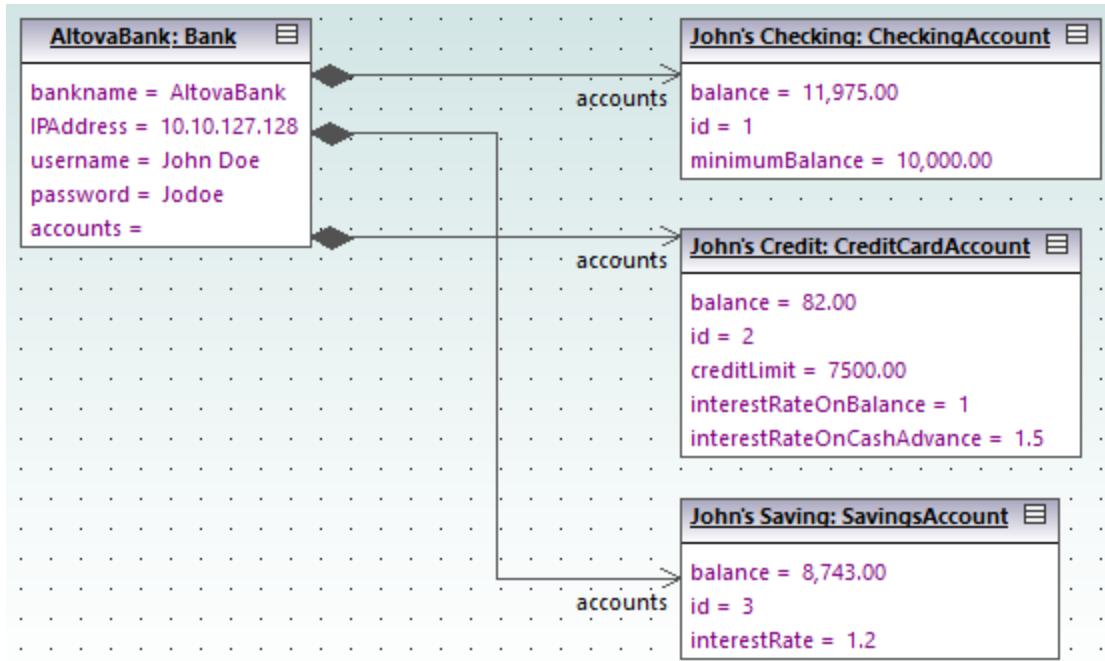
To format links between objects, place the cursor on the line and drag to the desired position. To reposition the line both horizontally and vertically, drag the corner waypoint, as illustrated below.



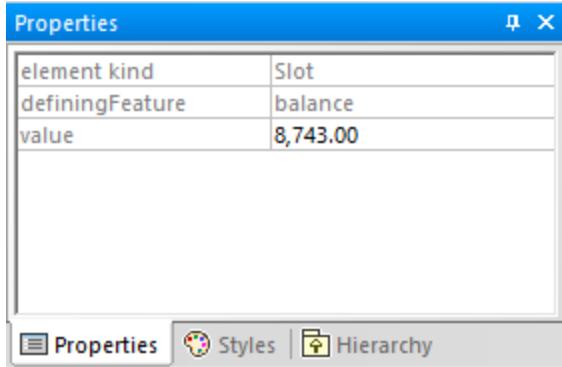
Links in an object diagram

## Entering sample data into objects

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



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



## 2.5 Component Diagrams

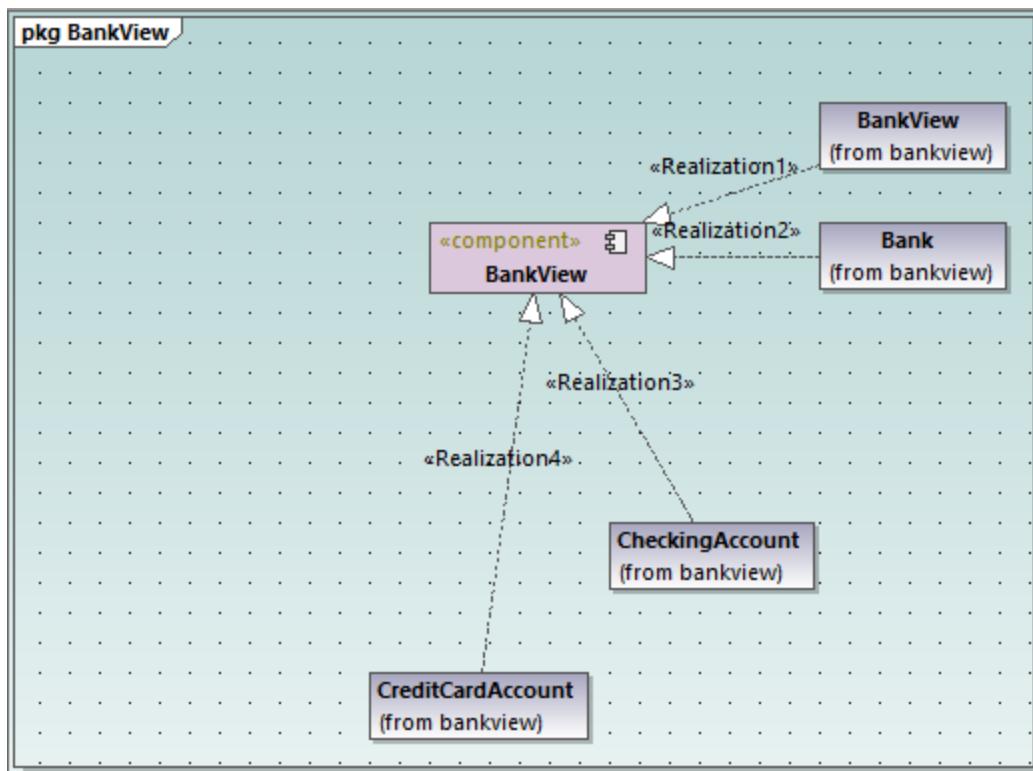
This tutorial section illustrates the following tasks:

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

To proceed, run UModel and open the **BankView-start.ump** project (see also [Opening the Tutorial Project](#)<sup>15</sup>). The project includes several predefined object diagrams which will be used to illustrate the tasks above. It is assumed you have already followed the tutorial section [Creating Derived Classes](#)<sup>36</sup> to create the class `SavingsAccount`.

### Creating realization dependencies between classes and components

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

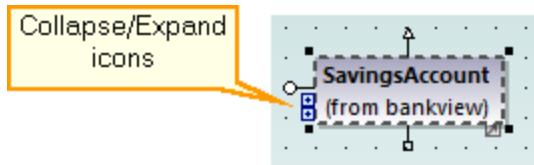


"Bank View realization" diagram

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

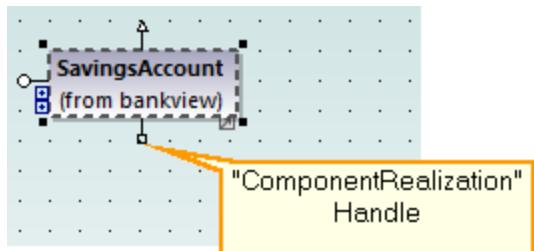
1. In the **Model Tree** window, locate the `SavingsAccount` class in the `bankview` package. If this class is missing, follow the tutorial section [Creating Derived Classes](#)<sup>36</sup> to create it first.
2. Drag the `SavingsAccount` class from the **Model Tree** into the diagram.

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

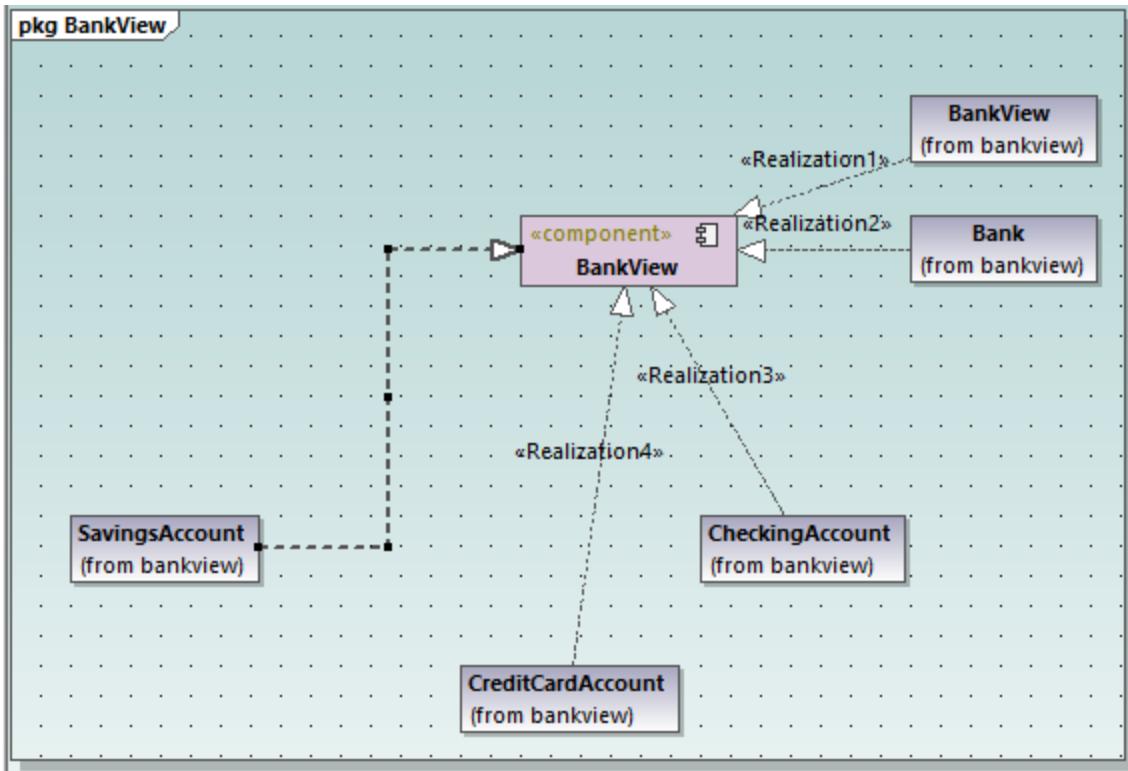


To create a realization dependency between the class and the component, do one of the following:

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



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



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

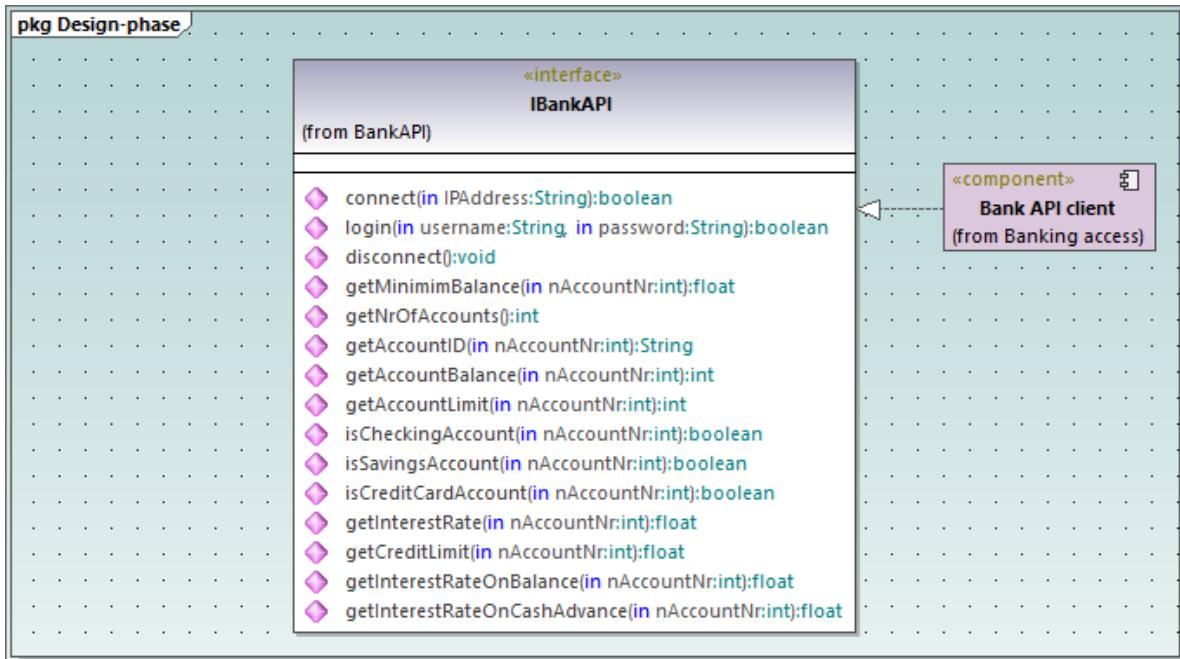
### Changing the appearance of diagram lines

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

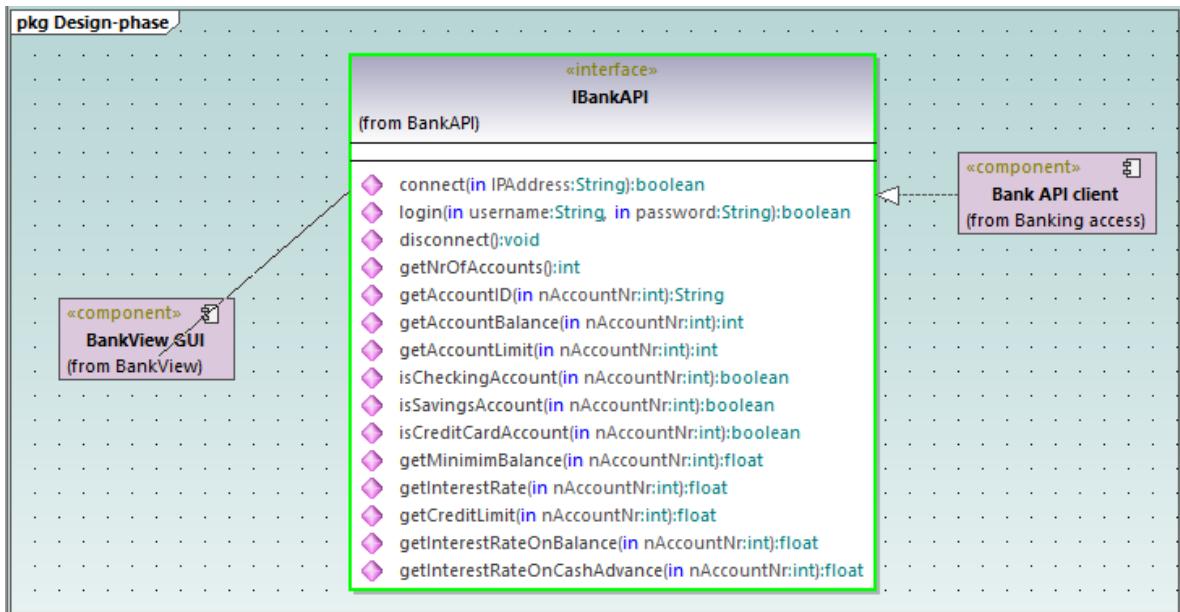
1. Select the line created previously (that is, the one between SavingsAccount and BankView).
2. Click the **Direct Line**  toolbar button.

### Adding usage dependencies to an interface

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

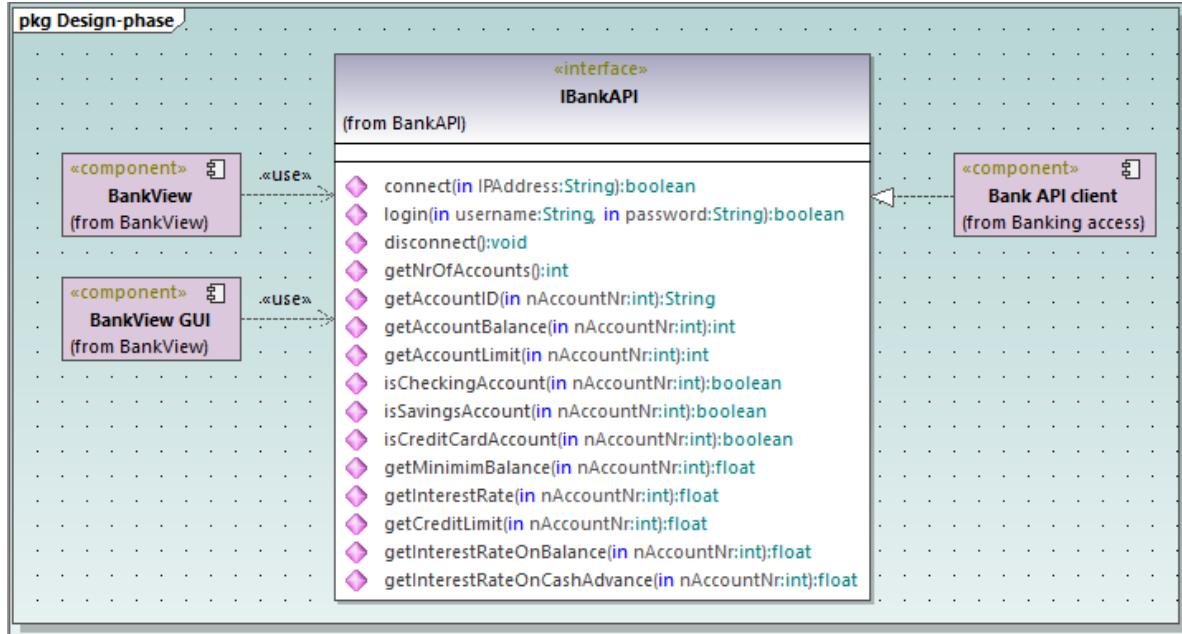


- In the **Model Tree** window, navigate to **Root | Component View | BankView** and drag the `BankView` GUI package into the diagram.
- Also drag the `BankView` package into the diagram.
- Click the **Usage** toolbar button and drag from the `BankView` GUI package to the `IBankAPI` Interface.



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

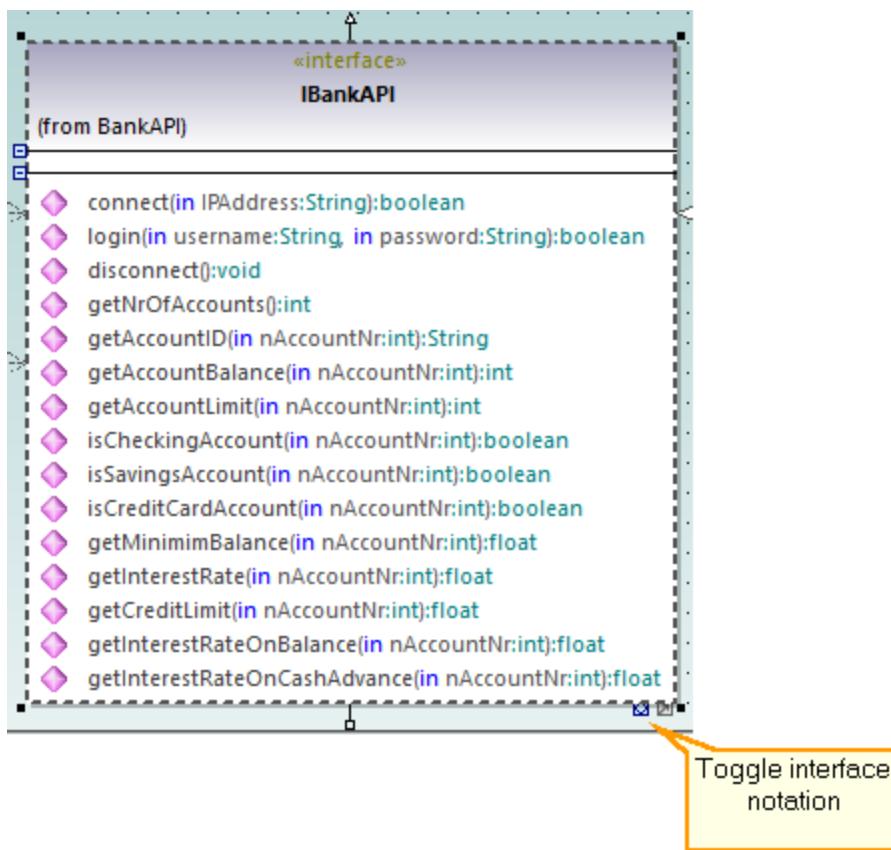
As illustrated below, both packages now have a usage dependency to the interface. Namely, the `IBankAPI` interface is required by the packages `BankView` and `BankView GUI`. As for the package `Bank API Client`, it provides the interface.



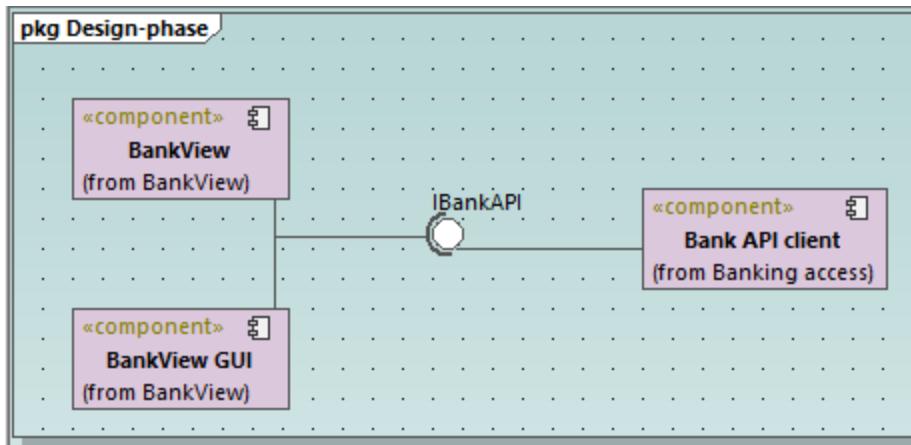
## Using "ball-and-socket" notation

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

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



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



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

## 2.6 Deployment Diagrams

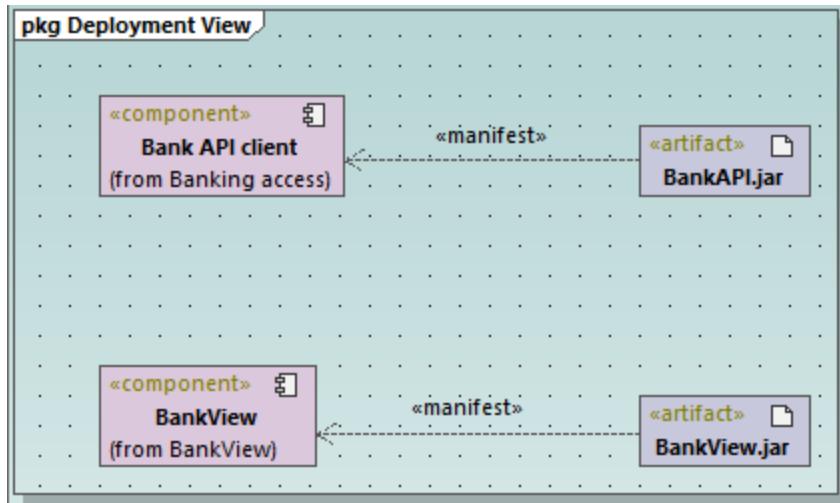
This tutorial section illustrates the following tasks:

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

To proceed, run UModel and open the **BankView-start.ump** project (see also [Opening the Tutorial Project](#) 15).

### Adding a dependency between two artifacts in a Deployment diagram

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



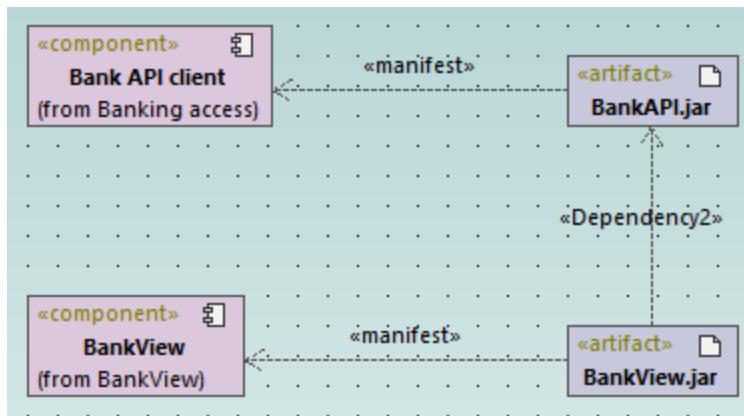
"Artifacts" diagram

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

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

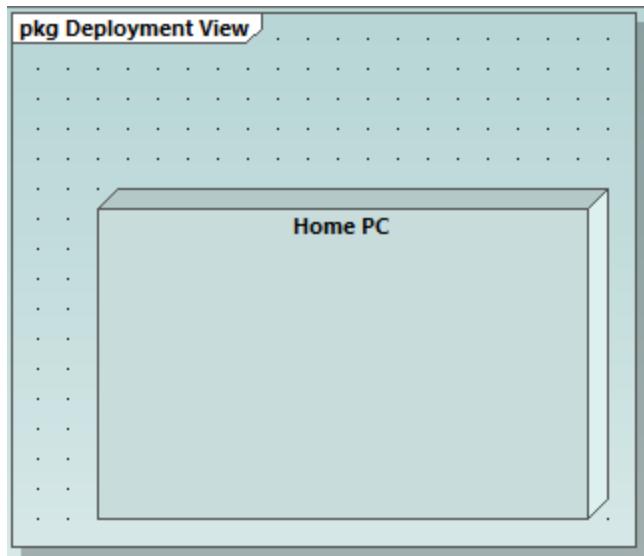
Using the same technique, let's also add a dependency between the two .jar files, as follows:

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



### Adding elements to a Deployment diagram

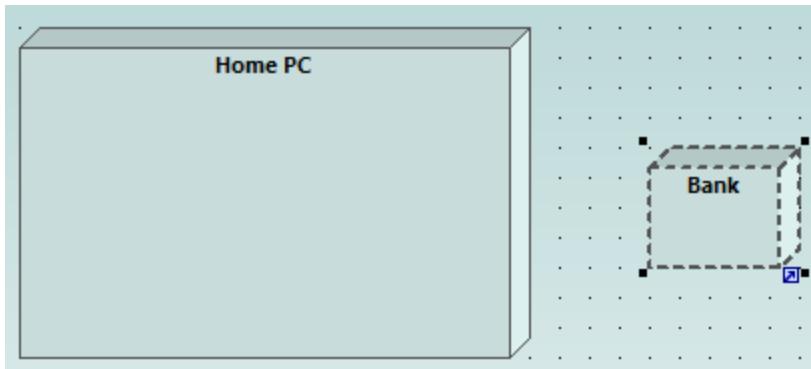
In the **Diagram Tree** window, under "Deployment Diagrams", double-click the icon next to the "Deployment" diagram to open it. This diagram is deliberately incomplete and consists of a single node, which represents a home PC. In the following steps, we will be adding more elements to this diagram.



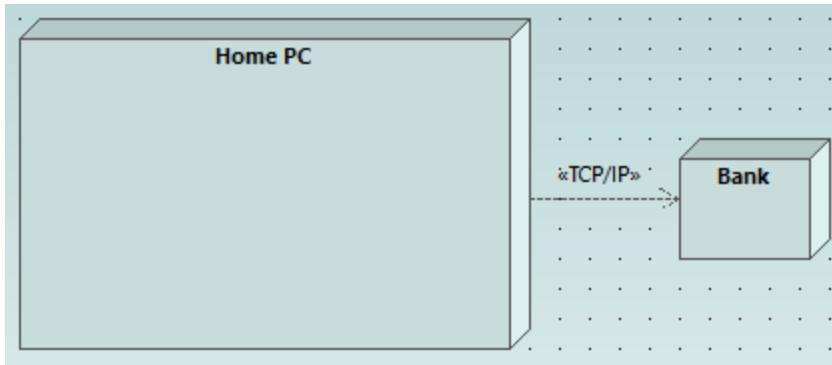
*"Deployment" diagram*

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

1. Click the **Node**
2. toolbar button, and click right of the Home PC node to insert it.

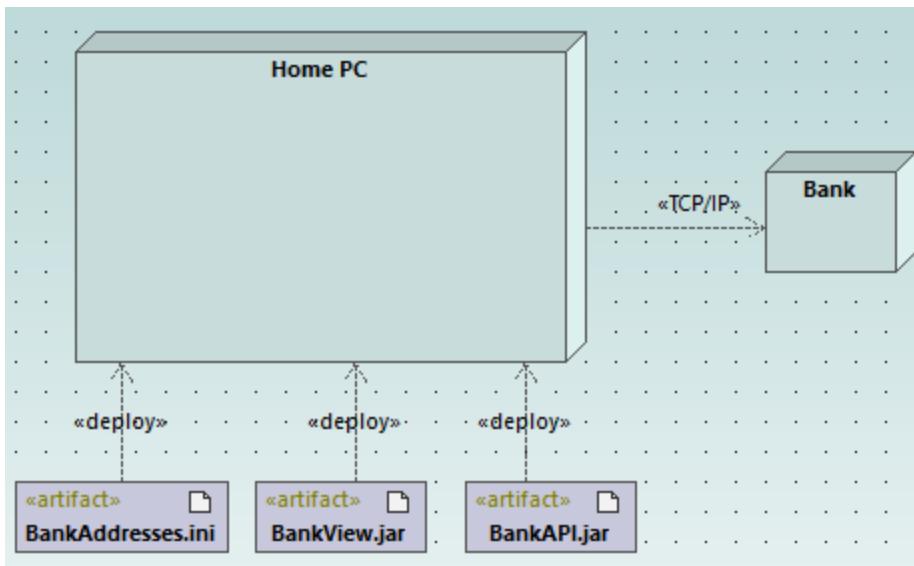


3. Click the **Dependency** toolbar button, and then drag from the "Home PC" node to the "Bank" node. This creates a dependency between the two nodes.
4. Select the dependency line and enter "TCP/IP" as name of the new dependency. (Alternatively, edit the **Name** property in the **Properties** window).

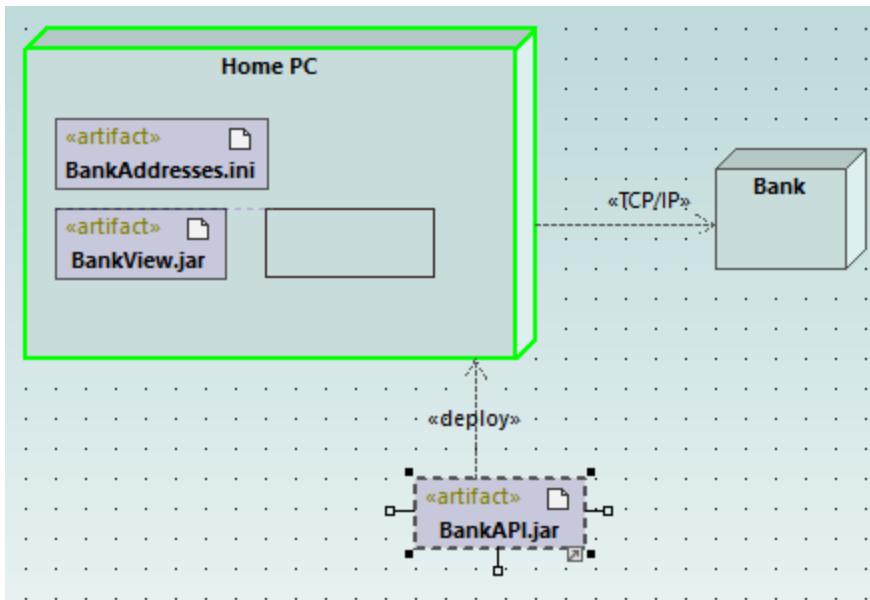


## Embedding artifacts

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

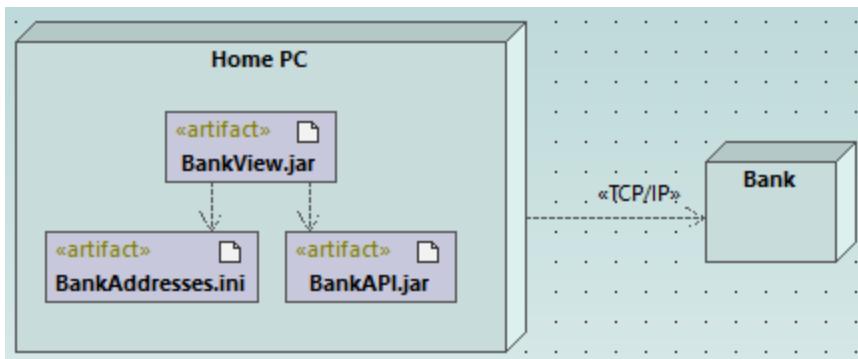


You can also embed the artifacts into the "Home PC" node, by dragging each of the artifacts into it. Notice that the deploy dependencies are no longer visible on the diagram, although they continue to exist logically.



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

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



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

### Creating artifact elements (properties, operations, nested artifacts)

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

## 2.7 Forward Engineering (from Model to Code)

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

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

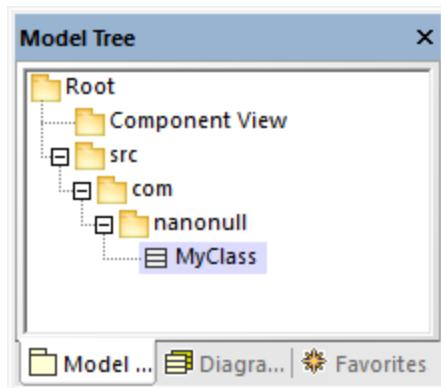
### Creating a new UModel project

You can create a new UModel project as follows:

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

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

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



### Preparing the project for code generation

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

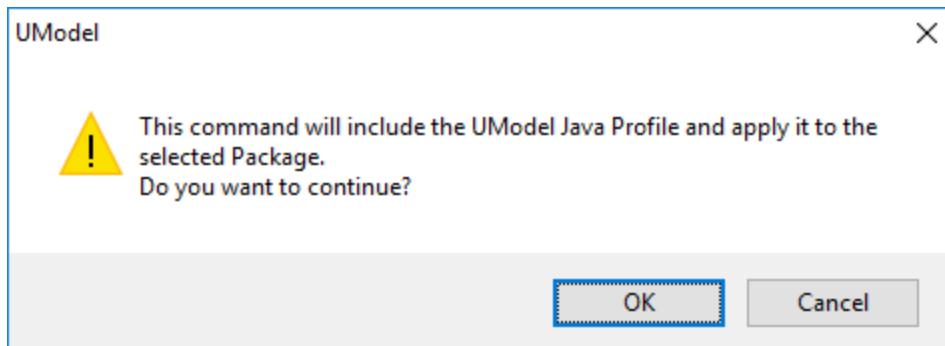
- A Java, C#, or VB.NET namespace root package must be defined.
- A component must exist which is realized by all classes or interfaces for which code must be generated.
- The component must have a physical location (directory) assigned to it. Code will be generated in this directory.
- The component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

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

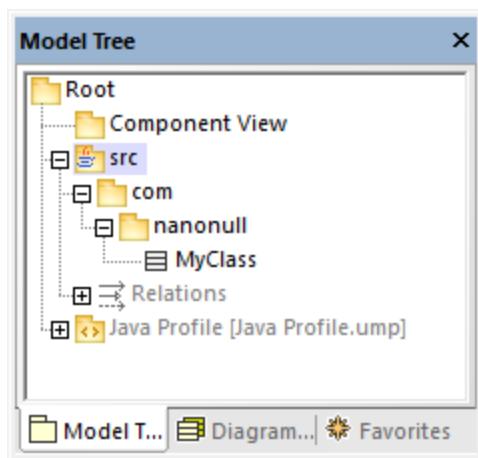
- On the **Project** menu, click **Check Project Syntax**. (Alternatively, press **F11**.)

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

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



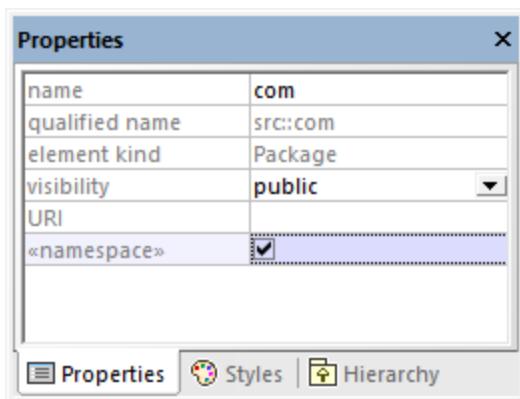
Notice the package icon has now changed to , which signifies that this package is a Java namespace root. Additionally, a Java Profile has been added to the project.



The actual namespace can be defined as follows:

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

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

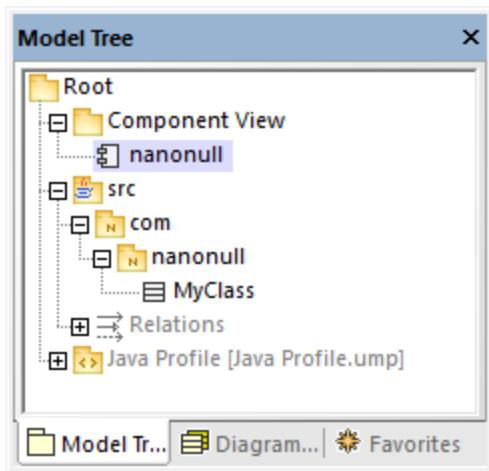


3. Repeat the step above for the "nanonull" package.

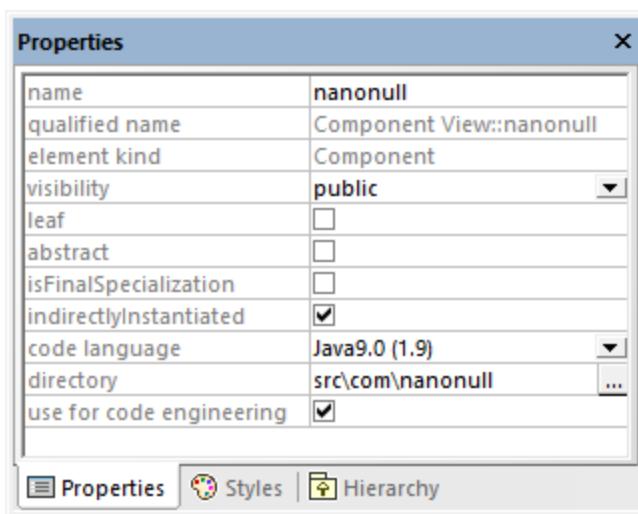
Notice that the icon of both "com" and "nanonull" packages has now changed to , which indicates these are now namespaces.

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

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



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



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

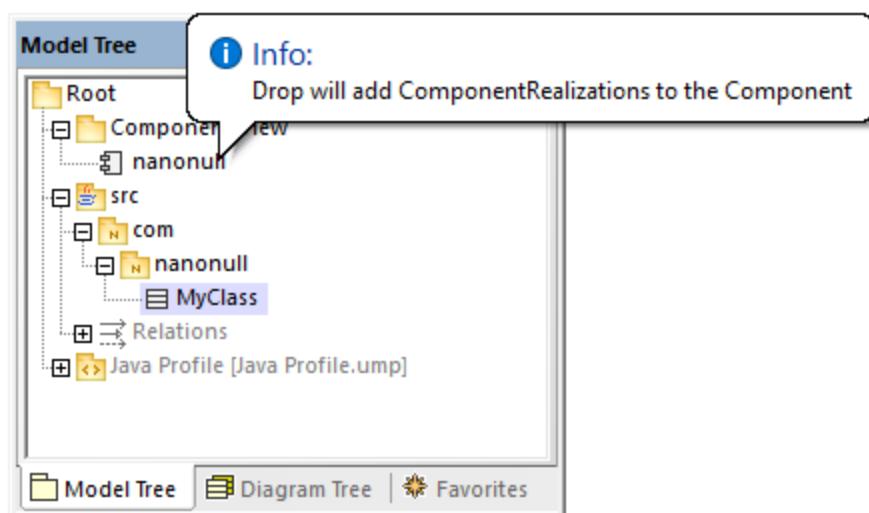
**Note:** The code generation path can be absolute or relative to the .ump project. If it is relative as in this example, a path such as **src\com\nanonull** would create all the directories in the same directory where the UModel project was saved.

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

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

The component realization relationship can be created as follows:

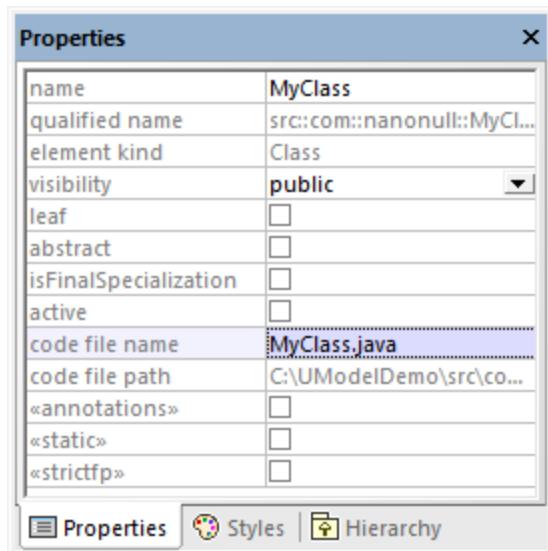
- In the **Model Tree** window, drag from the `MyClass` created previously and drop onto component nanonull.



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

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

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

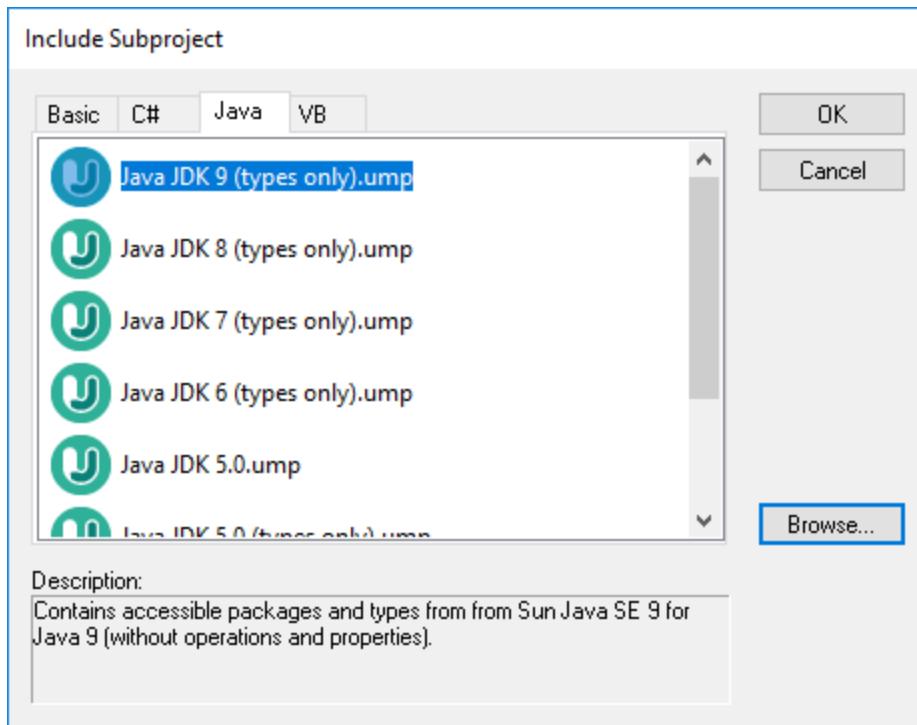


## Including the JDK types

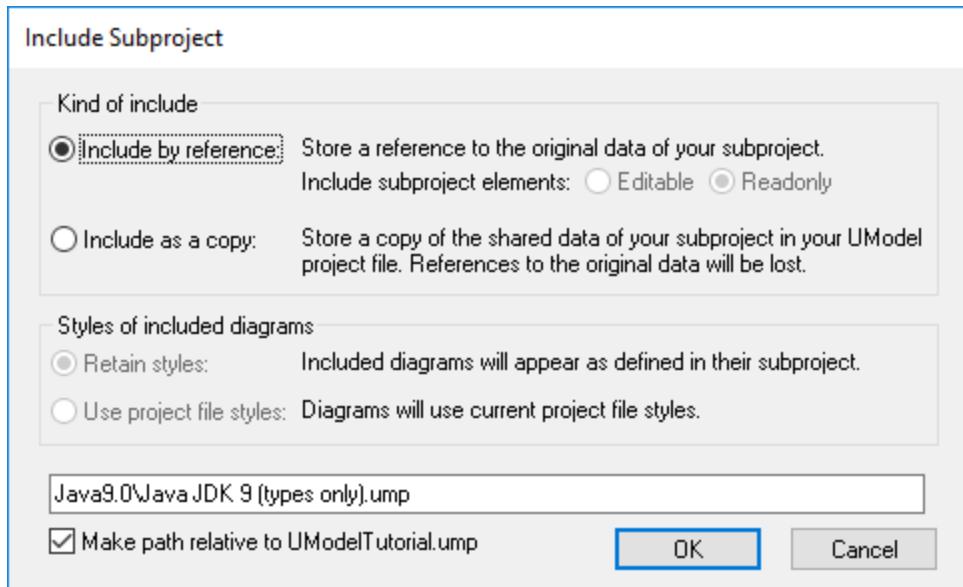
Although this step is optional, it is recommended that you include the Java Development Kit (JDK) language types, as a subproject of your current UModel project. Otherwise, the JDK types will not be available when you

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

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



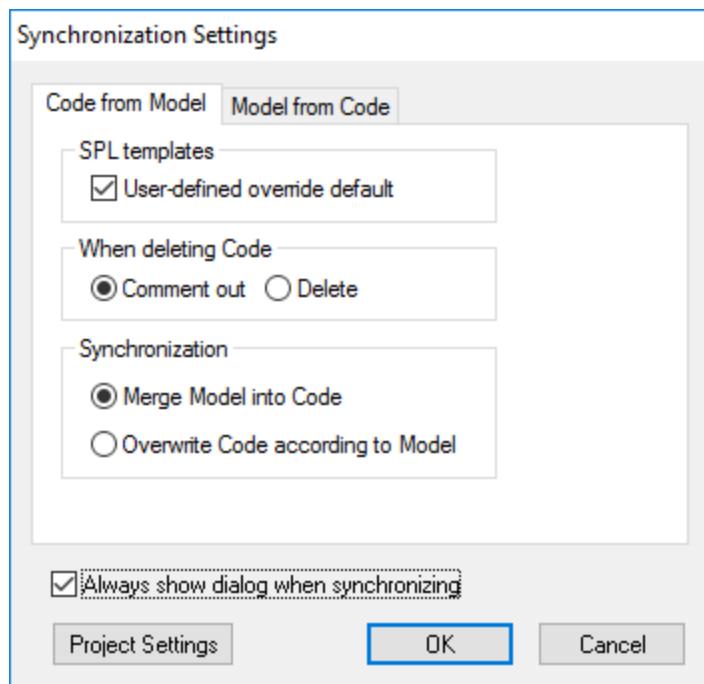
3. When prompted to include by reference or as a copy, select **Include by reference**.



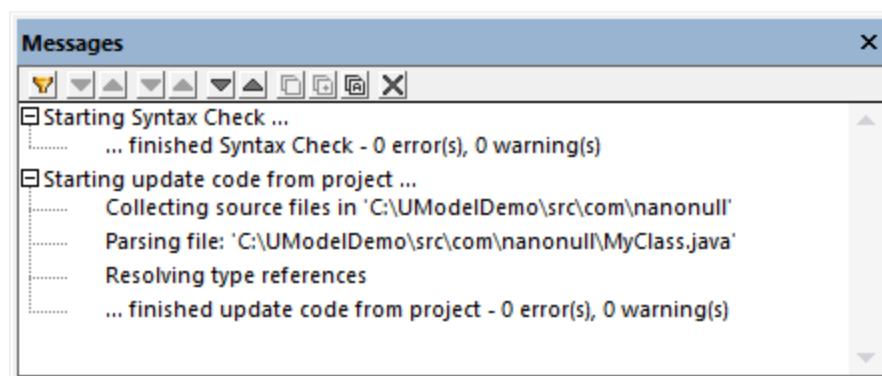
## Generating code

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

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



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



## Modifying code outside of UModel

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

```
package com.nanonull;
```

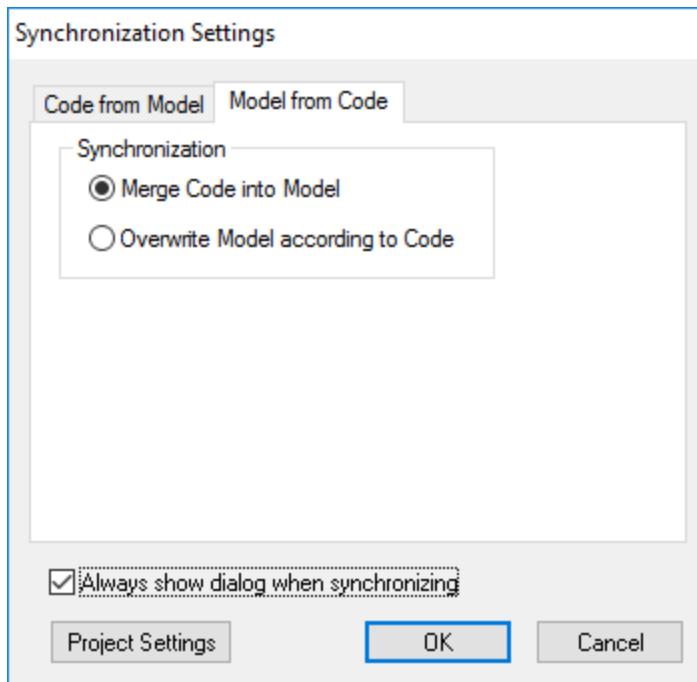
```
public class MyClass{
    public float sum(float num1, float num2) {
        return num1 + num2;
    }
}
```

MyClass.java

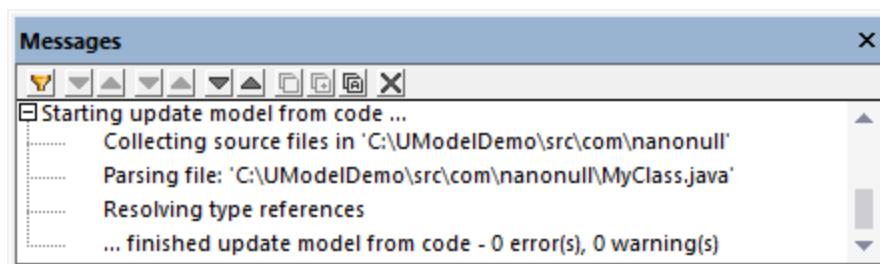
## Merging code changes back into the model

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

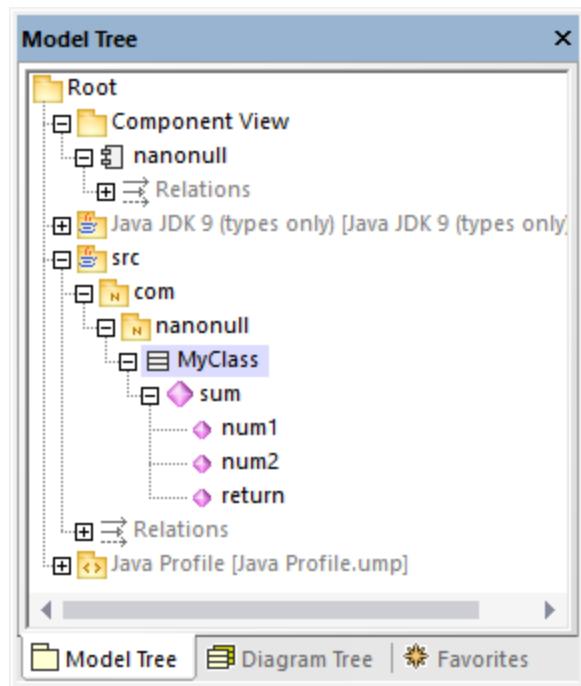
1. On the **Project** menu, click **Merge UModel Project from Program Code** (Alternatively, press **Ctrl + F12**).



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



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



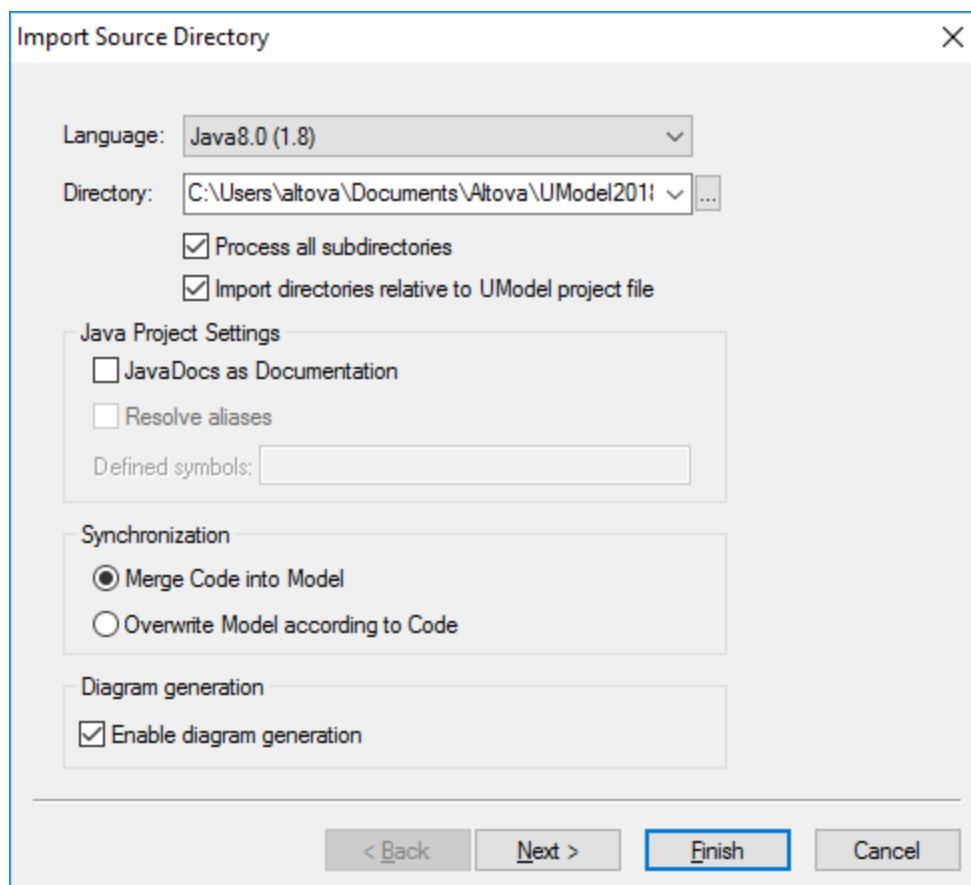
## 2.8 Reverse Engineering (from Code to Model)

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

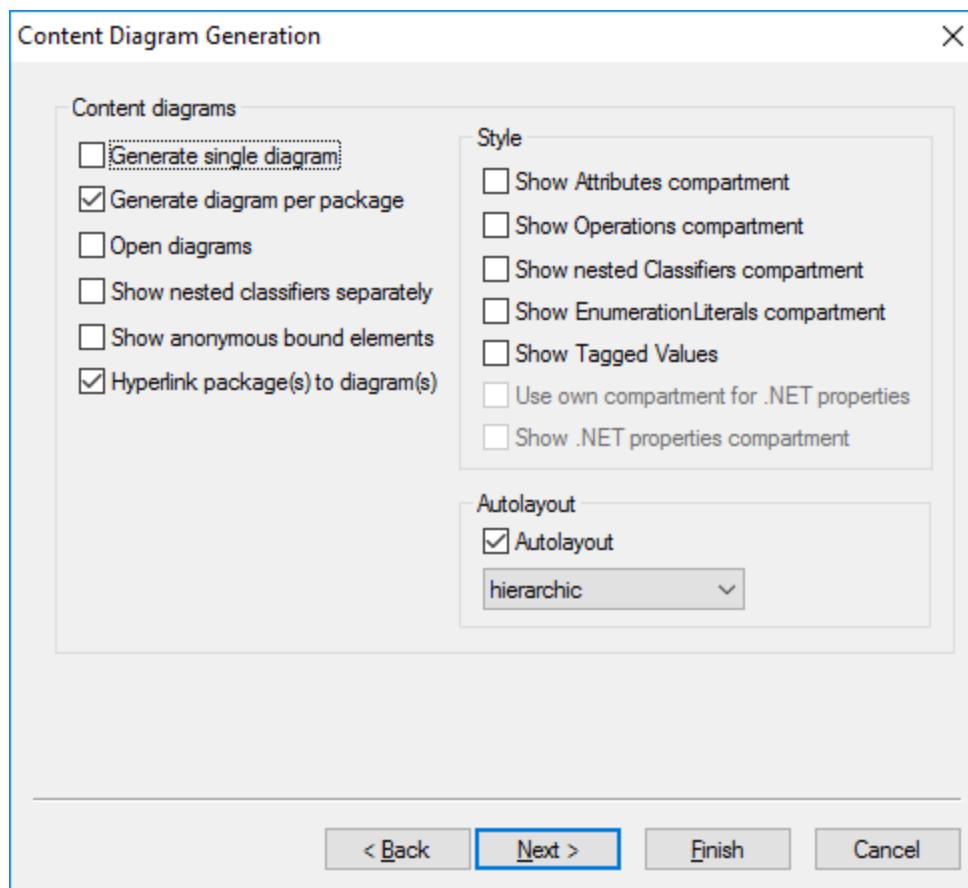
**Note:** The sample Java code used in this tutorial is available as a ZIP archive at the following path: **C:\Users\<username>\Documents\Altova\UModel2022\UModelExamples\OrgChart.zip**. Please unzip the archive to the same directory before starting the tutorial.

### Importing existing code from a directory

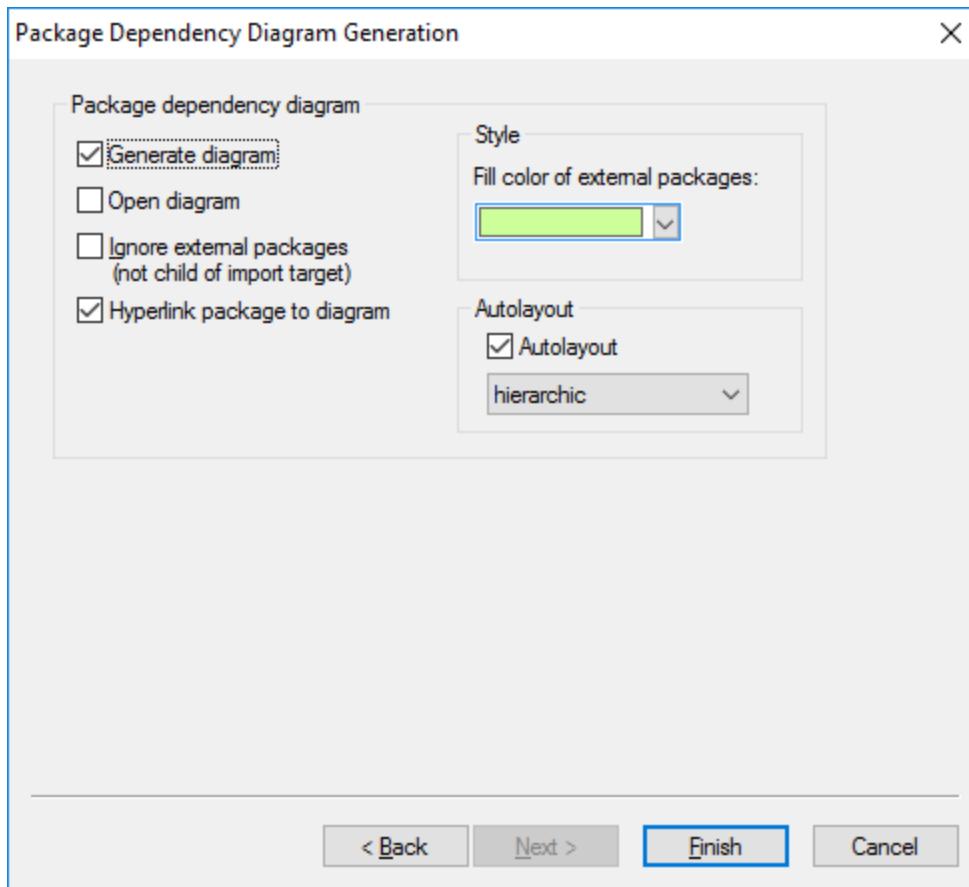
1. On the **File** menu, click **New**.
2. On the **Project** menu, click **Import Source Directory**.
3. Select the language of the source code (in this example, Java).
4. Click the Browse button  , select the **OrgChart** directory unzipped previously, and click **Next**. Notice the **Enable diagram generation** check box is selected, which instructs UModel to generate [Class Diagrams](#) <sup>380</sup> and [Package Diagrams](#) <sup>398</sup> from the source code.



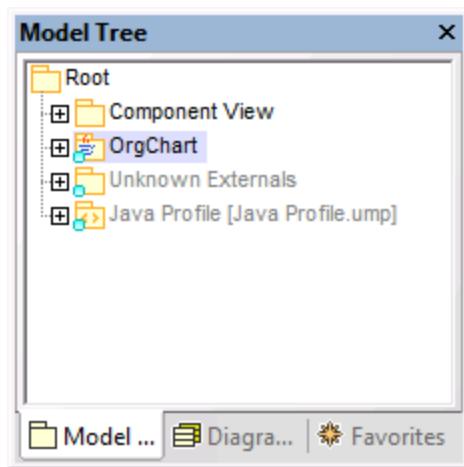
5. Select the **Generate diagram per package** option. This instructs UModel to create a new diagram for each package. The diagram styling options can be changed later if necessary.



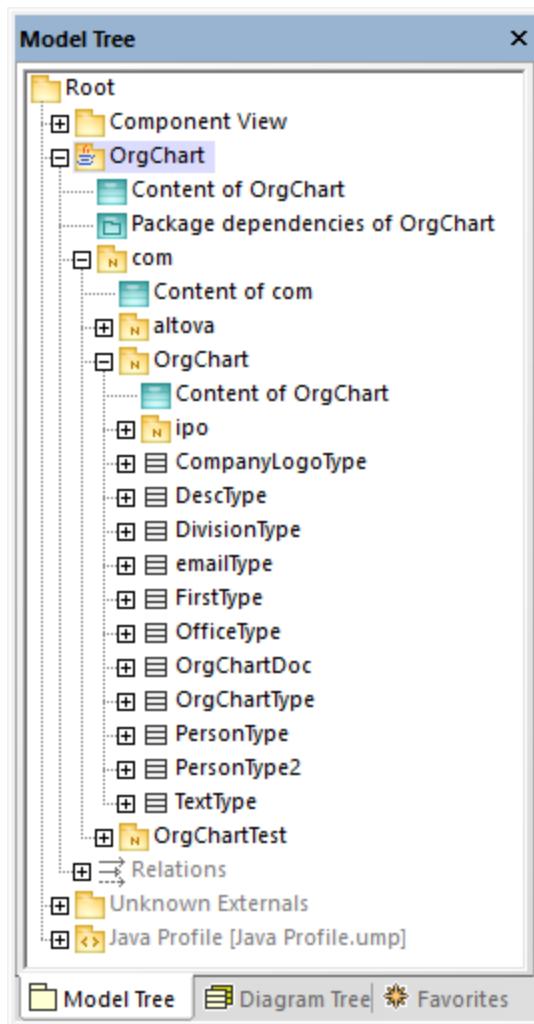
6. Click **Next** to continue. This dialog box allows you to define the package dependency generation settings.



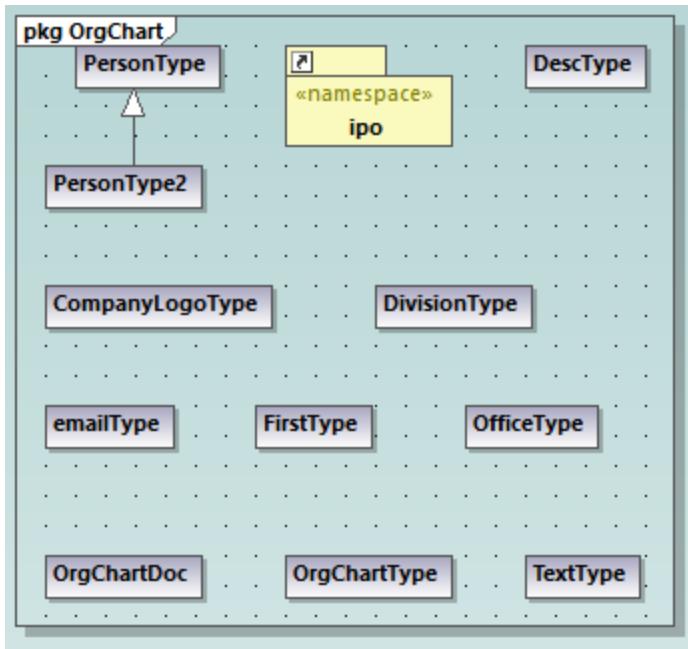
7. Click **Finish**. When prompted, save the new model to a directory on your system. The data is parsed, and a new package called "OrgChart" is created.



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



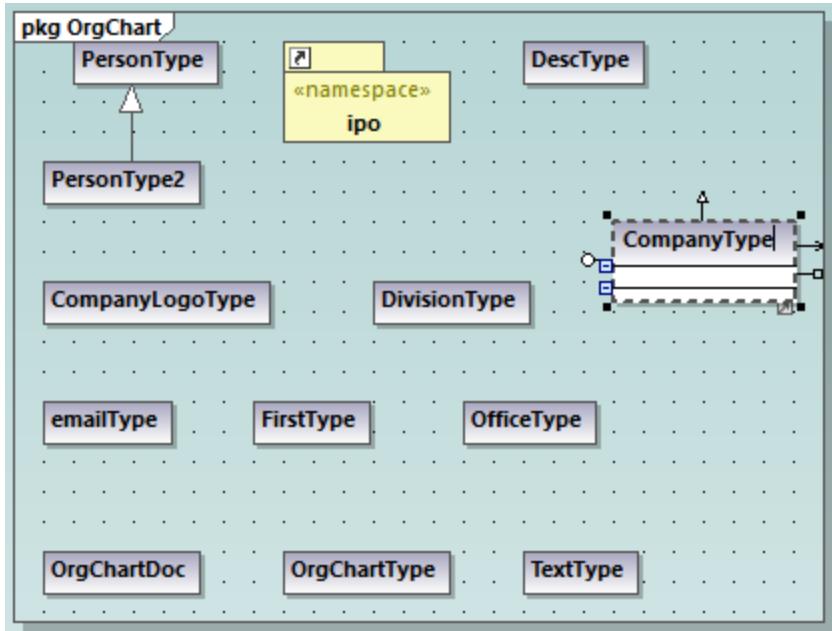
The "Content of OrgChart" diagram is now displayed in the main pane.



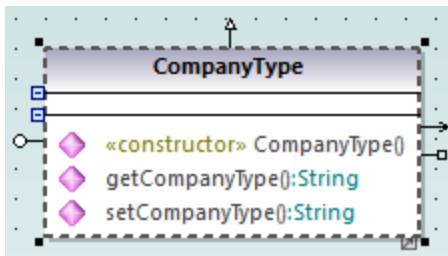
### Adding a new class to the OrgChart diagram

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

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



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

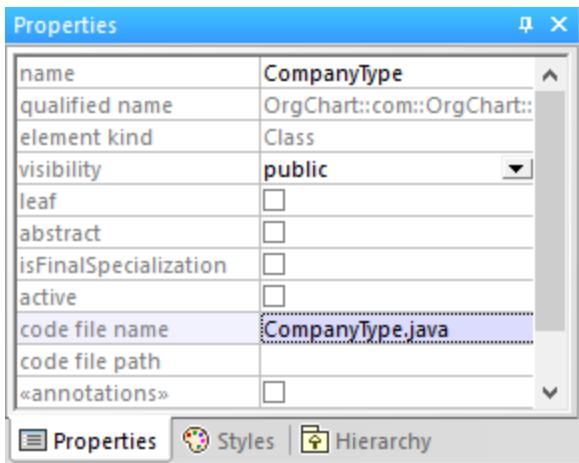


**Note:** Since the class name is `CompanyType`, the operation `CompanyType()` is automatically assigned the `<<constructor>>` stereotype.

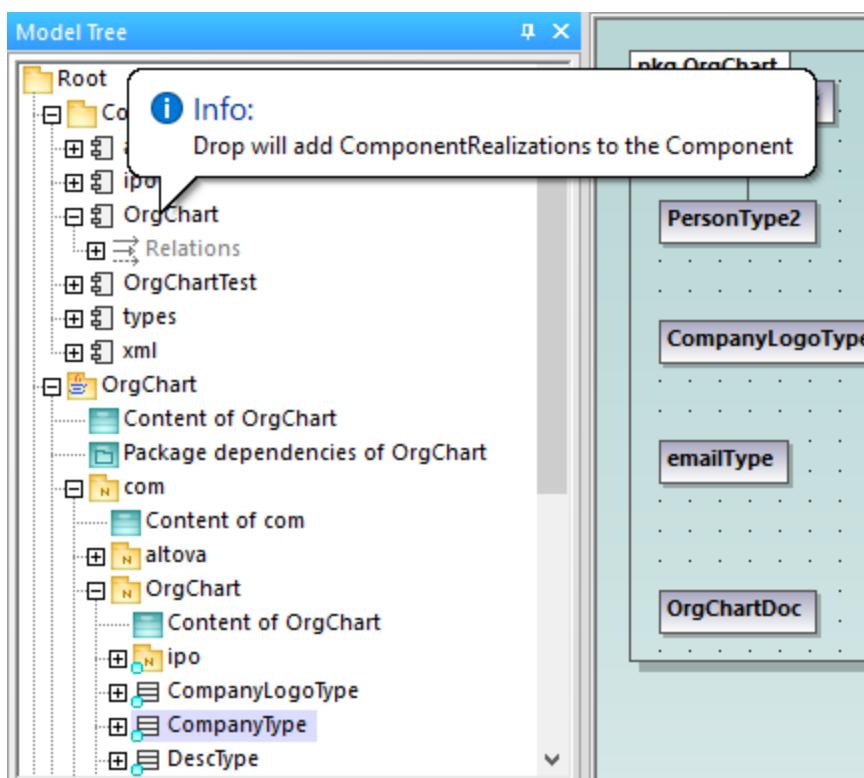
### Making the new class available for code generation

Now that the model has been extended with a new class, you will most likely want to update the underlying code accordingly, in order to keep both in sync. However, if you press **F11** to check the project syntax at this stage, a warning is displayed in the Messages window: '*'CompanyType' has no Component Realization to a Component - ComponentRealization to Component 'OrgChart' will be generated*'. The reason is that the new class requires realization to a component before code can be generated from it, as explained in [Round-Trip Engineering \(Model-Code-Model\)](#)<sup>60</sup>. In some cases (including this example), UModel can generate the required realization automatically; however, you can also define the realization dependency manually, as follows:

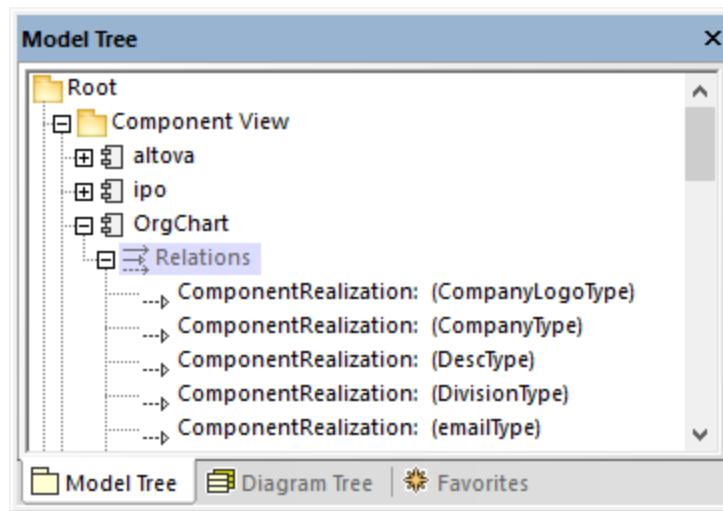
1. While the `CompanyType` class is selected in the diagram, locate the property "code file name" in the Properties window and enter "`CompanyType.java`" as file name.



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



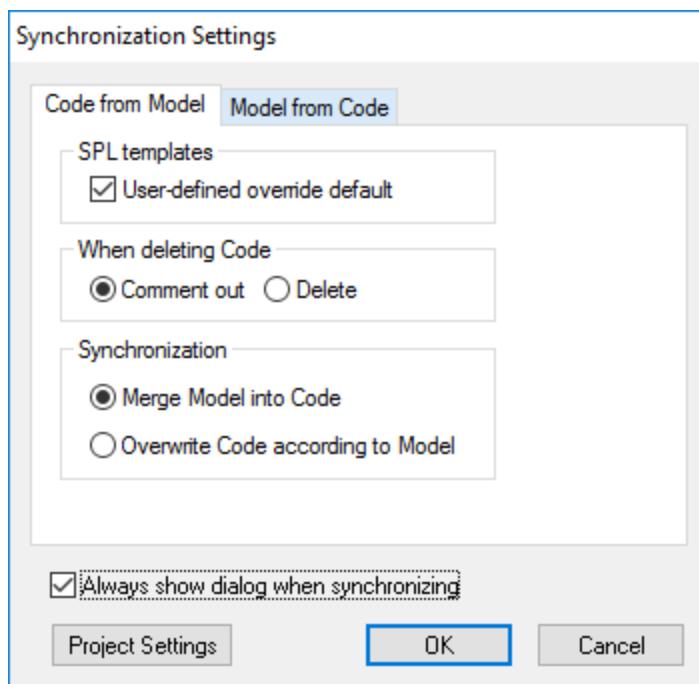
This method creates a relation of type "ComponentRealization" between a class and a component. An alternative way to do this is to draw the relation in a component diagram, see [Component Diagrams](#)<sup>49</sup>.  
 . Expand the **Relations** item below **OrgChart** to see the newly created relation.



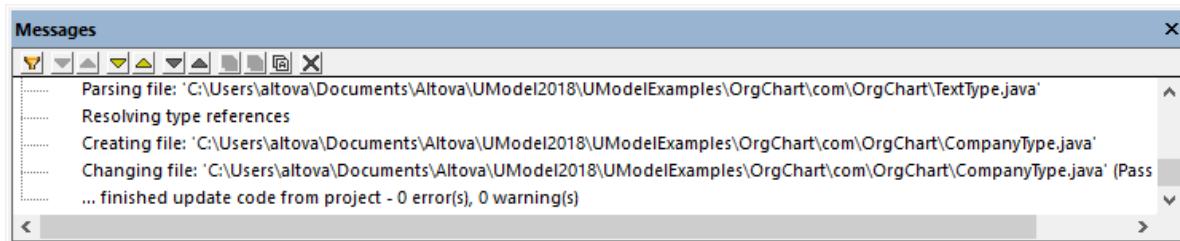
## Merging program code from a package

In UModel, you can generate code at package level, component level, or for the entire project, see also [Synchronizing the Model and Source Code](#)<sup>212</sup>. In this example, we will generate code at component level, as follows:

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



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



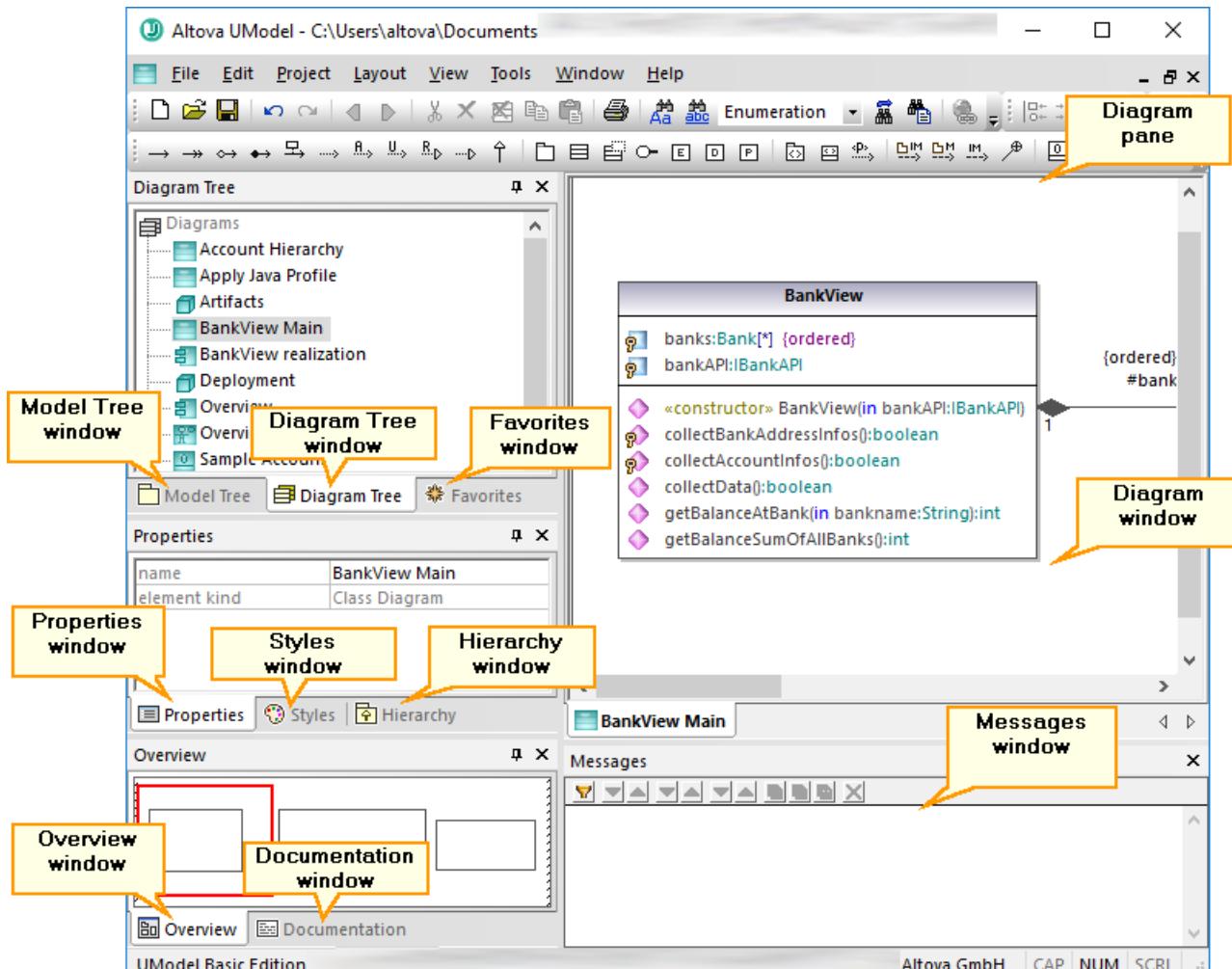
When the process completes, the new **CompanyType.java** class has been added to the folder ... \OrgChart\com\OrgChart\.

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

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

### 3 UModel Graphical User Interface

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



UModel graphical user interface

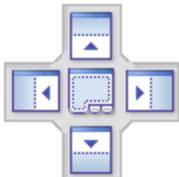
By default, the helper windows on the left side are docked in groups of three, and the Messages window appears below the diagram pane. You can, however, move and dock or undock any window as necessary. All windows can be searched using the **Find** combo box in the Main toolbar, or by pressing **Ctrl+F**. See also [Finding and Replacing Text](#)<sup>109</sup>.

#### To dock or undock a window:

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

**To move a window:**

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



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

**To reset all toolbars and windows to their default state:**

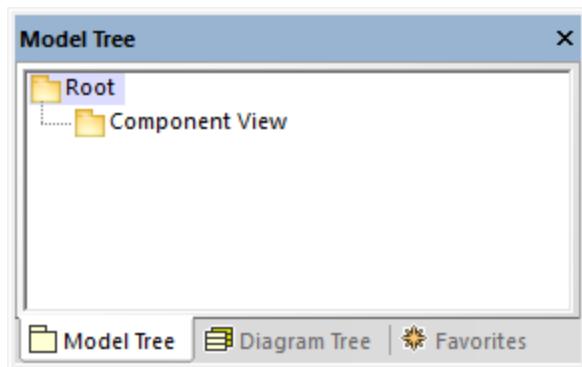
- On the **Tools** menu, click **Restore toolbars and Windows**.

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

- [Model Tree Window](#)<sup>79</sup>
- [Diagram Tree Window](#)<sup>83</sup>
- [Favorites Window](#)<sup>84</sup>
- [Properties Window](#)<sup>85</sup>
- [Styles Window](#)<sup>86</sup>
- [Hierarchy Window](#)<sup>87</sup>
- [Overview Window](#)<sup>89</sup>
- [Documentation Window](#)<sup>90</sup>
- [Messages Window](#)<sup>91</sup>
- [Diagram Window](#)<sup>93</sup>
- [Diagram Pane](#)<sup>94</sup>

## 3.1 Model Tree Window

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



Model Tree window

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

You can create additional packages, classes, diagrams, and their hierarchy either from this window or directly from a diagram, see [Creating Elements](#)<sup>104</sup>. For additional operations that you can take against items in the Model Tree, see the [How to Model...](#)<sup>103</sup> chapter.

**Note:** UModel includes several example projects that you can explore in order to learn the modeling basics and the graphical user interface. These can be found at the following path: C:  
\\Users\\<username>\\Documents\\Altova\\UModel2022\\UModelExamples.

### Showing, hiding, and sorting items in the Model Tree

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

### Collapsing and expanding items in the Model Tree

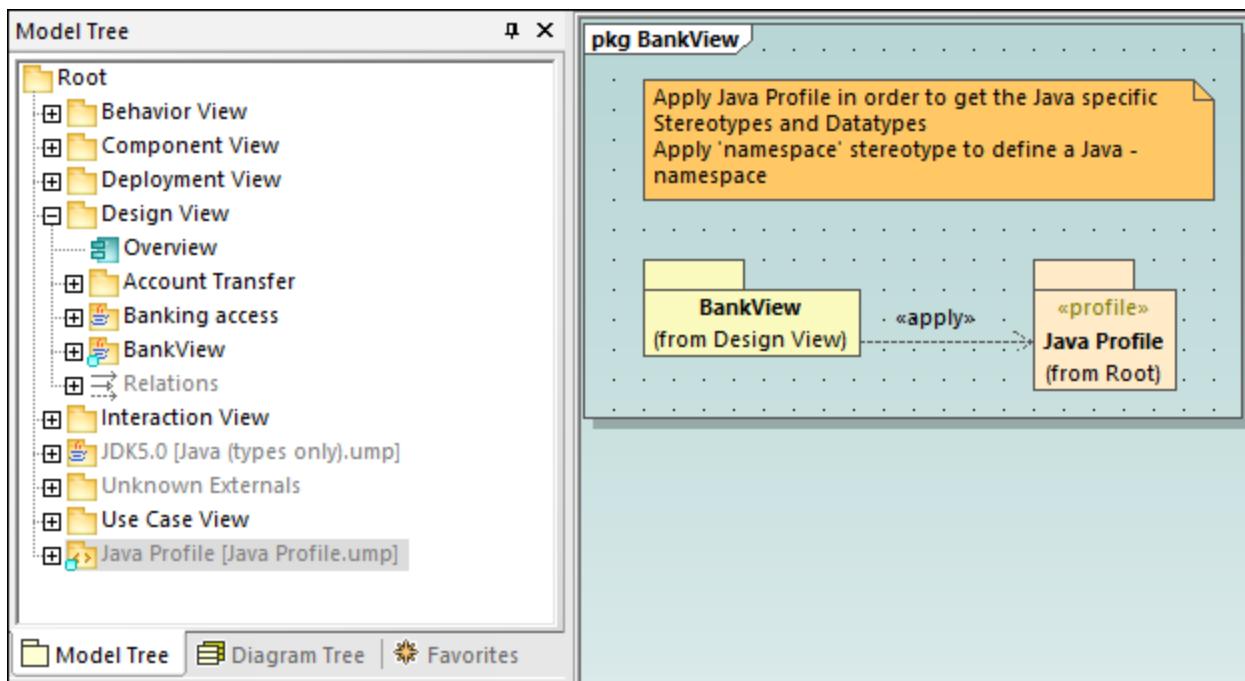
To expand items (for example, packages) in the Model Tree window:

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

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

## Identifying active diagram items

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



## Icon reference

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

Icon	Description
	Standard UML Package
	Java namespace root package. Used to generate or reverse engineer Java code
	C# namespace root package. Used to generate or reverse engineer C# code
	Visual Basic namespace root package. Used to generate or reverse engineer VB.NET code
	XML Schema namespace root package. Used to generate XML schemas from the model, or import them into the model, see <a href="#">XML Schema Diagrams</a> .
	A namespace package (a package with the <<namespace>> stereotype applied to it)
	A UML profile

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

Icon	Description
	Activity Diagram
	Class Diagram
	Communication Diagram
	Component Diagram
	Composite Structure Diagram
	Deployment Diagram
	Interaction Overview Diagram
	Object Diagram
	Package Diagram
	Profile Diagram
	Protocol State Machine Diagram
	Sequence Diagram
	State Machine Diagram
	Timing Diagram
	Use Case Diagram
	XML Schema Diagram

Below are some examples of UML modeling elements that can appear in the Model Tree window. For more information about UML elements and the diagram types where they occur, see the [UML Diagrams](#) 289 chapter.

Icon	Description
	Class
	Property
	Operation
	Parameter
	Actor
	Use Case
	Component

Icon	Description
	Node
	Artifact
	Interface
	Class Instance (Object)
	Class instance slot
	Relations
	Constraints

## 3.2 Diagram Tree Window

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

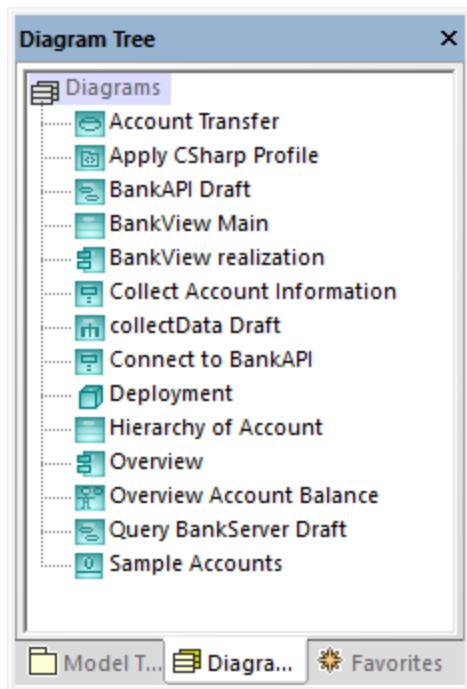


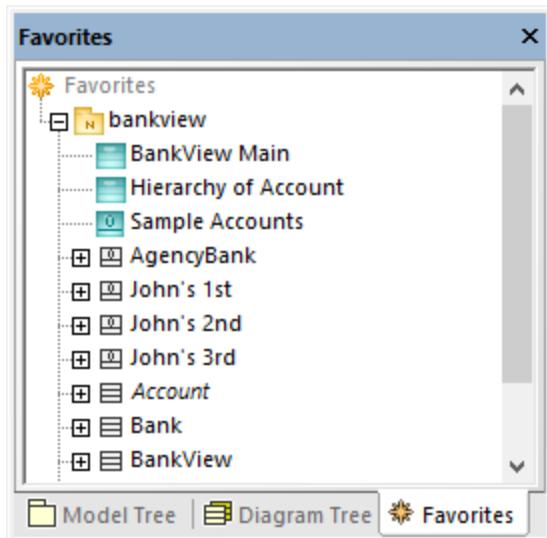
Diagram Tree window

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

For instructions about creating, opening, and generating diagrams, including how to model their content, refer to the [How to Model... 103](#) chapter. For specific information about each diagram type, refer to the [UML Diagrams 289](#) chapter.

### 3.3 Favorites Window

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



Favorites window

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

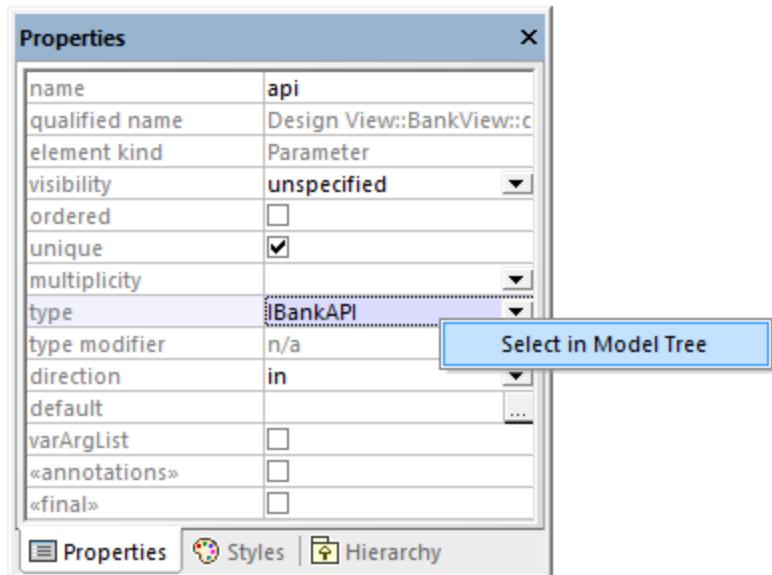
Items in the Favorites window are not copies or clones; they represent the actual elements or diagrams. Most actions that you take in the Model Tree window are also applicable in the Favorites window, including adding or deleting elements. For more information, see the [How to Model... \(103\)](#) chapter.

## 3.4 Properties Window

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

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

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



Properties window

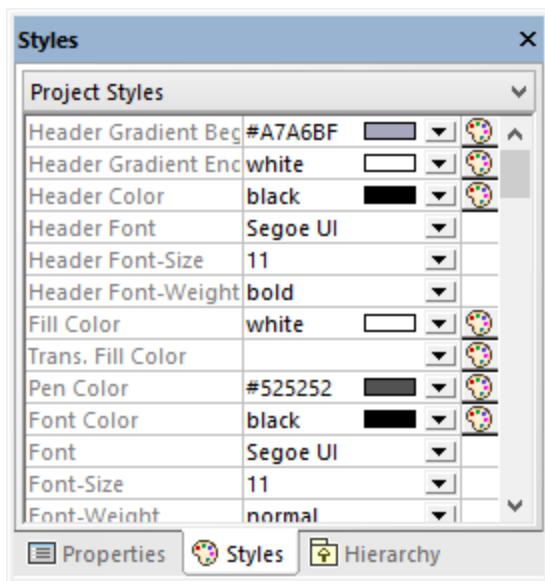
Changing a property of an element from the Properties window is immediately reflected in the diagram. Likewise, making a change in the diagram (for example, changing the visibility of an operation from `public` to `private`) affects the applicable property in the Properties window.

Properties that are enclosed within guillemets represent stereotypes (for example, «final»). You can add custom stereotypes to the project, in which case they would appear as properties in the Properties window, in addition to the default ones. For more information, see [Example: Creating and Applying Stereotypes](#) 409.

## 3.5 Styles Window

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

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



Styles window

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

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

- [Changing the Style of Elements](#) 117
- [Changing the Style of Diagrams](#) 123
- [Changing the Style of Lines and Relationships](#) 131

## 3.6 Hierarchy Window

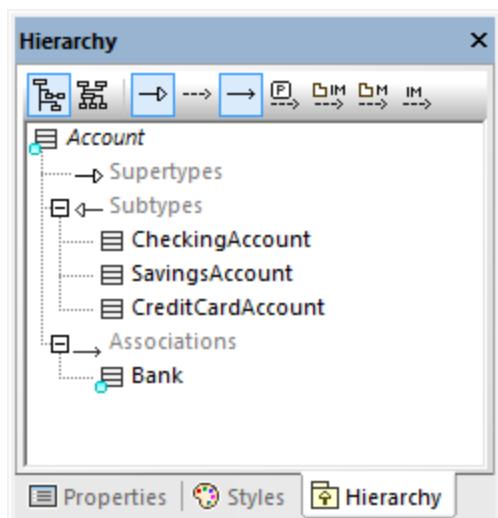
The Hierarchy window displays all relations of the currently selected modeling item, in two different views. The modeling element can be selected in a diagram, in the Model Tree window, or in the Favorites window.

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

- Tree view
- Graph view

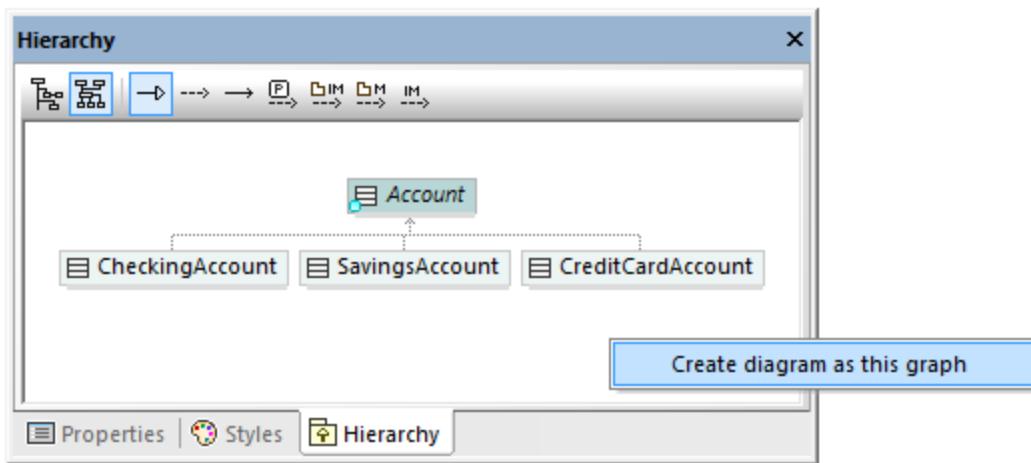
To switch between views, click the **Show tree view**  or **Show graph view**  buttons in the upper-left corner of the window.

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



Hierarchy window (tree view)

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



Hierarchy window (graph view)

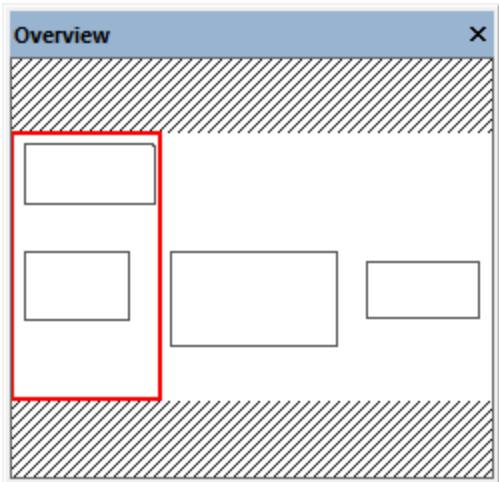
In the graph view, you can generate diagrams that include the elements visible in the window. To do this, right-click inside the window, and select **Create diagram as this graph** from the context menu.

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

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

## 3.7 Overview Window

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

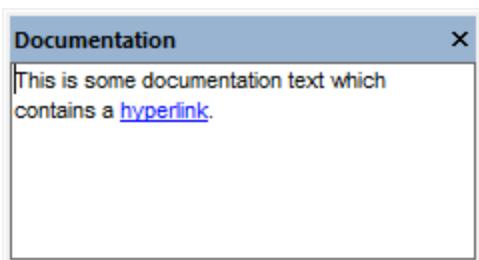


Overview window

See also [Zooming into/out of Diagrams](#) 129.

## 3.8 Documentation Window

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



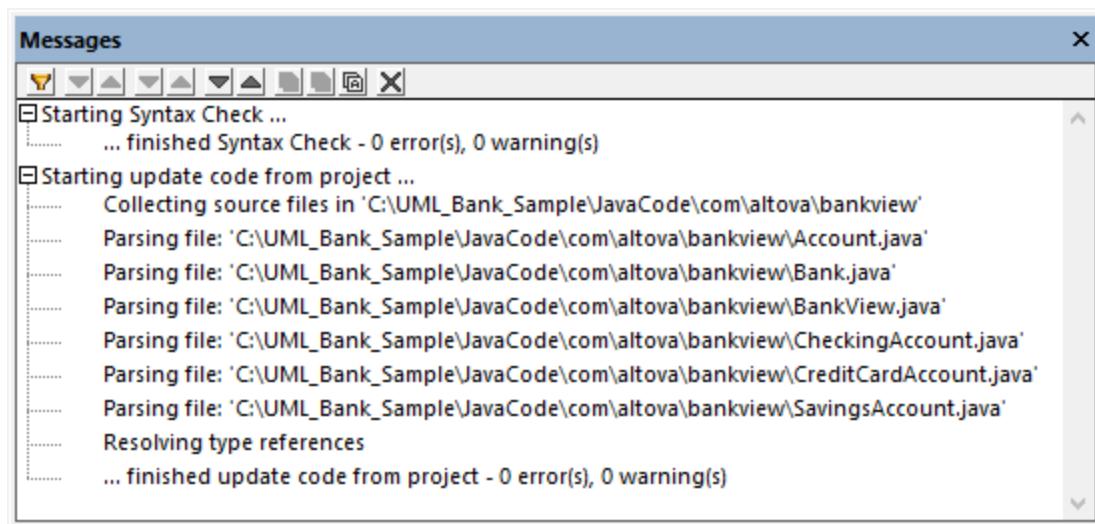
*Documentation window*

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

Documentation text can also be exported as comments in the generated source code, or imported from source code comments during reverse engineering. For more information, see [Documenting Elements](#)<sup>116</sup>.

## 3.9 Messages Window

The Messages window displays any of the following message types: information messages, warnings, and errors. Such messages may occur when you check the project syntax (see [Checking Project Syntax](#)<sup>167</sup>), or when you perform code engineering tasks. For more information about code engineering, see [Generating Program Code](#)<sup>164</sup> and [Importing Source Code](#)<sup>187</sup>.



Messages window

The table below lists possible message types and their icons.

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

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

Icon	Description
🔍	Filter messages by severity: information messages, and warnings. Select <b>Check All</b> to include all severity levels (this is the default behavior). Select <b>Uncheck All</b> to remove all severity levels from the filter.
➡	Jump to the next error.
⬅	Jump to the previous error.
⚠	Jump to the next warning.

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

## 3.10 Diagram Window

Whenever you create a new diagram, or open an existing one, a new Diagram window is loaded in the [Diagram Pane](#) <sup>94</sup>. The diagram window provides the canvas (drawing area) where you design UML diagrams. Various modeling commands are available when you right-click either the diagram canvas itself, or any element on it.

Importantly, the toolbar buttons and the context menu commands in UModel change based on the type of diagram that is currently active (in focus). For example, if you click inside a Class diagram, the toolbar buttons will include only elements applicable to class diagrams. To view the diagram type, click inside an empty area in the diagram, and observe the "element kind" property displayed in the [Properties window](#) <sup>85</sup>. The diagram type can also be distinguished by the icon accompanying the diagram, see [Creating Diagrams](#) <sup>119</sup>.

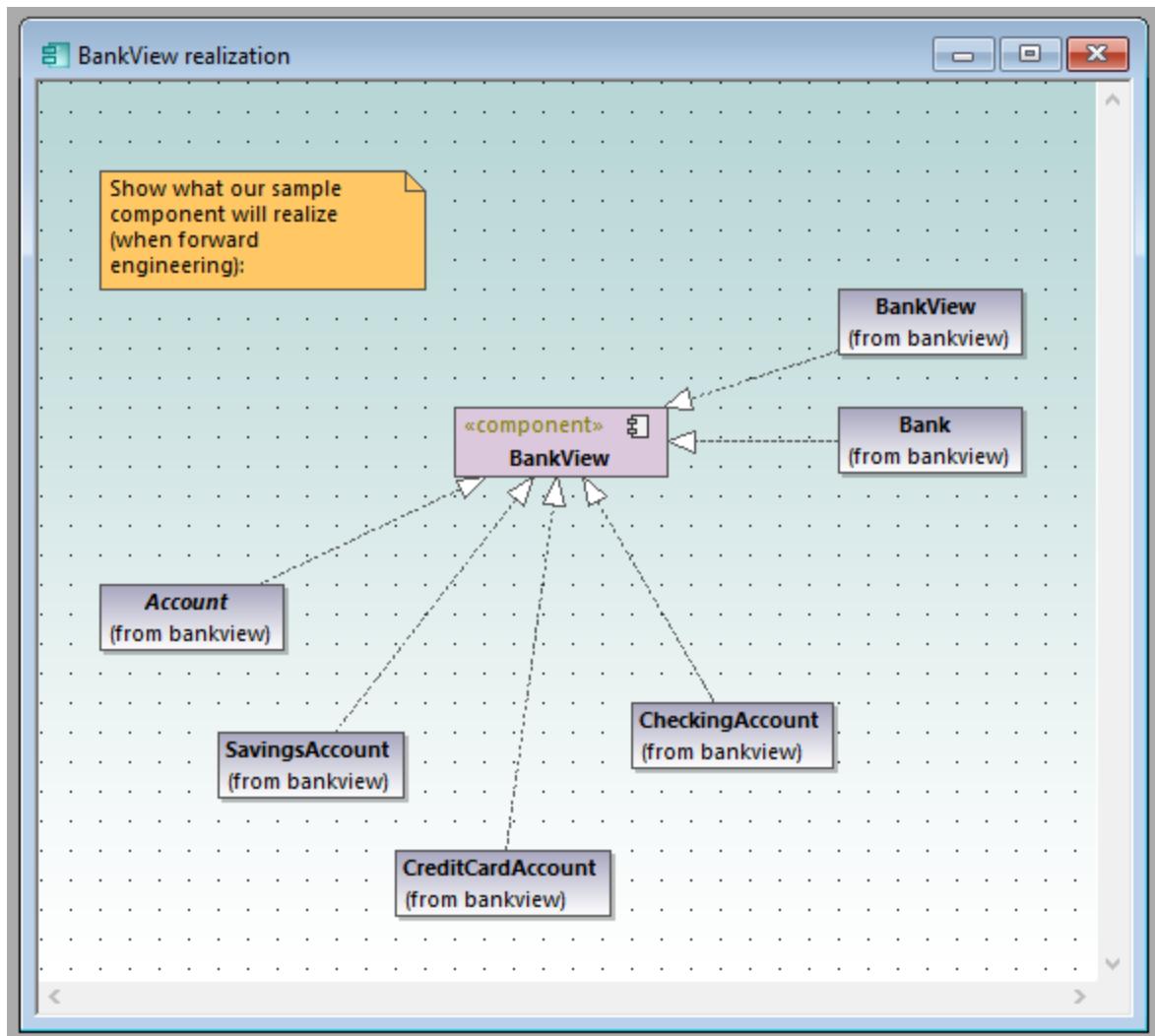


Diagram window

For information about creating new diagrams, opening existing ones, and manipulating elements inside the diagram, see the [How to Model...](#) <sup>103</sup> chapter.

## 3.11 Diagram Pane

The diagram pane hosts all diagram windows that are currently open. For information about creating new diagrams, opening existing ones, and manipulating elements inside the diagram, see the [How to Model... \(103\)](#) chapter.

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

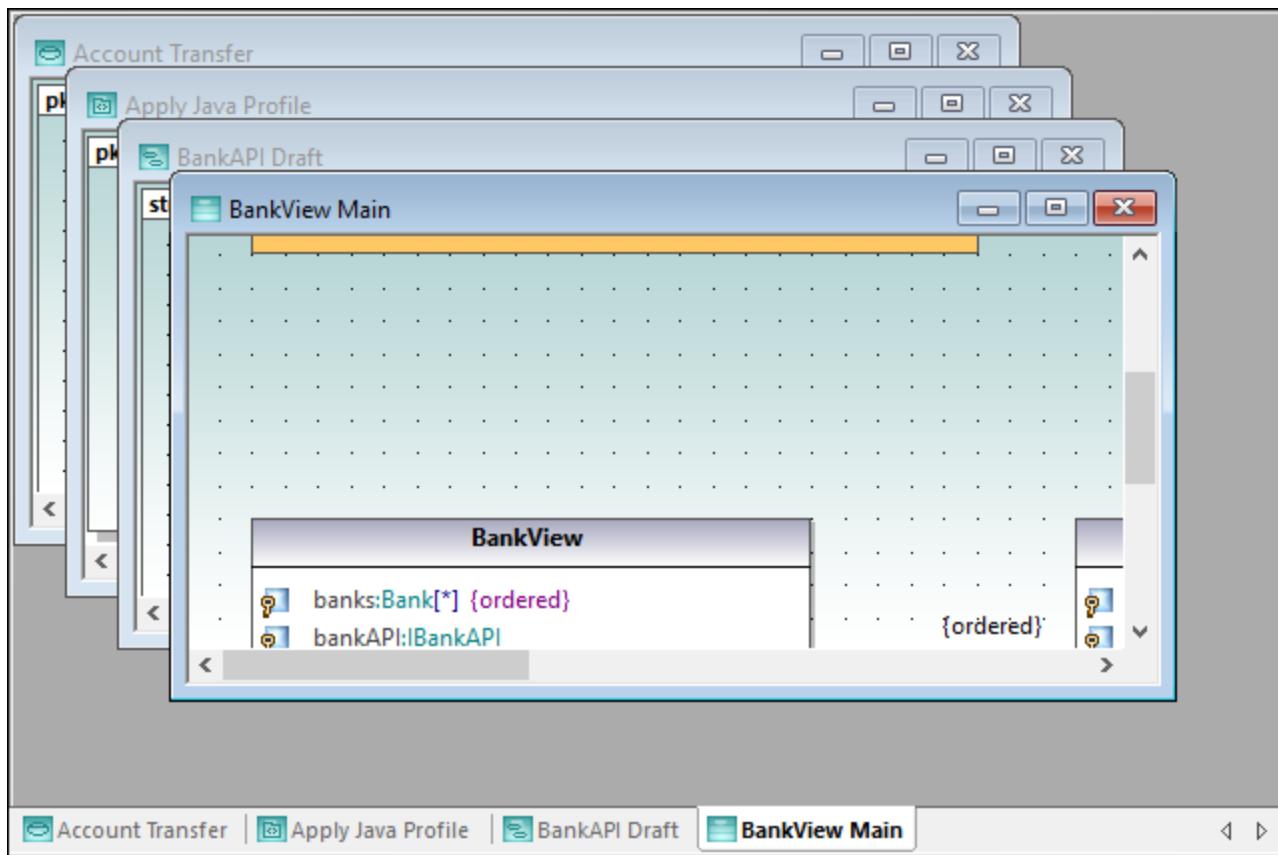
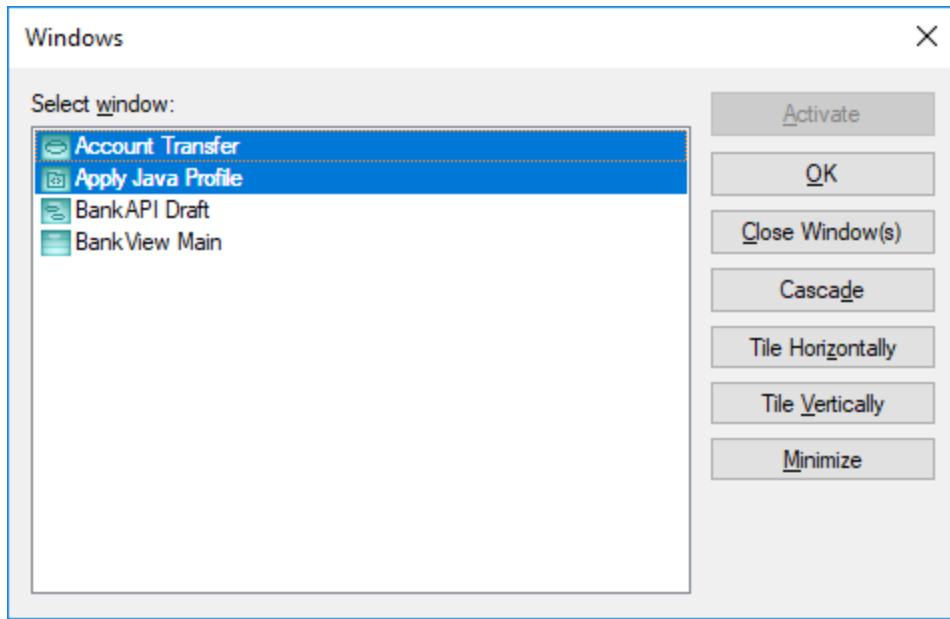


Diagram pane

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

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



*Windows dialog box*

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

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

## 4 UModel Command Line Interface

In addition to the graphical user interface, UModel also has a command line interface. To open the command line interface, run the **UModelBatch.exe** file available in the **C:\Program Files\Altova\UModel2022** directory. If you run UModel 32-bit on a 64-bit operating system, the path is **C:\Program Files (x86)\Altova\UModel2022**.

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

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

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

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

project      ... project file (*.ump)
/new[=file]  ... create/save/save as new project, see Creating, Loading, and Saving Projects in Batch Mode101
/set         ... set options permanent
/gui         ... display UModel user interface

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

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

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

```
Java10.0 | Java11.0 | Java12.0 | Java13.0 | Java14.0 |
Java15.0 |
C#1.2 | C#2.0 | C#3.0 | C#4.0 | C#5.0 | C#6.0 | C#7.0 |
C#7.1 | C#7.2 | C#7.3 | C#8.0 | C#9.0 |
VB7.1 | VB8.0 | VB9.0 |
C++98 | C++11 | C++14 | C++17)

/ipsd[=0|1] ... process sub directories (recursive)
/irpf[=0|1] ... import relative to UModel project file
/ijdc[=0|1] ... JavaDocs as Java comments
/icdc[=0|1] ... DocComments as C# comments
/icds[=lst] ... C# defined symbols
/ivdc[=0|1] ... DocComments as VB comments
/ivds[=lst] ... VB defined symbols (custom constants)
/icppdm[=lst] ... C++ defined macros
/icpphi[=0|1] ... read only C++ header files
/icpphc[=0|1] ... treat .h files a .cpp files
/icppms[=0|1] ... enable C++ Microsoft Compiler compatibility
/icppmv[=ver] ... MSVC version to use (1900 | 1800 | 1700 | 1600 | 1500 | 1400 | 1310
| 1300 | 1200)
/icppsy[=0|1] ... auto detect C++ system include files
/icppid[=lst] ... list of C++ include directories to use
/icppsd[=lst] ... list of C++ system include directories to use
/icppag[=arg] ... Additional C++ arguments for the compiler
/imrg[=0|1] ... synchronize merged
/iudf[=0|1] ... use directory filter
/iflt[=lst] ... directory filter (presets /iudf)

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

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

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

```

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

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

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

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

```

### Example 1: Import Java source code and preserve settings

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

```
"C:\Program Files\Altova\UModel2022\UModelBatch.exe" /new="C:\My
Projects\Fred.ump" /isd="X:\TestCases\UModel\Fred" /set /gui /iclg=Java8.0 /ipsd=1 /ijdc=1
/dgen=1 /dopn=1 /dmax=5 /chk
```

The meaning of all options is as follows:

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

## Example 2: Synchronize code from the model

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

```
"C:\Program Files\Altova\UModel2022\UModelBatch.exe" "C:
\UModel\Fred.ump" /m2c /ejdc=1 /ecod=1 /emrg=1 /egfn=1 /eusc=1
```

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

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

## Example 3: Import Java binaries into the model

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

```
"<C:\Program Files\Altova\UModel2022\UModelBatch.exe>" /new="C:
\JavaProject\Result.ump" /ibt=*C:
\JavaProject\bin /iclg=Java8.0 /ibrt=JDK1.8.0_144 /dgen=1 /chk
```

The options used are as follows:

/new	Creates a new UModel project at the specified path.
/ibt	Instructs UModel to import binary types. The asterisk before the path indicates that all binary types at that path must be imported.
/iclg	Specifies the code generation language ("Java8.0", in this example).
/ibrt	Specifies the runtime environment ("JDK1.8.0_144" in this example). This is the same value that appears on the "Import Binary Types" dialog box in the "Runtime" drop-down list, see <a href="#">Importing Java, C# and VB.NET Binaries</a> <sup>199</sup> . You can also use a value like "jdk-10.0.1" as set in the <code>JAVA_HOME</code> environment variable.  For C#, you can use the value <code>/ibrt:any</code> or otherwise values as they appear in the GUI in the "Runtime" drop-down list, making sure to omit any spaces. Examples:  <code>/ibrt:any</code> <code>/ibrt:.NET5</code> <code>/ibrt:.NETFramework4.8 (v4.8.3752)</code>

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

## 4.1 Creating, Loading, and Saving Projects in Batch Mode

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

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

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

new

**UModelBatch /new=xxx.ump (options)**

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

auto save

**UModelBatch xxx.ump (options)**

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

save

**UModelBatch xxx.ump (options) /new**

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

save as

**UModelBatch xxx.ump (options) /new=yyy.ump**

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

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

new

**UModelBatch /gui /new (options)**

creates a new project, executes options, nothing saved, the GUI is left open

save new

**UModelBatch /gui /new=xxx.ump (options)**

creates a new project, executes options, xxx.ump saved, the GUI is left open

user mode

**UModelBatch /gui xxx.ump (options)**

loads project xxx.ump, executes options, nothing saved, the GUI is left open

save

**UModelBatch /gui xxx.ump (options) /new**

loads project xxx.ump, executes options, xxx.ump is saved, the GUI is left open

save as

**UModelBatch /gui xxx.ump (options) /new=yyy.ump**

loads project xxx.ump, executes options, xxx.ump is saved as yyy.ump, the GUI is left open

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

## 5 How to Model...

Altova website:  [UML modeling](#)

This chapter provides instructions for creating and manipulating UML elements, diagrams, and relationships from the UModel graphical user interface. It is intended as a "how to" guide to modeling with UModel. The enclosed instructions are generic across UModel and not specific to a particular element or diagram type, unless explicitly mentioned. For information applicable to (and grouped by) each diagram type, refer to the [UML Diagrams](#)<sup>289</sup> chapter.

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

Elements	Diagrams	Relationships	Stereotypes
<a href="#">Creating Elements</a> <sup>104</sup>	<a href="#">Creating Diagrams</a> <sup>119</sup>	<a href="#">Creating Relationships</a> <sup>130</sup>	<a href="#">Stereotypes and Tagged Values</a> <sup>140</sup>
<a href="#">Inserting Elements from the Model into a Diagram</a> <sup>105</sup>	<a href="#">Generating Diagrams</a> <sup>120</sup>	<a href="#">Changing the Style of Lines and Relationships</a> <sup>131</sup>	<a href="#">Tagged Values</a> <sup>141</sup>
<a href="#">Renaming, Moving, and Copying Elements</a> <sup>107</sup>	<a href="#">Opening Diagrams</a> <sup>122</sup>	<a href="#">Viewing Element Relationships</a> <sup>133</sup>	<a href="#">Applying Stereotypes</a> <sup>142</sup>
<a href="#">Deleting Elements</a> <sup>108</sup>	<a href="#">Deleting Diagrams</a> <sup>123</sup>	<a href="#">Associations</a> <sup>133</sup>	<a href="#">Showing or Hiding Tagged Values</a> <sup>144</sup>
<a href="#">Converting Elements</a> <sup>109</sup>	<a href="#">Changing the Style of Diagrams</a> <sup>123</sup>	<a href="#">Collection Associations</a> <sup>136</sup>	
<a href="#">Finding and Replacing Text</a> <sup>109</sup>	<a href="#">Aligning and Resizing Modeling Elements</a> <sup>125</sup>	<a href="#">Containment</a> <sup>139</sup>	
<a href="#">Checking Where and If Elements Are Used</a> <sup>111</sup>	<a href="#">Type Autocompletion in Classes</a> <sup>127</sup>		
<a href="#">Constraining Elements</a> <sup>112</sup>	<a href="#">Zooming into/out of Diagrams</a> <sup>129</sup>		
<a href="#">Hyperlinking Elements</a> <sup>113</sup>			
<a href="#">Documenting Elements</a> <sup>116</sup>			
<a href="#">Changing the Style of Elements</a> <sup>117</sup>			

**Note:** UModel includes several example projects that you can explore in order to learn the modeling basics and the graphical user interface. These can be found at the following path: **C:\Users\<username>\Documents\Altova\UModel2022\UModelExamples**.

## 5.1 Elements

### 5.1.1 Creating Elements

With UModel, new elements can be created as follows:

- From the [Model Tree](#) <sup>79</sup> window. With this approach, elements are added to the model only, and you can insert them later into diagrams if necessary.
- From any diagram window. Any elements added to a diagram are also automatically added to the model as well. Should you need to delete an element later, you can choose whether it should be removed from the diagram only, or deleted from the model as well.

**To add elements from the Model Tree window:**

- In the [Model Tree](#) <sup>79</sup> window (or [Favorites](#) <sup>84</sup> window), right-click the element (for example, package) under which you want the new element to appear, and select **New Element | <Element Name>** from the context menu. For example, to add a new package under the "Root" package, right-click the "Root" package, and select **New Element | Package**.

**To add elements from the Diagram window:**

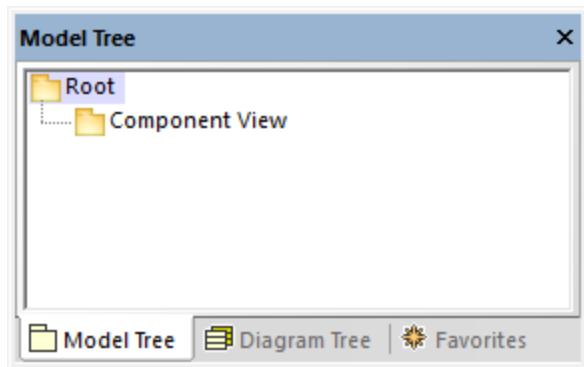
1. Create a new diagram (see [Creating Diagrams](#) <sup>119</sup>) or open an existing one (see [Opening Diagrams](#) <sup>122</sup>).
2. Do one of the following:
  - a. Right-click inside the diagram and select **New | <Element Name>** from the context menu.
  - b. Click the toolbar button of the element you wish to add, and then click inside the diagram. To insert multiple elements of the same type, hold down the **Ctrl** key before clicking inside the diagram.

## Packages

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

- They can be created at any position in the Model Tree.
- They can be moved or copied to other packages (as well as into valid model diagrams), see [Renaming, Moving, and Copying Elements](#) <sup>107</sup>.
- They can be used as source or target elements when code is generated or synchronized with the model, see [Forward Engineering \(from Model to Code\)](#) <sup>60</sup> and [Reverse Engineering \(from Code to Model\)](#) <sup>69</sup>.

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



*Default UModel packages*

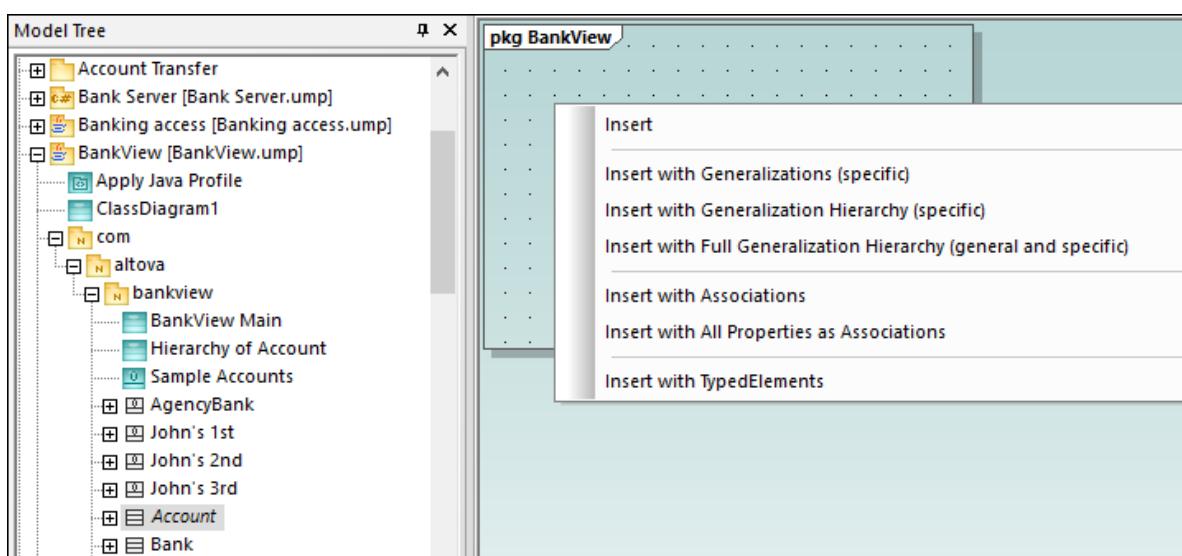
### 5.1.2 Inserting Elements from the Model into a Diagram

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

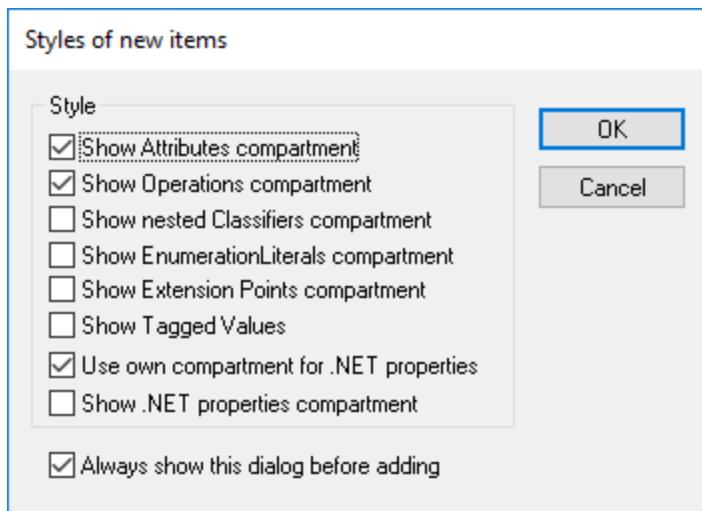
- **Drag left** (holding down the left mouse button and releasing it in the diagram) inserts elements immediately at the cursor position. In this case, any associations, dependencies etc. that exist between the currently inserted elements and the new one, are automatically displayed.
- **Drag right**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

For example, let's suppose that you want to create a new class diagram from a class that already exists in the model. To illustrate this scenario, open the sample project **Bank\_MultiLanguage.ump** available at the following path: **C:\Users\<username>\Documents\Altova\UModel2022\UModelExamples**. Assuming that you want to replicate the "Account Hierarchy" diagram in a new class diagram, do the following:

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



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



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

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

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

### 5.1.3 Renaming, Moving, and Copying Elements

You can cut, copy, rename and move elements in the [Model Tree](#) 79 window and inside diagrams of the same type. These actions may also be possible across diagrams of different type if applicable. You can also copy or move elements from the Model Tree window into a diagram, provided that the diagram is allowed to contain the corresponding element according to the UML specification.

#### To rename an element:

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

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

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

#### To copy or move elements:

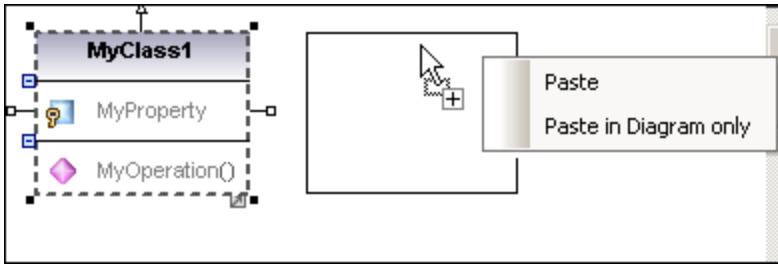
- Use the standard Windows commands **Cut**, **Copy**, or **Paste**. These commands can be triggered from keyboard shortcuts (**Ctrl+X**, **Ctrl+C**, **Ctrl+V**, respectively), from the corresponding toolbar buttons, as well as from the **Edit** menu.
- Alternatively, drag an element to a destination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

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

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

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

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



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

#### 5.1.4 Deleting Elements

Elements can be deleted in one of the following ways:

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

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

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

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

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

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

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

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

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

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

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

## 5.1.5 Converting Elements

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

- Class
- Interface
- Enumeration
- PrimitiveType
- DataType

You can convert the element kinds listed above either from the [Diagram window](#)<sup>83</sup> or from the [Model Tree](#)<sup>79</sup>.

### To convert elements:

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

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

## 5.1.6 Finding and Replacing Text

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

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

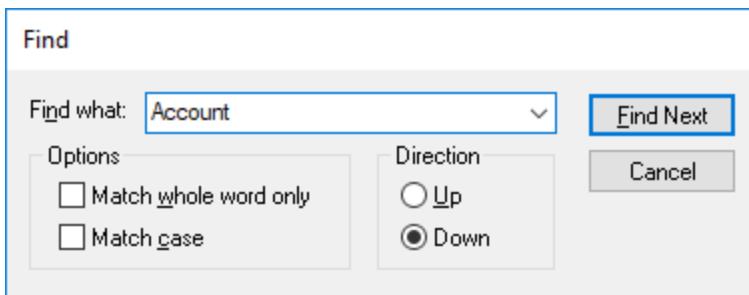
The search scope is applicable to the window where the cursor is currently placed. Therefore, if you want to search for text inside a diagram, for example, click inside the diagram first. Likewise, if you want to search for an item in the UModel project, click inside the Model Tree window first.

**To search for text or elements:**

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



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

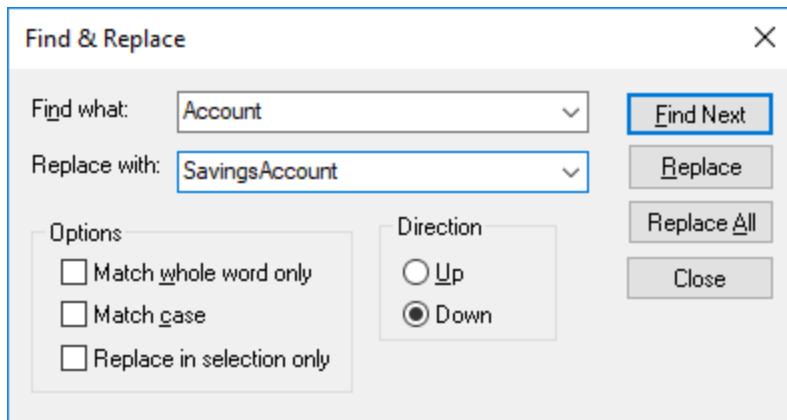
**Find and replace**

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

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

**To find and replace text:**

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



### 5.1.7 Checking Where and If Elements Are Used

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

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

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

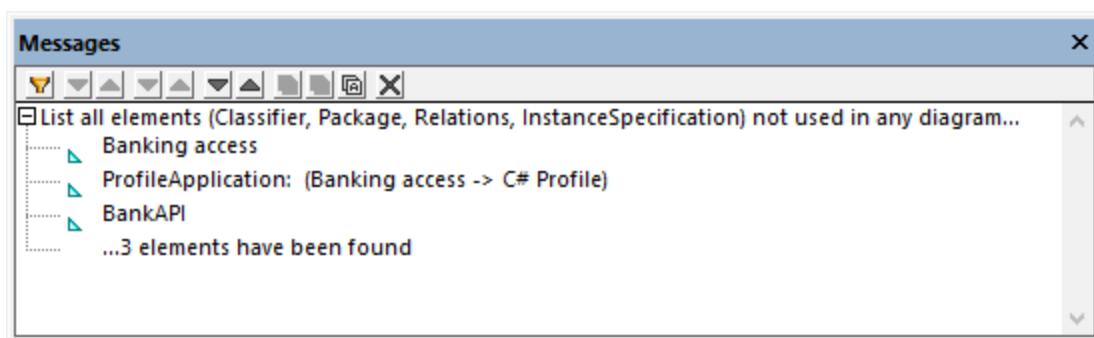
**To find unused elements in the entire project:**

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

**To find unused elements for a specific package:**

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

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



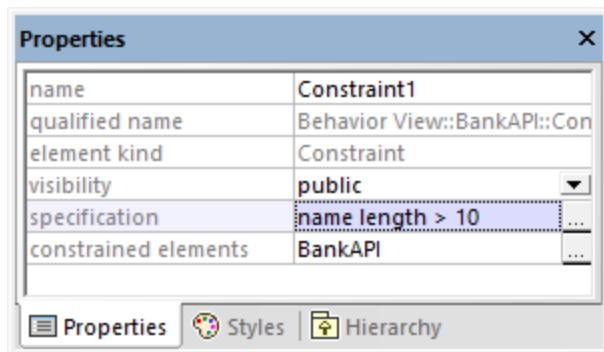
Click the element name in the Messages window to locate it in the Model Tree.

### 5.1.8 Constraining Elements

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

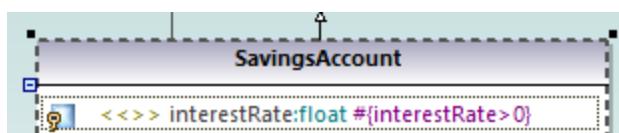
#### To constrain an element (from the Model Tree):

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



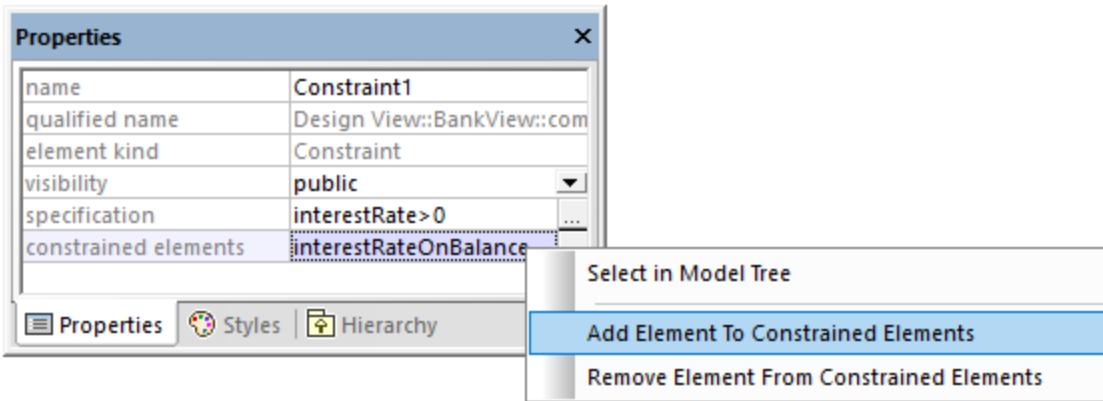
#### To constrain an element (from a diagram):

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

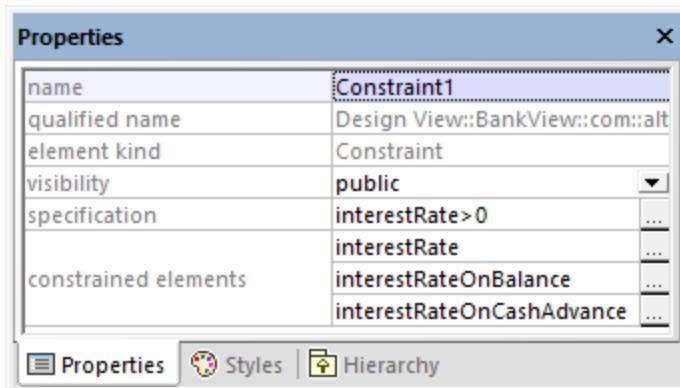


### To assign constraints to multiple modeling elements:

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



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



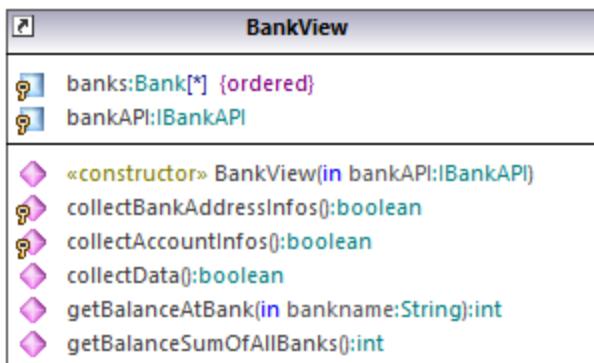
The "constrained elements" field contains the names of the modeling elements it has been assigned to. For example, in the image above, Constraint1 has been assigned to the following properties: interestRate, interestRateOnBalance, interestRateOnCashAdvance.

## 5.1.9 Hyperlinking Elements

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

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

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



Class containing hyperlinks

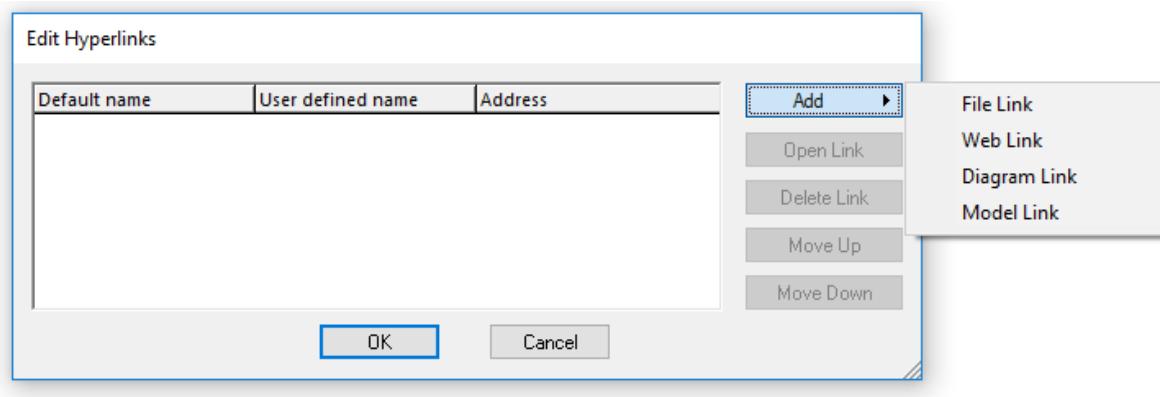
**Tip:** As you navigate through the UModel graphical user interface, either with or without hyperlinks, you can easily go back and forward between views by clicking the **Back**  or **Forward**  toolbar buttons, respectively.

You can automatically generate hyperlinks between dependent packages and diagrams when importing source code or binary files into a model, provided that you selected the specific settings on the import dialog box. For more information, see [Importing Source Code](#)<sup>187</sup> and [Importing Java, C# and VB.NET Binaries](#)<sup>199</sup>. Also, when you generate UML documentation from the project, you can choose whether to include hyperlinks in the generated output, see [Generating UML documentation](#)<sup>278</sup>.

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

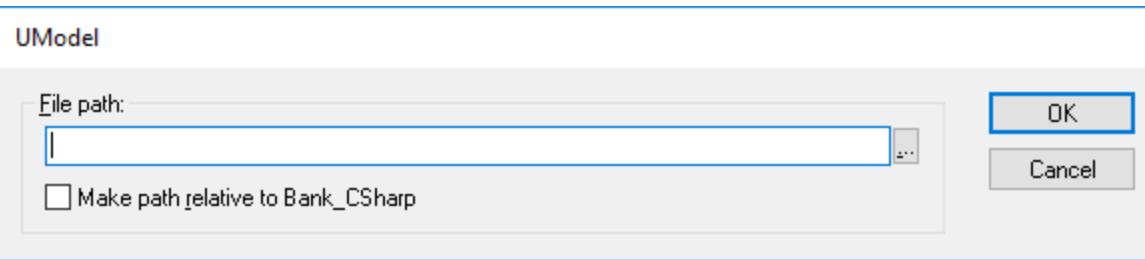
#### To create a hyperlink from an element:

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

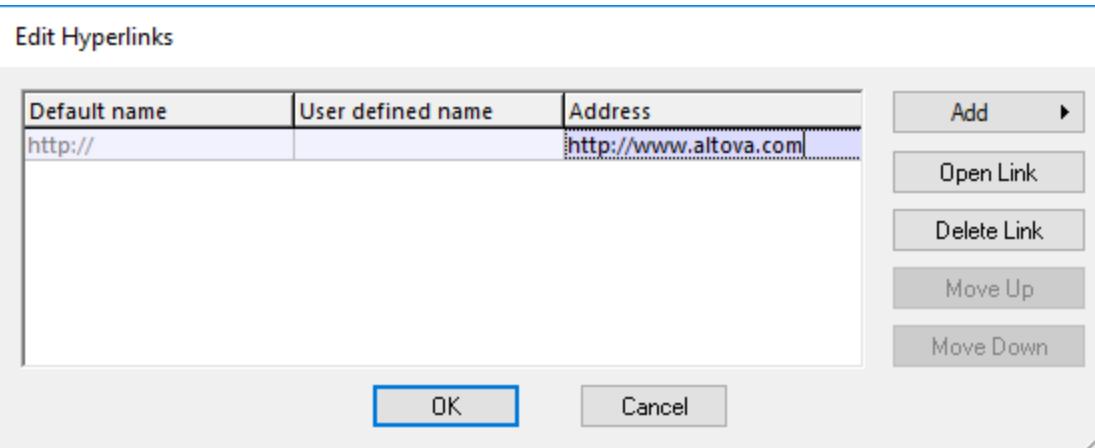


3. Do one of the following:

- To create a diagram or hyperlink, select the target element or diagram when prompted.
- To create a file hyperlink, click the Ellipsis button and browse for the target file.



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



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

**To create a hyperlink inside a note:**

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

This is a [hyperlink](#) inside a note.

### To change or remove a hyperlink:

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

## 5.1.10 Documenting Elements

You can add documentation comments to modeling elements as follows:

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

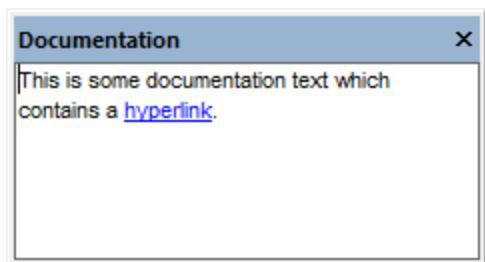
Any documentation text will be saved together with the project.

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

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

### Documentation hyperlinks

To create a hyperlink inside the Documentation window, select some text inside the window, right-click it and then select **Insert/Edit Hyperlinks** from the context menu. The hyperlink target can be a Web site, a diagram, a file, or another element, see also [Hyperlinking Elements](#)<sup>113</sup>.



Documentation window

### Code generation and documentation comments

If you generate code from class diagrams, any comments applied to classes and their members (in class diagrams) can be exported to the generated code as well. To do this, select the check box **Write Documentation as Java Docs** (for Java) or **Write Documentation as DocComments** (for C#, VB.NET) before generating program code, see also [Code Generation Options](#)<sup>169</sup>.

Likewise, if you reverse engineer program code into a model, the code comments can be imported into the model. To do this, select the check box **JavaDocs as Documentation** (for Java) or **DocComments as Documentation** (for C#, VB.NET) before reverse engineering program code, see also [Code Import Options](#)<sup>189</sup>.

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

- [C# Mappings](#)<sup>219</sup>
- [VB.NET Mappings](#)<sup>239</sup>
- [Java Mappings](#)<sup>253</sup>
- [XML Schema Mappings](#)<sup>259</sup>

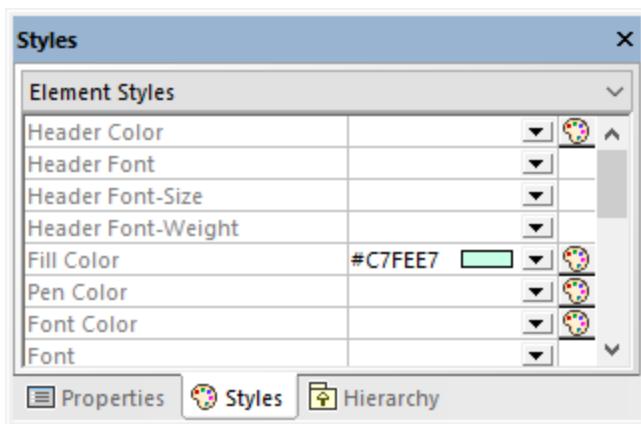
## 5.1.11 Changing the Style of Elements

You can change the appearance (style) of modeling elements, including their color, font size, font weight, background color, line thickness, and others. The appearance of elements can be changed at various levels: globally for all elements in the project, selectively for all elements of the same family (for example, classes), or for each individual element. For information about changing the style of the diagram itself, see [Changing the Style of Diagrams](#)<sup>123</sup>.

If you would like to use custom images instead of conventional element representations in diagrams, this is possible by extending your project with custom profiles and stereotypes. For more information, see [Example: Customizing Icons and Styles](#)<sup>414</sup>.

### To change the appearance of elements:

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

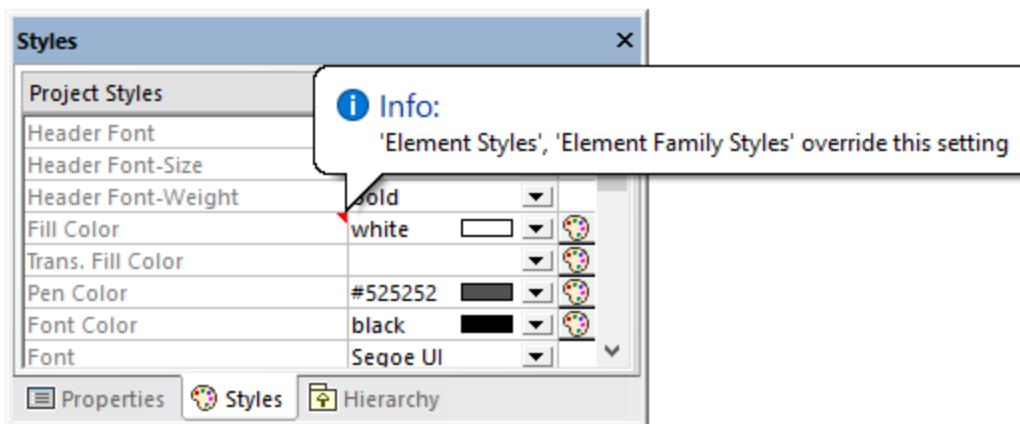


- b. To edit the properties of all elements of the same kind (for example, classes), select "Element Family Styles" from the list.
  - c. To edit the properties of all elements globally at the project level, select "Project Styles".
  - d. To edit the properties of all lines in the project, including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

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

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

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



Overridden element style

## 5.2 Diagrams

### 5.2.1 Creating Diagrams

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

**To create a new diagram:**

1. Right-click a package in the [Model Tree window](#)<sup>79</sup>.
2. Select **New Diagram | <Diagram Kind>**.

You can also create a new diagram from the [Diagram Tree window](#)<sup>83</sup>, as follows:

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

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

Icon	Description
	Activity Diagram
	Class Diagram
	Communication Diagram
	Component Diagram
	Composite Structure Diagram
	Deployment Diagram
	Interaction Overview Diagram
	Object Diagram
	Package Diagram
	Profile Diagram
	Protocol State Machine Diagram
	Sequence Diagram
	State Machine Diagram

Icon	Description
	Timing Diagram
	Use Case Diagram
	XML Schema Diagram

## 5.2.2 Generating Diagrams

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

- [Generating Class Diagrams](#) 392
- [Generating Sequence Diagrams from Source Code](#) 359
- [Generating Package Diagrams While Importing Code or Binaries](#) 401

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

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

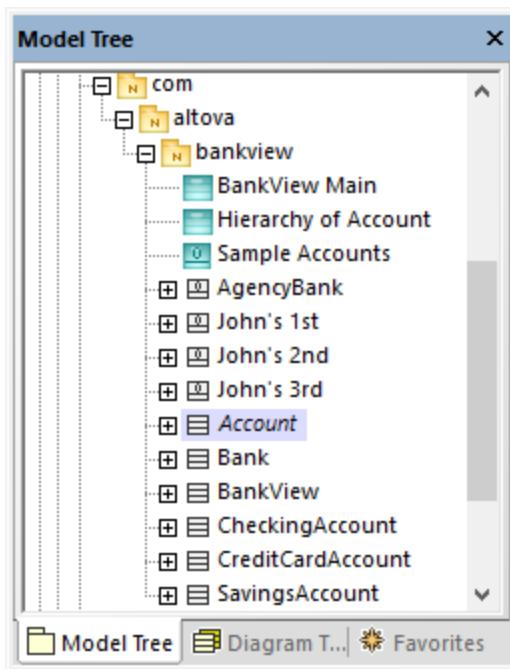
- Right-click a package in the Model Tree window and select **Show in new Diagram | Content** from the context menu.

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

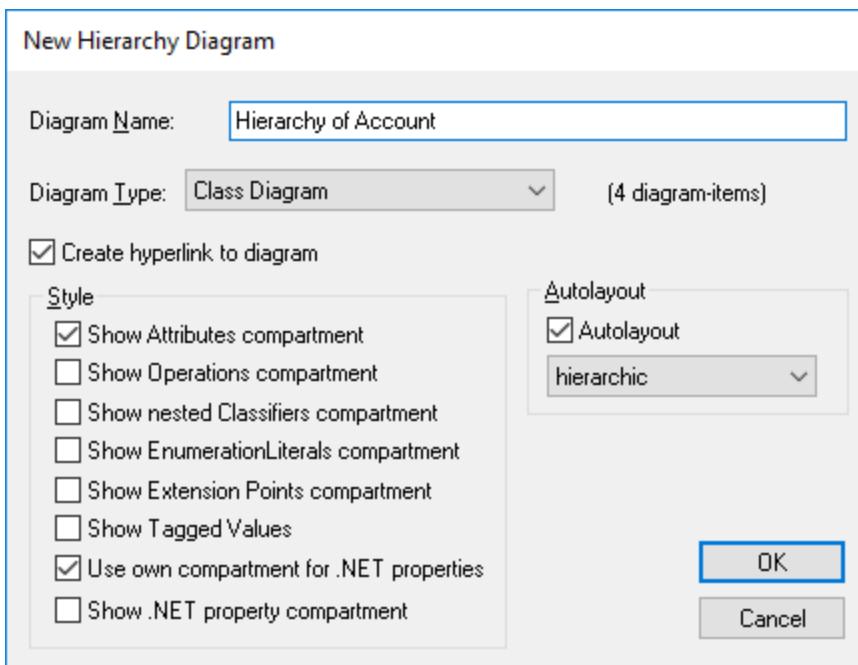
- Right-click a package in the Model Tree window and select **Show in new Diagram | Package dependencies** from the context menu.

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

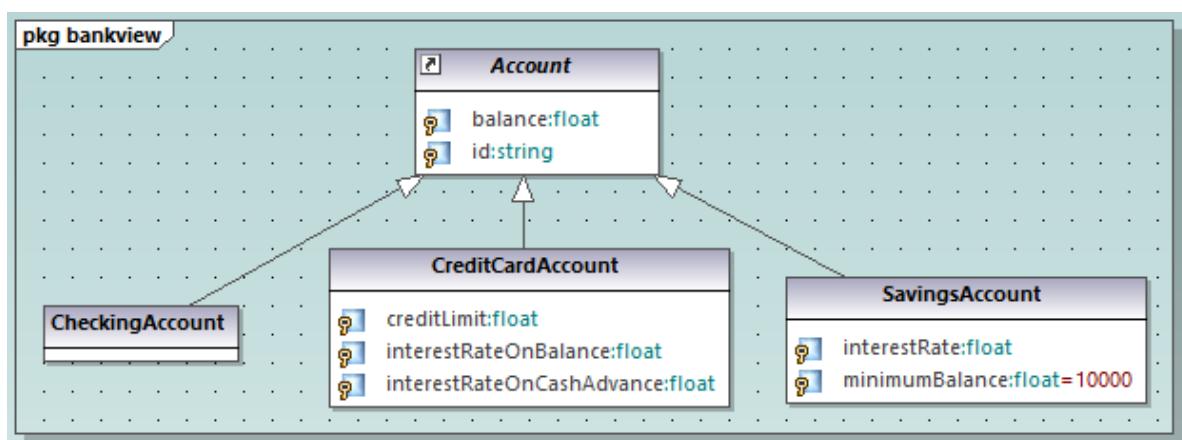
1. In the Model Tree window, right-click a class which has generalization relationships to or from other classes (for example, class `Account` from the sample project **C:\Users\<username>\Documents\Altova\UModel2022\UModelExamples\Bank\_CSharp.ump**).



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



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



### 5.2.3 Opening Diagrams

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

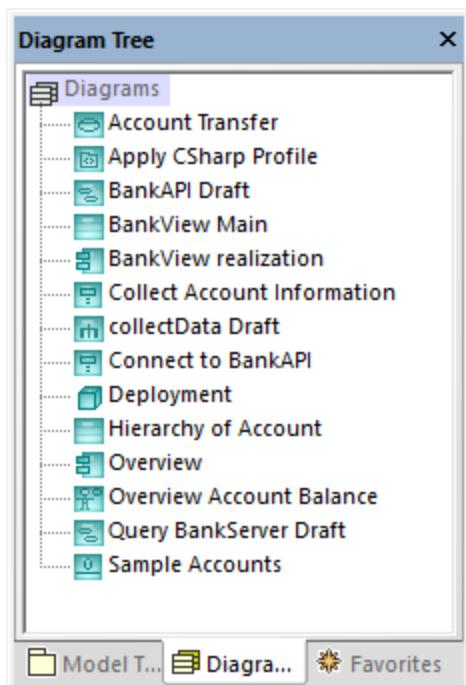
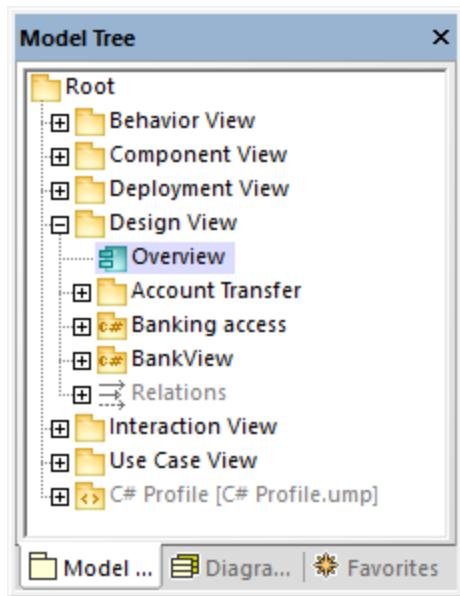


Diagram Tree window

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

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



#### To open an existing diagram:

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

### 5.2.4 Deleting Diagrams

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

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

Deleting a diagram does not remove any elements from the project except the diagram itself. To check if elements are used in any diagrams, right-click the package you would like to inspect, and select **List elements not used in any diagram**, see also [Checking Where and If Elements Are Used](#)<sup>111</sup>.

For information about deleting elements from a diagram or from a project, see [Deleting Elements](#)<sup>108</sup>.

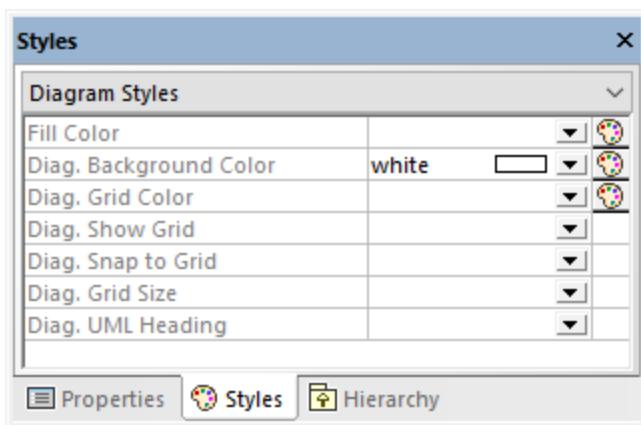
### 5.2.5 Changing the Style of Diagrams

You can change the appearance (style) of a diagram, including the background color, grid visibility, grid size and color, as well as the appearance of the diagram heading. You can either change the style of individual diagrams in the project, or apply the same properties to all diagrams in the project. For information about changing the style of elements inside a diagram, see [Changing the Style of Elements](#)<sup>117</sup>.

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

### To change the appearance of diagrams:

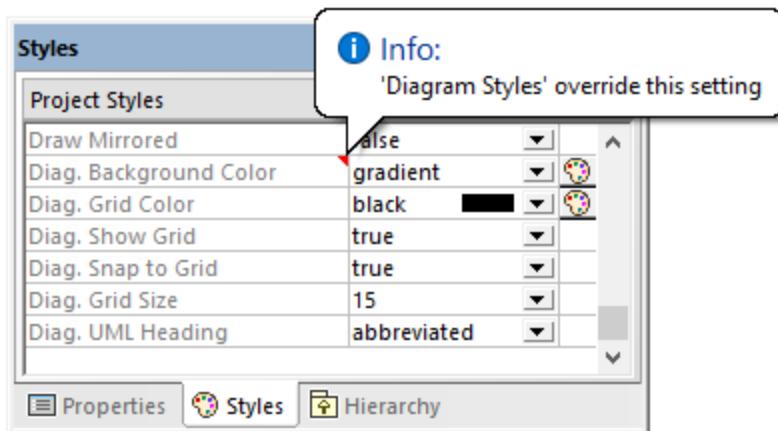
1. Open a diagram (see [Opening Diagrams](#) <sup>122</sup>).
2. Notice the dropdown list at the top of the Style Window and do one of the following as applicable:
  - a. To edit the properties of the current diagram only, select "Diagram Styles" from the list. This value is selected by default if you click anywhere where the diagram background is empty (that is, when you do not click any diagram elements).



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

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

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



#### Overridden diagram style

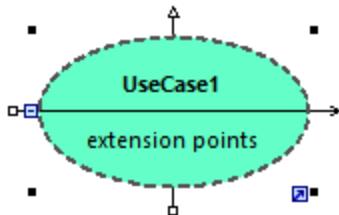
The following diagram-specific properties are available as toolbar buttons. Changing the property in the Styles window will update the state of the toolbar button, and vice versa.

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

## 5.2.6 Aligning and Resizing Modeling Elements

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

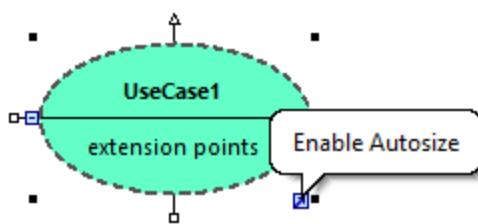
1. Click an element on the diagram. A set of black dots appear at the element's edges.



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

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

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



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

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



*Layout toolbar*

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

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

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

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

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

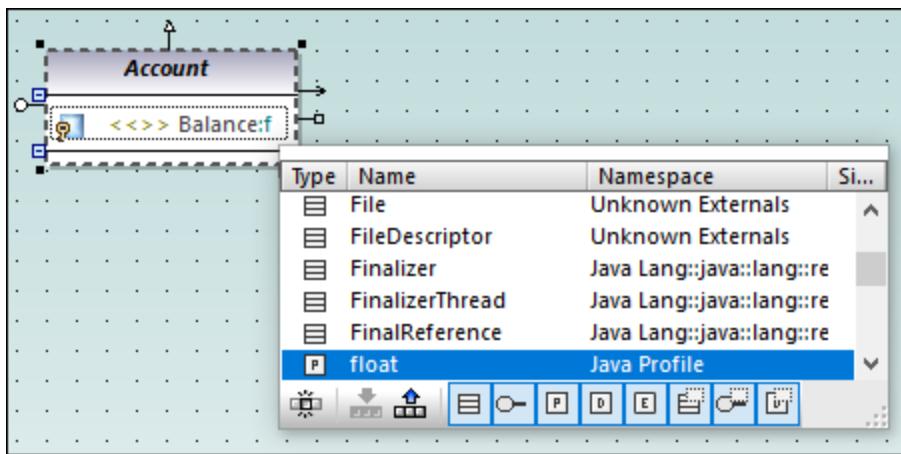
- On the **Layout** menu, click **Autolayout All** and choose one of the following options: **Force Directed**, **Hierarchic**, or **Block**.

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

## 5.2.7 Type Autocompletion in Classes

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

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



*Autocompletion window*

The autocomplete window has the following features:

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

#### To enable only one of the filters at a time:

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

#### To select or clear all filters simultaneously:

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

#### To disable autocomplete:

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

#### To trigger autocomplete on demand (when it is disabled):

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

## 5.2.8 Zooming into/out of Diagrams

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

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



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

To fit the diagram area to the visible window:

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

## 5.3 Relationships

### 5.3.1 Creating Relationships

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

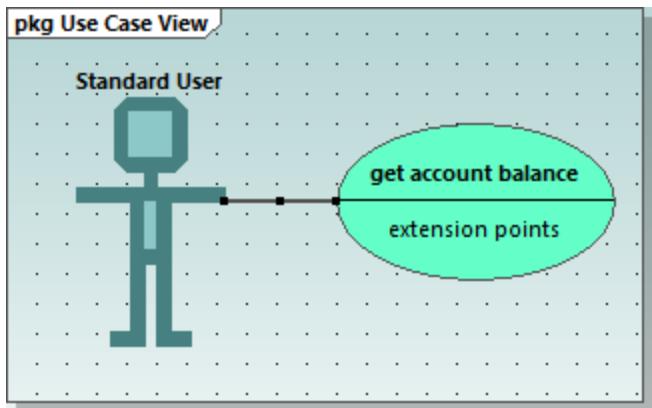
1. By using a toolbar button that depicts the relationship you need (for example, Association ).
2. By using handles that appear when you click on any element on the diagram.

#### Creating relationships using toolbar buttons

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

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

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



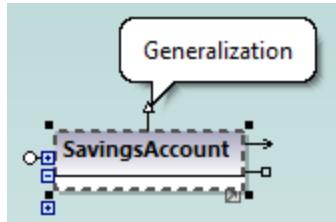
Association in a Use Case diagram

#### Creating relationships using handles

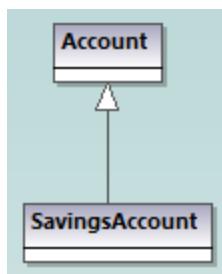
When you click an element on a diagram, several handles may appear to the left, right, top, or bottom of the element. The handles appear only for elements which support relationships. Each handle corresponds to a relationship kind. For example, class elements have the following handles:

- InterfaceRealization
- Generalization
- Association
- Collection Association

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



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



*Generalization relationship between two classes*

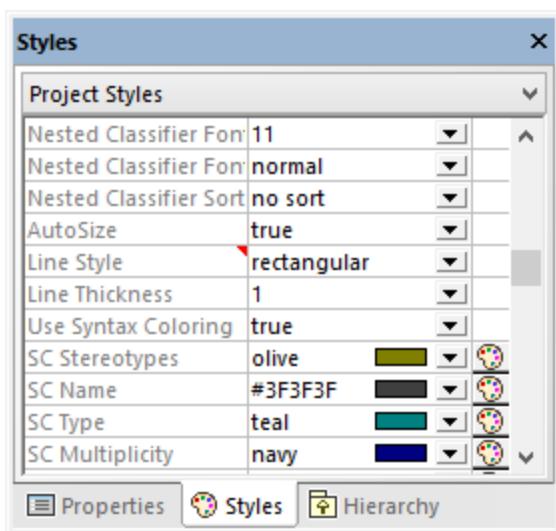
### 5.3.2 Changing the Style of Lines and Relationships

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

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

#### To change line properties:

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



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

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

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

#### To add label text to a relationship:

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

#### To move the label text:

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

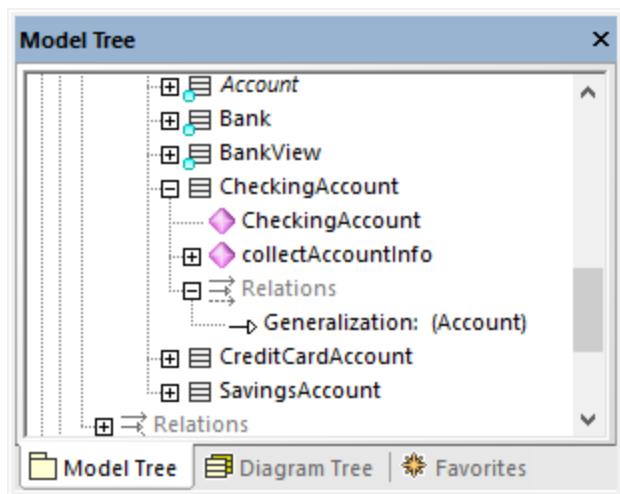
- To reposition multiple labels simultaneously, select one or more relationships on the diagram, and then run the menu command **Layout | Reposition Text Labels**.

#### To show or hide the label text:

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

### 5.3.3 Viewing Element Relationships

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



*Relationship in the Model Tree window*

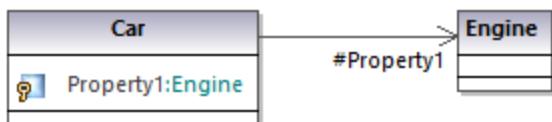
**Note:** To hide relationships from the Model Tree window, right-click inside the window and clear the **Show Relations in Tree** option.

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

### 5.3.4 Associations

An association is a conceptual connection between two elements. You can create association relationships like any other relationship in UModel, see [Creating Relationships](#) 130.

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



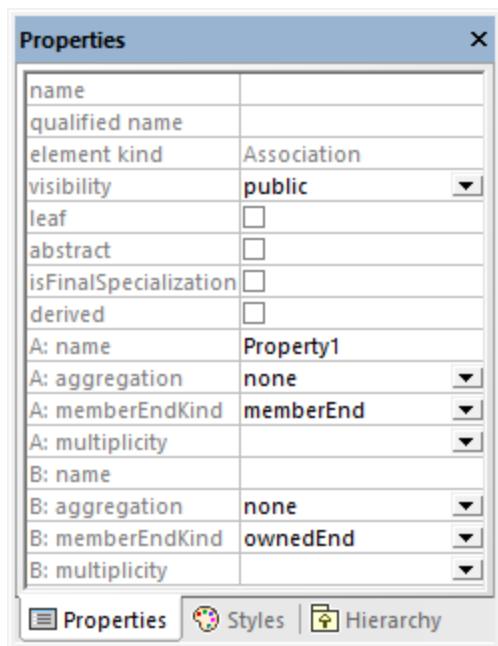
When a class is added to a diagram, its associations are shown automatically on the diagram, provided that the following conditions are met:

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

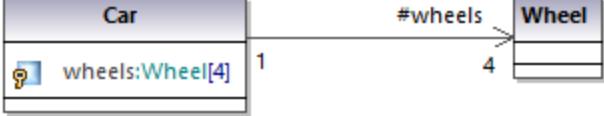
You can also explicitly show the class properties of any class as associations on the diagram. To do this, right-click a class property, and select one of the following commands:

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

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

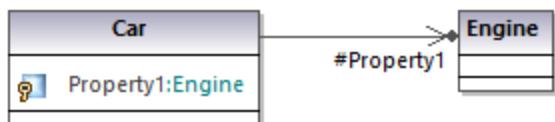


It is important to note the properties listed below. Modifying these properties changes the appearance of the association on the diagram, or adds various informative text labels. For information about showing or hiding text labels, or changing the appearance of the relationship (such as color or line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

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

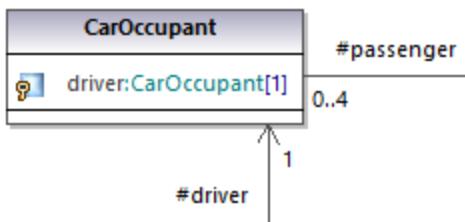
The same set of attributes are available for end B of the relationship.

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



## Creating reflexive associations

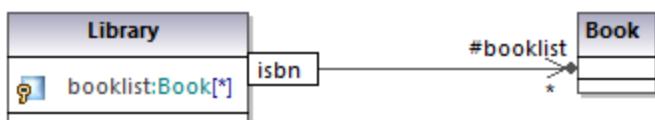
Associations can be created using the same class as both the source and target. This is a so-called "self link", or reflexive association. It may describe, for example, the ability of an object to send a message to itself, for recursive calls. To create a self link, click the association toolbar button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



## Creating association qualifiers

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

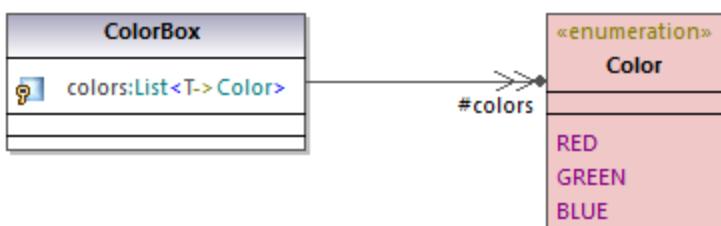
1. Create an association between two classes.
2. Right-click the association and select **New | Qualifier**.



To rename or delete association qualifiers, use the same steps as for all other elements, see [Renaming, Moving, and Copying Elements](#)<sup>107</sup> and [Deleting Elements](#)<sup>108</sup>.

## 5.3.5 Collection Associations

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



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

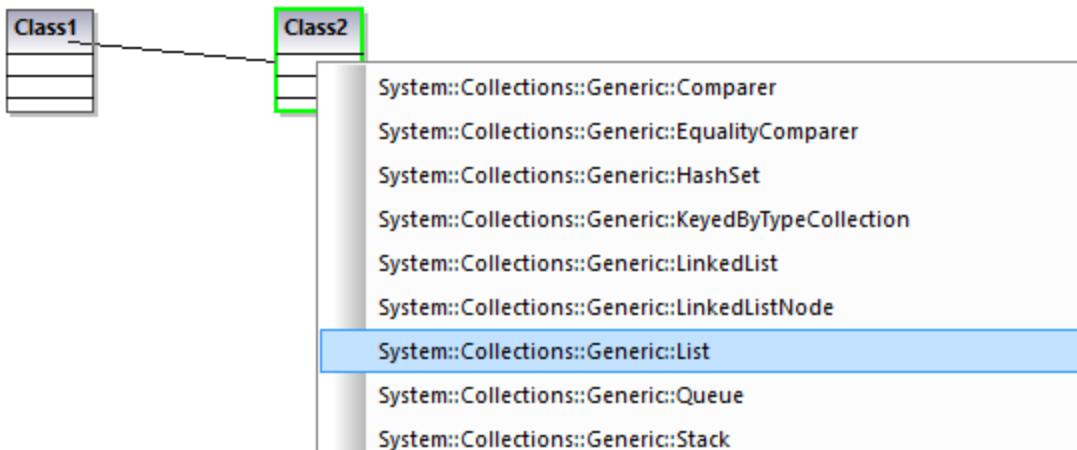


If your project is UML only (without support for a specific code engineering language), you can define collection templates from the menu **Tools | Options | Diagram Editing | Collection Templates | UML** tab.

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

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

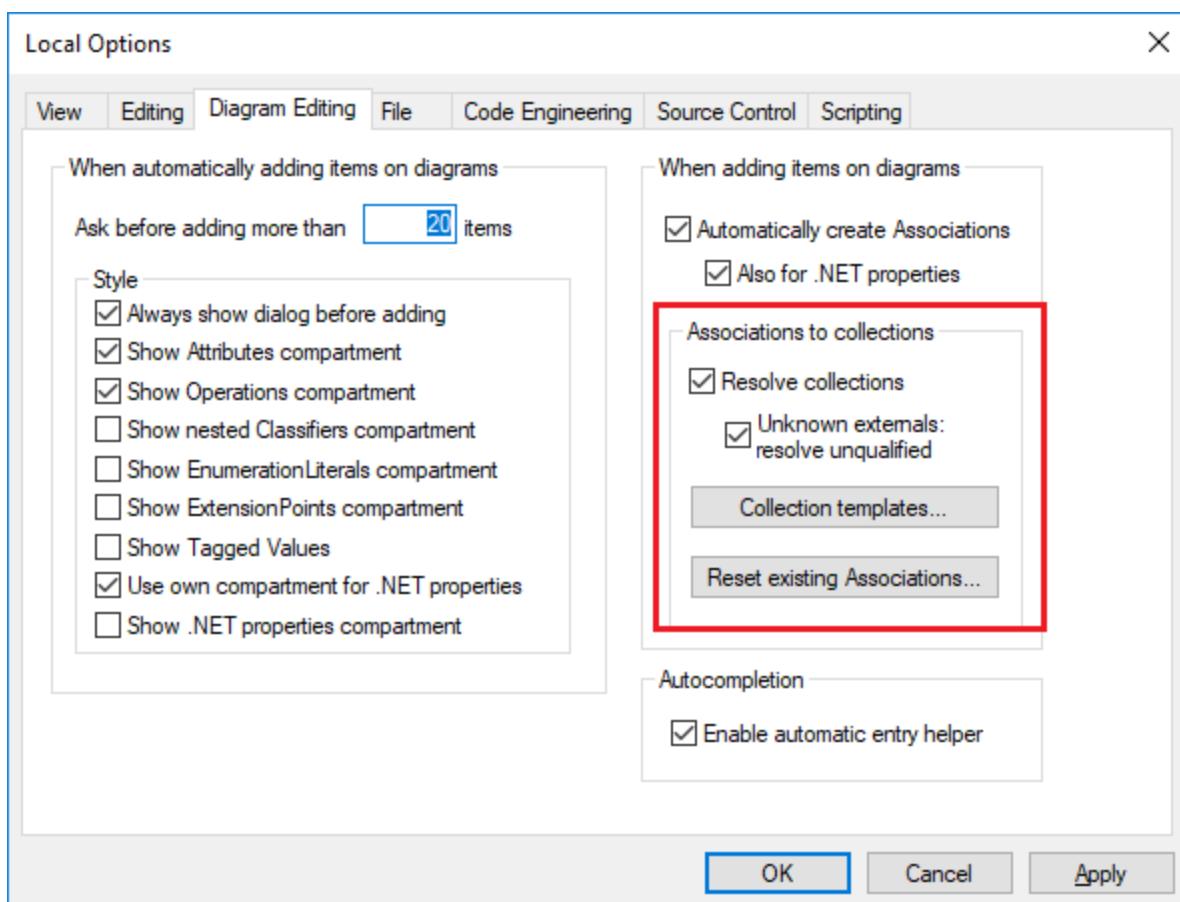
1. Add two classes to the diagram.
2. Click the **Collection Association** toolbar button.
3. Drag from the first class and drop it onto the second class. The collection templates defined for the project appear in the context menu, and you can select the required one.



#### Collection associations and code engineering

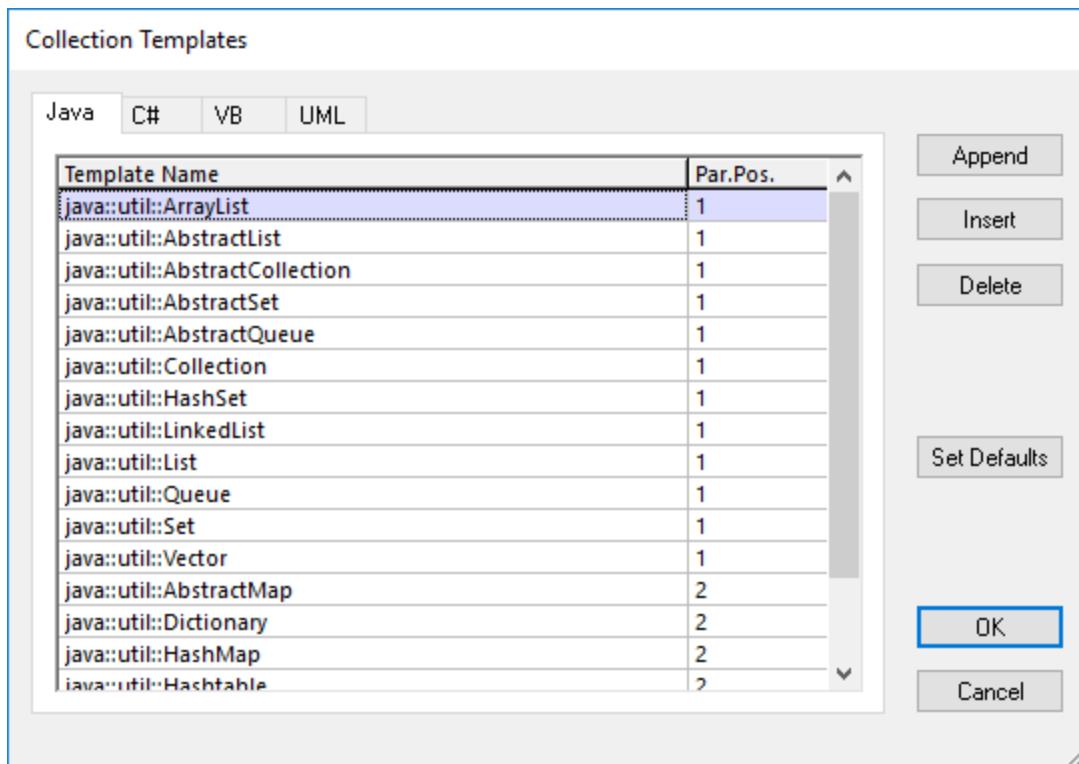
If you import program code into the model, collection associations are created automatically by default, based on predefined collection templates. To enable or disable this option:

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



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

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



Collection Templates dialog box

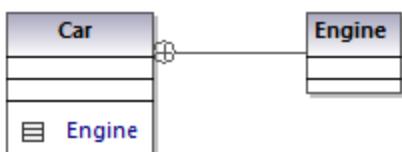
To reset the collection templates to their default values, click **Set default**.

### 5.3.6 Containment

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

To illustrate containment between two classes:

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



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

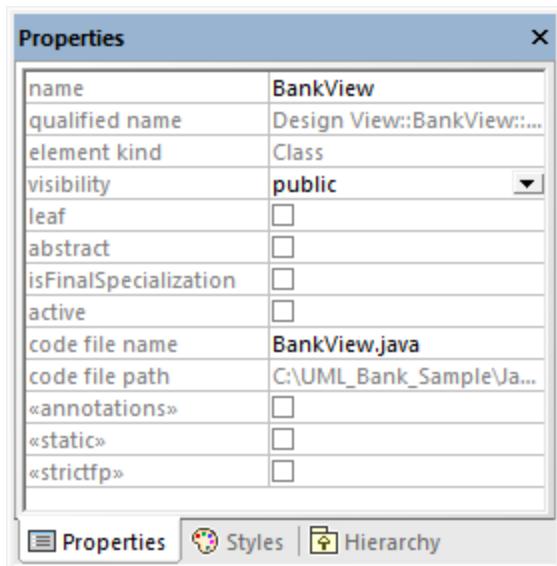
## 5.4 Stereotypes and Tagged Values

A stereotype is an extension mechanism; it is intended as a flexible way to extend an existing UML element and capture some aspect of it that standard UML doesn't. Stereotypes applied to an element signify that that element has some special use. The UModel built-in profiles (C#, Java, VB.NET, and so on) contain all the stereotypes required to model projects in the respective languages. However, you can also create your own profiles (and their respective stereotypes), see [Creating and Applying Custom Profiles](#)<sup>405</sup>.

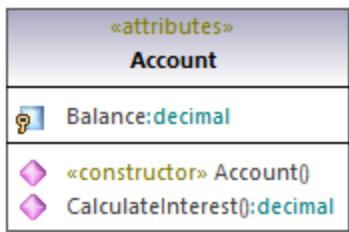
When you import source code or binaries into the model, UModel applies stereotypes to elements automatically, based on the structure of the original code. For example, if annotations modifiers exist in the imported Java source code, the corresponding elements in the model get the «annotations» stereotype. For information about how various language constructs map to UModel elements and become stereotypes in the model, see [UModel Element Mappings](#)<sup>219</sup>.

You can also apply stereotypes to elements manually, while modeling them. For example, you can apply the «attributes» stereotype to a C# class, which would indicate that the class must be decorated with attributes in generated code. To specify the attribute values in the generated code, you can add so-called "tagged values" in UModel, as shown in [Applying Stereotypes](#)<sup>142</sup>. Stereotypes are also used extensively in XML schema modeling, to model elements such as simple types, complex types, facets, and so on.

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



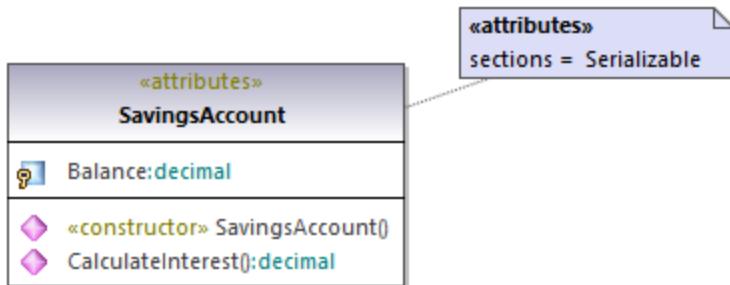
In class diagrams, stereotypes are visible above the name of the class. For example, the class below has the «attributes» stereotype applied to it.



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

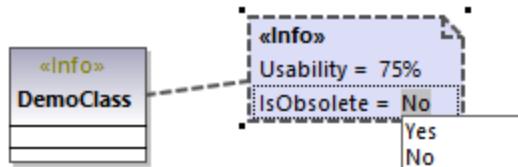
### 5.4.1 Tagged Values

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



*Tagged values*

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



You can change how tagged values are displayed on the diagram, or hide them altogether, see [Showing or Hiding Tagged Values](#)<sup>144</sup>. For information about changing a stereotype's tagged values, see [Applying Stereotypes](#)<sup>142</sup>. For an example that illustrates how to create stereotypes with tagged values, see [Example: Creating and Applying Stereotypes](#)<sup>409</sup>.

## 5.4.2 Applying Stereotypes

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

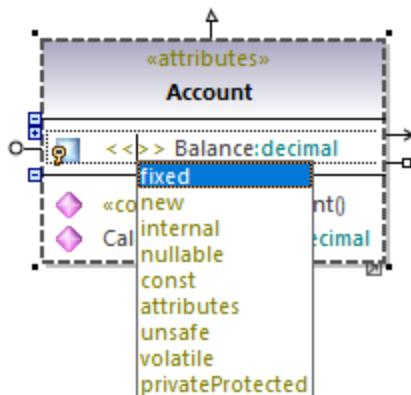
Before you can apply stereotypes, the corresponding profile must be applied to your package(s) first. This is done automatically by UModel if you right-click a package and select the **Code Engineering | Set as {language} namespace root** command. For more information, see [Applying UModel Profiles](#)<sup>154</sup>.

If you created custom profiles, these must be applied manually to the package, see [Creating and Applying Custom Profiles](#)<sup>405</sup>.

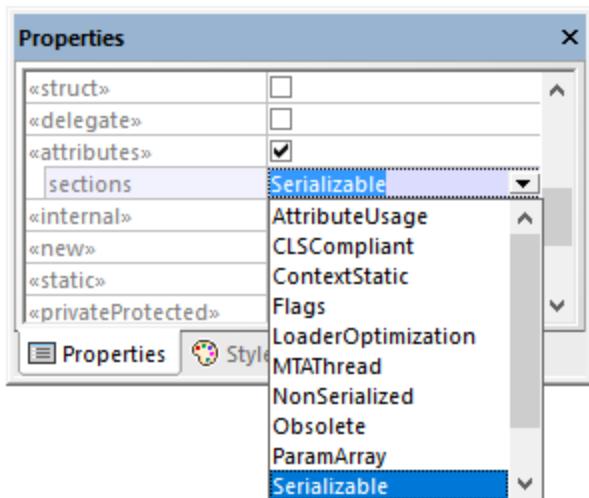
### To apply a stereotype to an element:

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

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

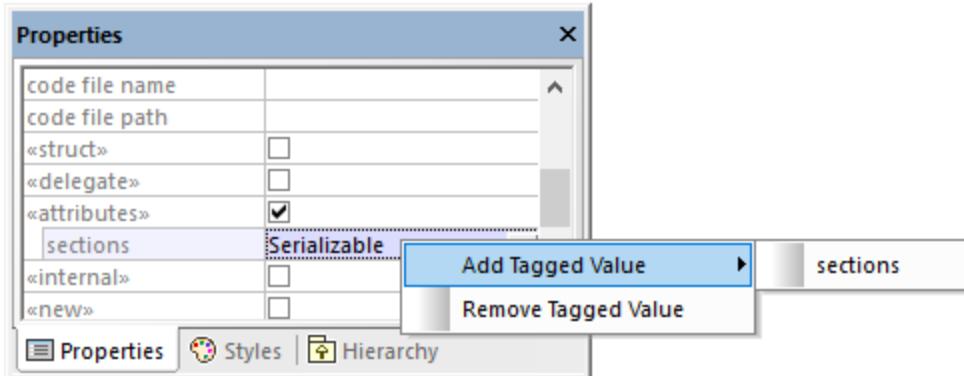


Some stereotypes are associated with a list of name-value pairs referred in UML as "tagged values". To apply a stereotype with tagged values to an element, select the stereotype's check box in the Properties window (in this example, «attributes»). This adds an indented entry where you can select the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

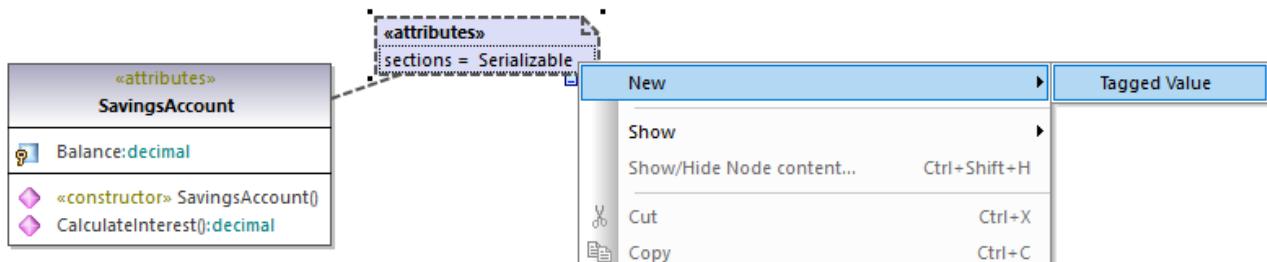


### Tagged values

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

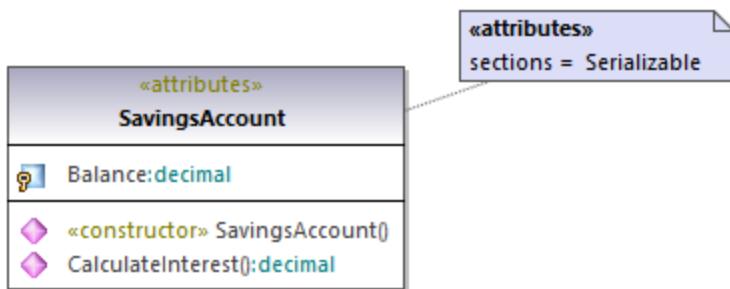


Alternatively, you can add tagged values directly from the diagram, by right-clicking a value, and selecting **New | Tagged Value** from the context menu.



### 5.4.3 Showing or Hiding Tagged Values

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

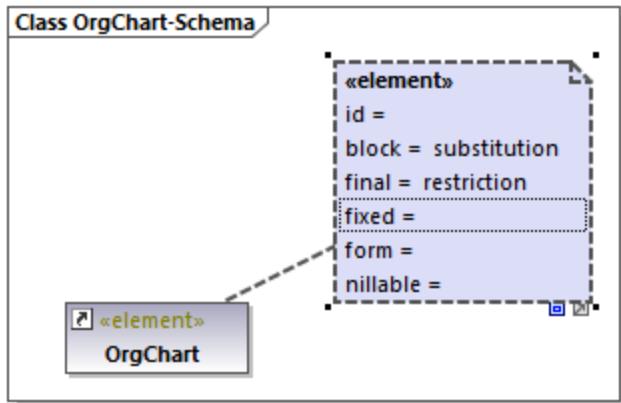


*Tagged values displayed outside a class*

#### Toggle compact mode

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

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



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

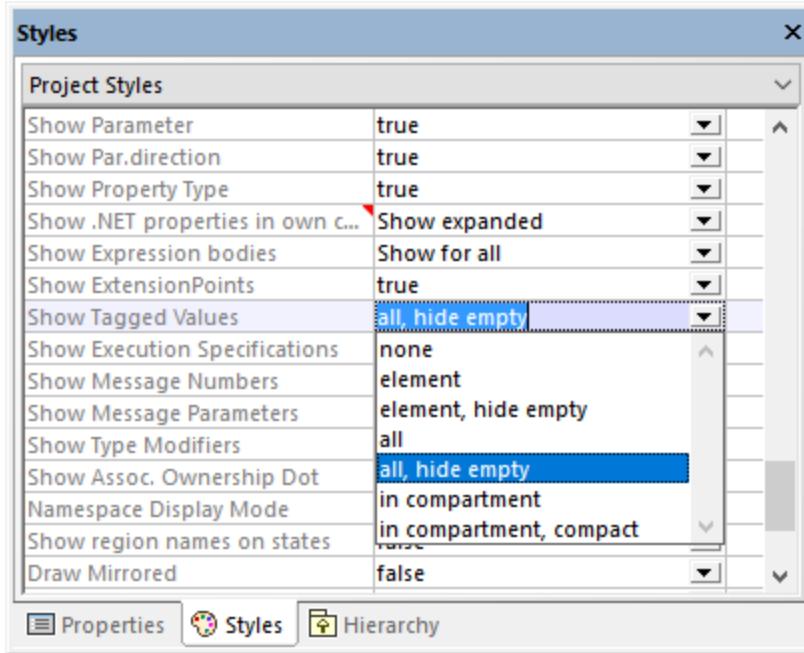
When the handle is in expanded state the empty values are shown as well. When the handle is in collapsed state , the empty values are hidden.

#### Changing the display of tagged values globally

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

### To change tag values at project level:

1. Select **Project Styles** from the list at the top of the [Styles Window](#)<sup>86</sup>.
2. Scroll down until to the **Show Tagged Values** property and select the required option from the list (for example, **all, hide empty**).

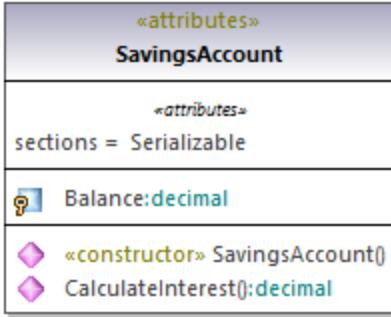


For information about changing styles at various levels, see [Changing the Style of Elements](#)<sup>117</sup>.

### Possible display options

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

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

<i>In compartment</i>	Displays the tagged values in a separate compartment. For example, the class illustrated below has an « <i>attributes</i> » compartment that contains tagged values.
	 <p>The diagram shows a UML class named 'SavingsAccount'. It features a compartment labeled «<i>attributes</i>» containing the following elements:</p> <ul style="list-style-type: none"><li>A note: <code>sections = Serializable</code></li><li>An attribute: <code>Balance:decimal</code> (indicated by a key icon)</li><li>Two operations: <code>SavingsAccount()</code> and <code>CalculateInterest():decimal</code> (indicated by diamond icons)</li></ul>
<i>In compartment, hide empty</i>	Displays only those tagged values where a value exists, in a compartment.
<i>In compartment, compact</i>	Same as above.

## 6 Projects and Code Engineering

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

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

The menu commands applicable to code engineering are available in the **Project** menu. For example, the menu command **Project | Import Source Project** enables you to import C#, or VB.NET Visual Studio solutions, or Java code, and generate UModel diagrams based on it. When no project solution is available, use the menu command **Project | Import Source Directory**, see [Importing Source Code \(Reverse Engineering\)](#)<sup>187</sup>. Java, C#, and VB.NET binaries can also be imported, provided that a few basic prerequisites are met, see [Importing Java, C# and VB.NET Binaries](#)<sup>188</sup>.

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

For the list of mappings between UModel elements and elements in each supported language profile (including databases and XML Schema), see [UModel Element Mappings](#)<sup>219</sup>.

## 6.1 Managing UModel Projects

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

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

### 6.1.1 Creating, Opening, and Saving Projects

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

**Note:** UModel includes several example projects that you can explore in order to learn the modeling basics and the graphical user interface. These can be found at the following path: **C:\Users\<username>\Documents\Altova\UModel2022\UModelExamples**.

#### To create a new project:

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

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

- Root
- Component View

These two packages have special use and are the only ones that cannot be renamed, or deleted, as explained in the tutorial, see [Forward Engineering \(from Model to Code\)](#)<sup>60</sup>.

Once the project is created, you can add modeling elements to it, such as UML packages and diagrams, see [Creating Elements](#)<sup>104</sup> and [Creating Diagrams](#)<sup>119</sup>.

#### To add a new package:

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

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

#### To add a new diagram:

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

**To add elements to a diagram:**

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

For a worked example of how to create a project and generate program code from it, see [Forward Engineering \(from Model to Code\)](#)<sup>60</sup>.

**To open an existing project:**

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

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

**To save a project:**

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

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

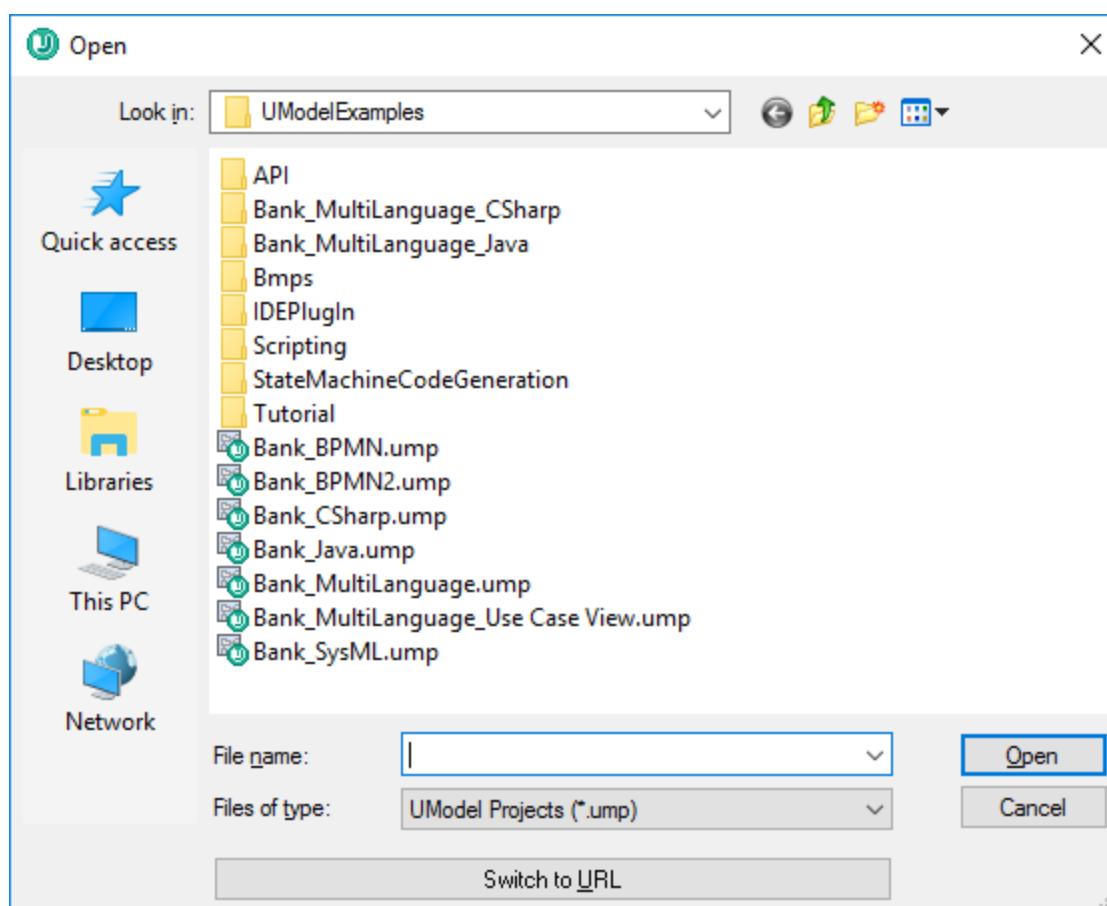
Note: The \*.ump file is an XML file format which can be optionally "prettified" on saving. Pretty-printing can be enabled from the **Tools | Options | File** tab.

## 6.1.2 Opening Projects from a URL

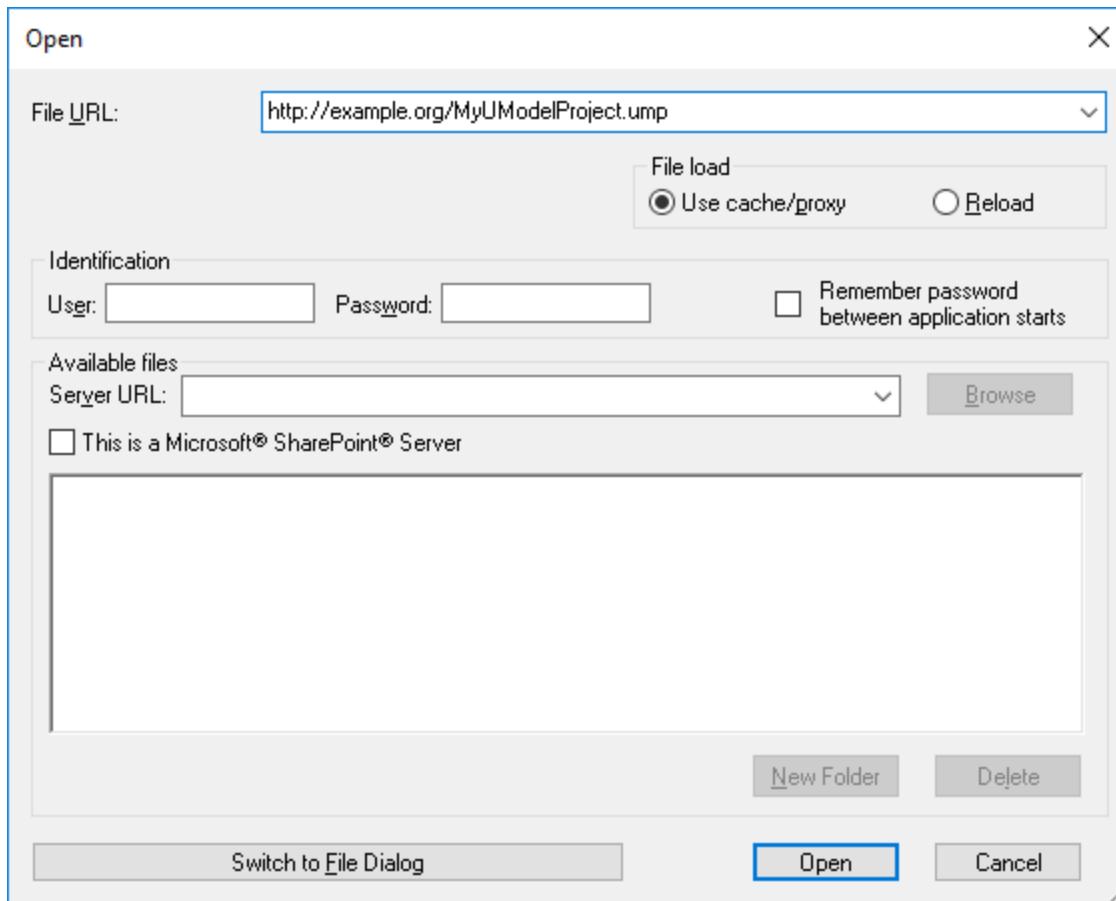
In addition to opening local project files, you can also open files from a URL. The supported protocols are HTTP, HTTPS, and FTP. Note that files loaded from URLs cannot be saved back to their original location (in other words, access to the file is read-only), unless they are checked out from a Microsoft®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

**To open a file from a URL:**

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



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



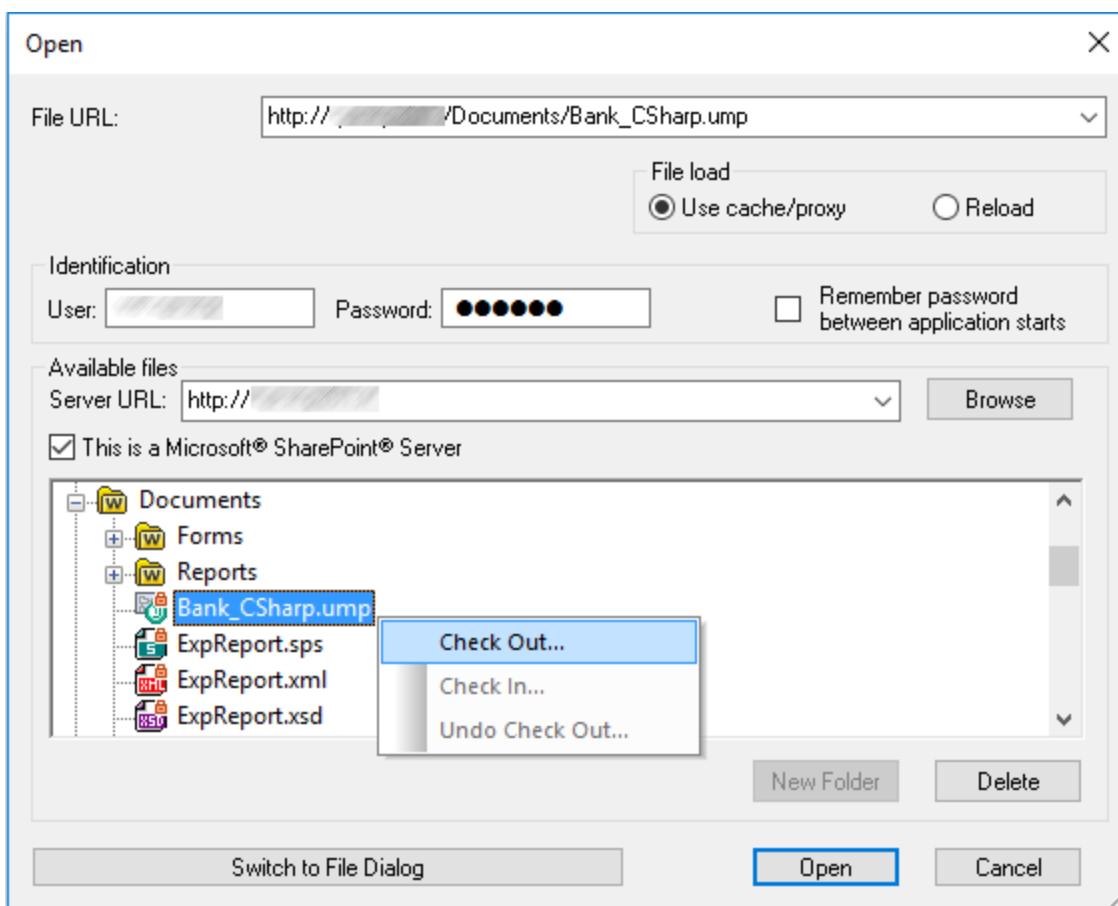
If the server requires password authentication, you will be prompted to enter the user name and password. If you want the user name and password to be remembered next time you start UModel, enter them in the Open dialog box and select the **Remember password between application starts** check box.

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

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

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

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



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

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

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

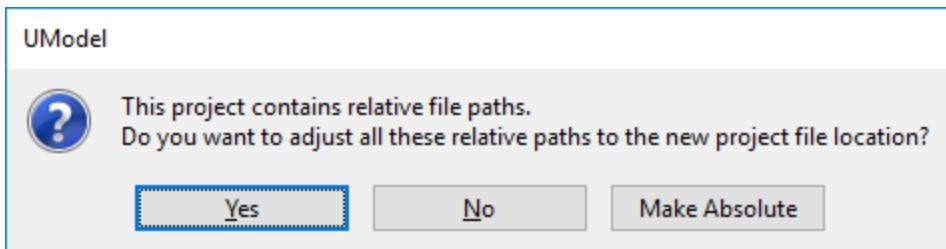
Note the following:

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

### 6.1.3 Moving Projects to a New Directory

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

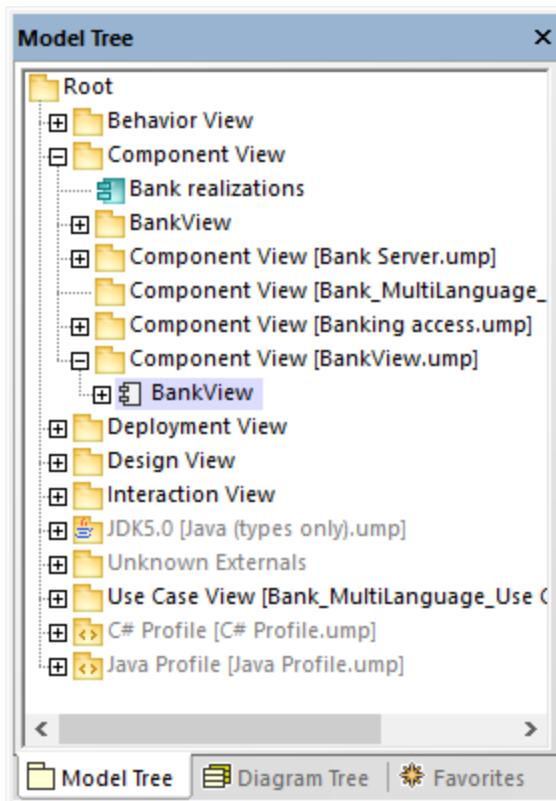
- Select the menu option **File | Save As...**, and click **Yes** when prompted to adjust the file paths to the new project location.



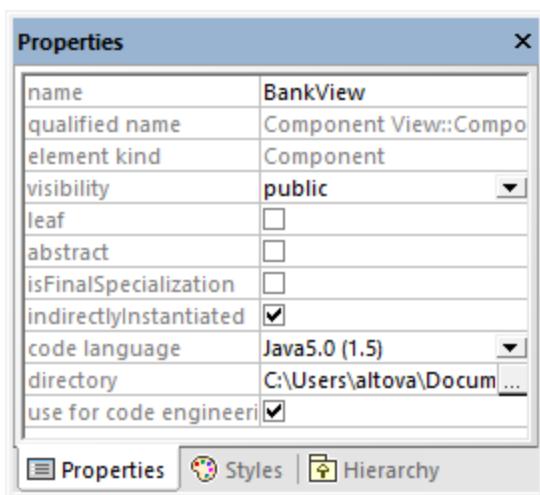
- Copy the UModel project (\*.ump) to a new location, and then adjust the code generation paths for each component involved in code generation.

For an example of the second approach, open the following sample project: C:  
`\Users\<username>\Documents\Altova\UModel2022\UModel\Examples\Bank_Multilanguage.ump`.

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



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



3. Re-synchronize the model and code.

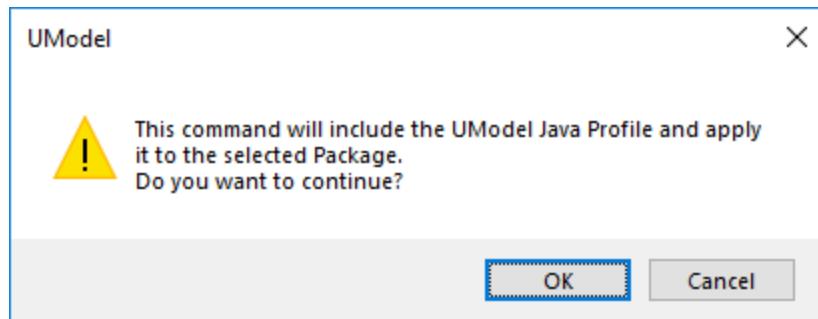
#### 6.1.4 Applying UModel Profiles

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

One must distinguish between two types of profiles:

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

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



To view the full list of UModel built-in profiles or add them to your model manually, select the menu command **Project | Include Subproject**. See also [Including Subprojects](#)<sup>158</sup>.

For instructions about creating custom profiles in order to extend or adapt UML, see [Creating and Applying Custom Profiles](#).

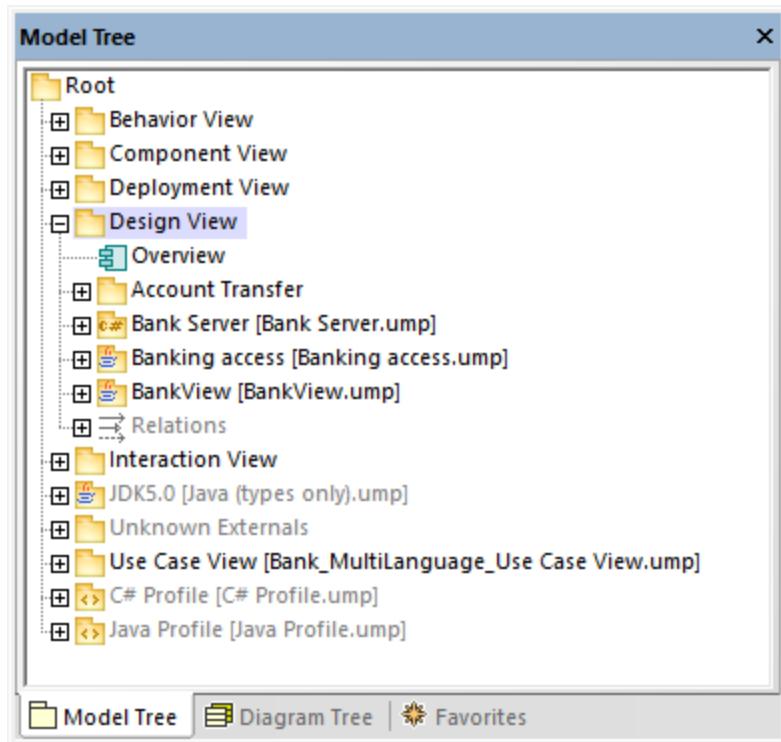
## 6.1.5 Splitting UModel Projects

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

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

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

In the [Model Tree Window](#), subprojects appear with the respective .ump file name displayed to the right, enclosed within square brackets. For example, the project illustrated below includes several subprojects (this is the **Bank\_MultiLanguage.ump** from the C:\Users\<username>\Documents\Altova\UModel\2022\UModel\Examples directory).

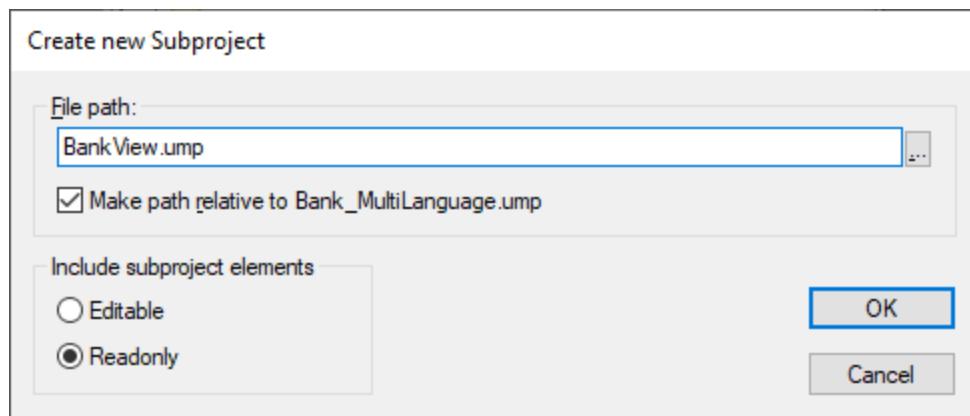


During the code-engineering process, all subordinate components of a subproject are considered. There is no difference between a single project file or one that consists of multiple editable subprojects. This also applies to UML diagrams—they can also be edited at the main, or subproject, level.

**Note:** You can also share packages and UML diagrams they might contain, between different projects. For more information, see [Sharing Packages and Diagrams](#).

## Creating subprojects

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



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

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

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

## Opening and editing subprojects

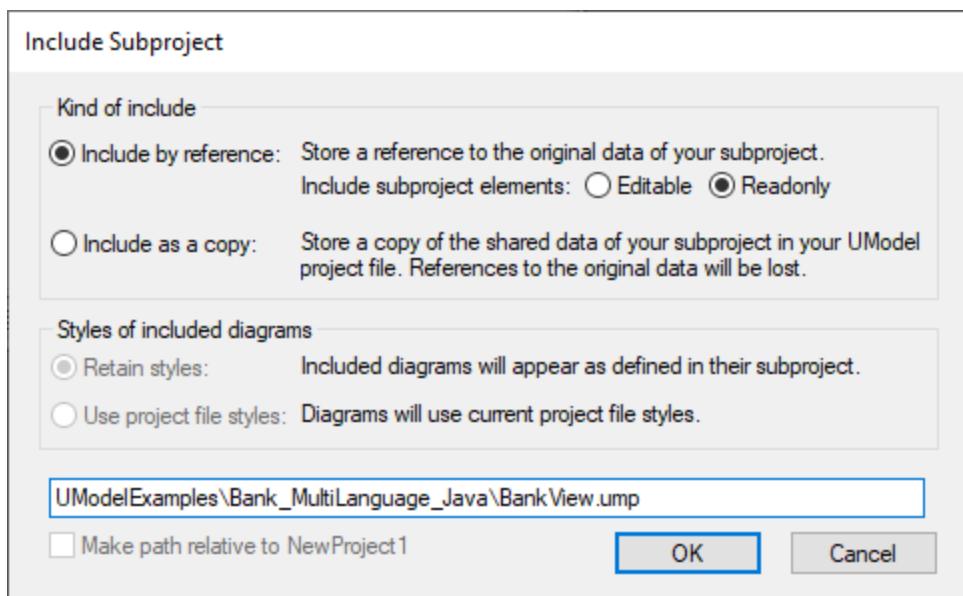
You can open a subproject as a standalone UModel project, directly from the main project. For this to be successful, there should not be any unresolved references to other elements. UModel automatically performs checks when creating a subproject from the "main" project, and whenever a file is saved.

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

## Reusing subprojects

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

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



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

## Saving projects

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

## Saving subproject files

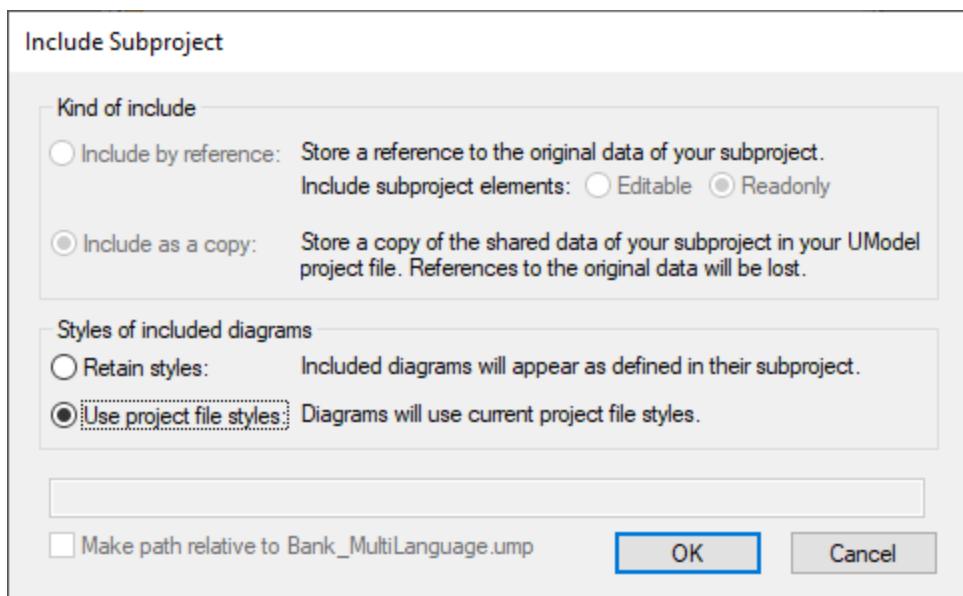
When saving subprojects (from the main project level), all references to sibling, as well as child subprojects, are considered and saved. For example, if two sibling subprojects, "sub1" and "sub2", exist and "sub1" uses elements from "sub2", then "sub1" is saved in such a way that it automatically saves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

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

## Reintegrating subprojects into the main project

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

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



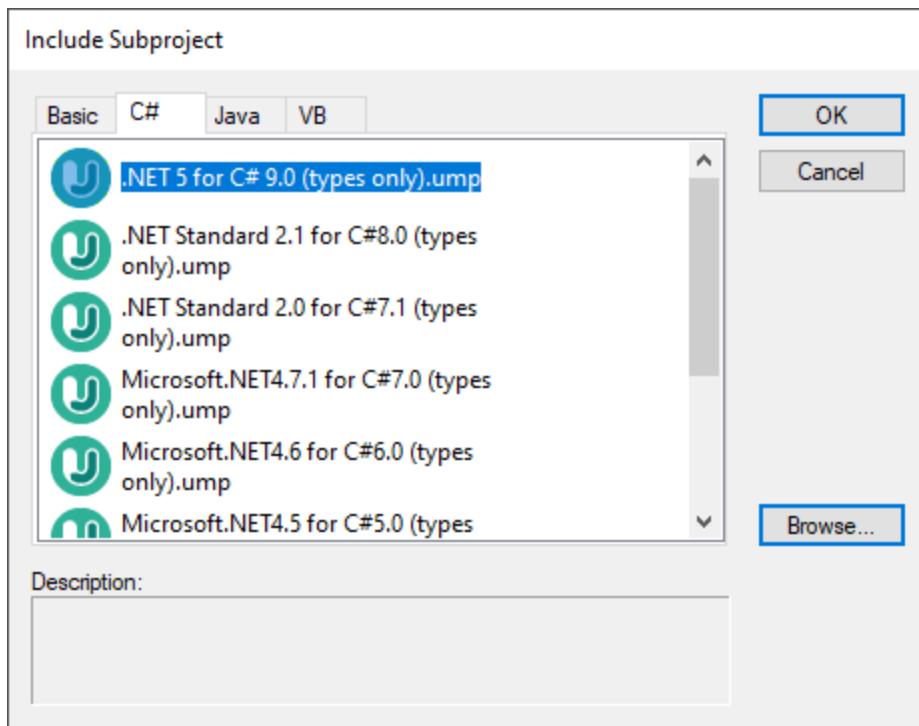
2. Select the style option that you want to use, and then click **OK**.

### 6.1.6 Including Subprojects

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

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

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

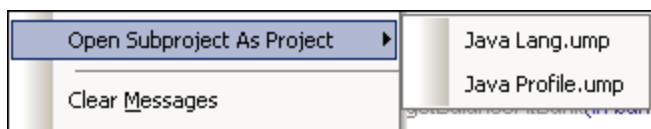


*Include Subproject dialog box*

The tabs and UModel projects (.ump files) available on the "Include Subproject" dialog box are configurable. Namely, UModel reads this information from the following path relative to the "Program Files" folder on your operating system: **\Altova\UModel2022\UModel\Include**. Note that the project files available on the **Basic** tab exist directly under the **UModel\Include** folder, while projects in each of the Java, VB, and C# tabs exist as subfolders of the **UModel\Include** folder.

#### To view all currently imported projects:

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



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

- Navigate to the **\Altova\UModel2022\UModel\Include** folder (relative to your "Program Files"), and create your custom folder in it, for example **\UModel\Include\myfolder**. The name you give to the folder determines the name of the tab on the "Include Subproject" dialog box.
- Copy to your custom folder any .ump files that you want to make available on the corresponding tab.

**To create descriptive text for each UModel project file:**

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

**To remove an included project:**

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

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

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

## 6.1.7 Sharing Packages and Diagrams

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

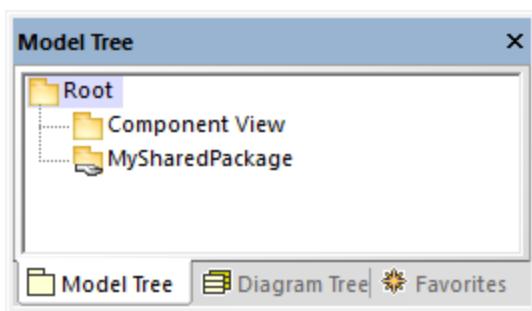
Also note that subproject files can be split off a main, or subproject, file at any time. The subproject files can be included as editable or read-only from the main project; each package is shared and saved as a subproject file. Subprojects can be added to a source control system, see [Teamwork support for UModel projects](#)<sup>155</sup>.

**Notes**

- In order to be shareable, a package must not contain links to external elements (elements outside of the shared scope).
- When creating UModel project files, do not use one project file as a "template/copy" for another project file into which you intend to share a package. This will cause conflicts due to the fact that every element should be globally unique (see [uuid](#)<sup>436</sup>) and this will not be the case, as two projects will have elements that have identical uuids.

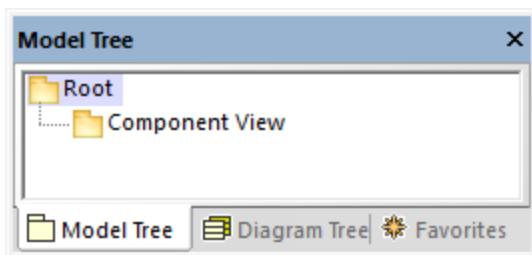
**To share a package between projects:**

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

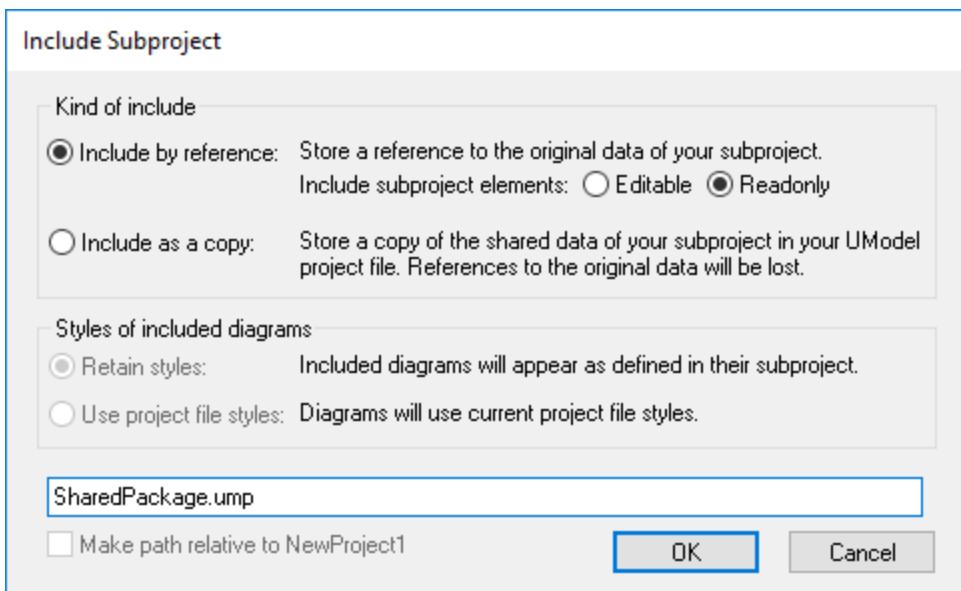


### To include/import a shared folder in a project:

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

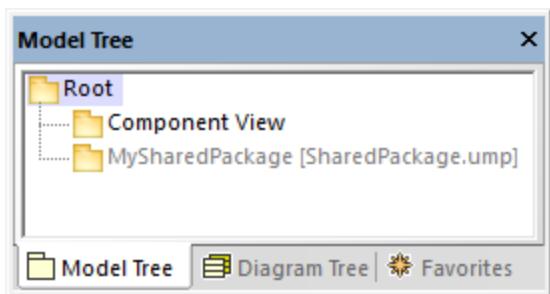


2. Select the menu item **Project | Include Subproject...**
3. Click **Browse**, select the project that contains the shared package, and click **Open**. The "Include Subproject" dialog box allows you to choose between including the package/project by reference, or as a copy.



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

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



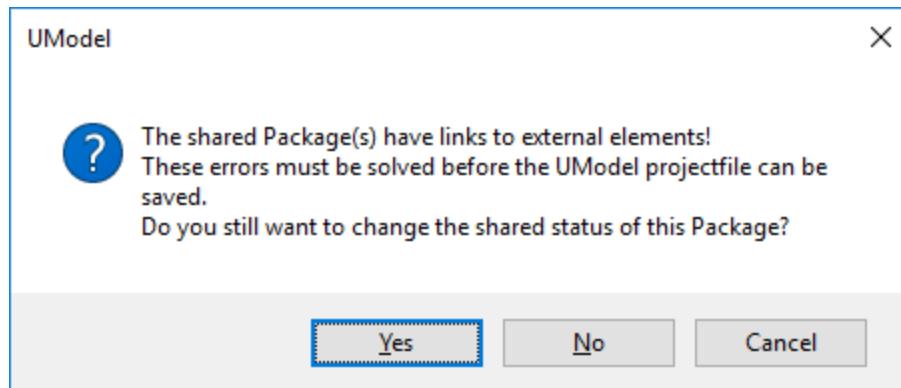
Notes:

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

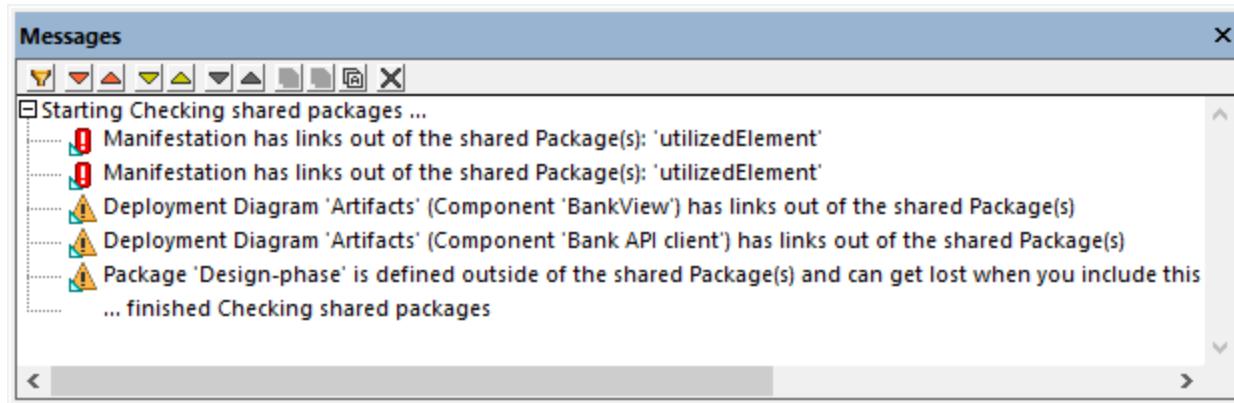
### Resolving links to external elements

Attempting to share a package which has links to external elements causes a warning dialog box to appear. For example, the following message appears if you attempt to share the "Deployment View" package of the sample project C:

\Users\<username>\Documents\Altova\UModel2022\UModel\Examples\Tutorial\BankView-start.ump.



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



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

## 6.1.8 Tips for Enhancing Performance

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

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

## 6.2 Generating Program Code

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

Code generation is also applicable to XML schemas and databases\*. For example, you could design an XML schema or a database with UModel and then generate the corresponding file (or SQL script, in case of databases) from the model. To achieve this, consult the mapping tables to find out which schema or database elements map to UModel elements, see [UModel Element Mappings](#)<sup>219</sup>.

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

### Prerequisites

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

- One of the packages in your project must be designated as namespace root. The namespace root can be a C#, Java, VB.NET, XSD, or Database namespace. This package must contain all classes and interfaces from which code is to be generated. For more information, see [Setting a Package as Namespace Root](#)<sup>164</sup>.
- A code engineering component must be added to the project. This component must be realized by all the classes or interfaces from which code is to be generated. For more information, see [Adding a Code Engineering Component](#)<sup>165</sup>.

In addition to this, it is recommended that you include one of the built-in UModel subprojects corresponding to the language (or the language version) you want to use, see [Including Subprojects](#)<sup>158</sup>. For example, if your application must target a specific version of C#, Java, or VB.NET, this would enable you to use the corresponding data types while designing your UML classes, interfaces, and so on.

For a worked example of how to create a project from scratch and generate code from it, see [Example: Generate Java Code](#)<sup>176</sup>.

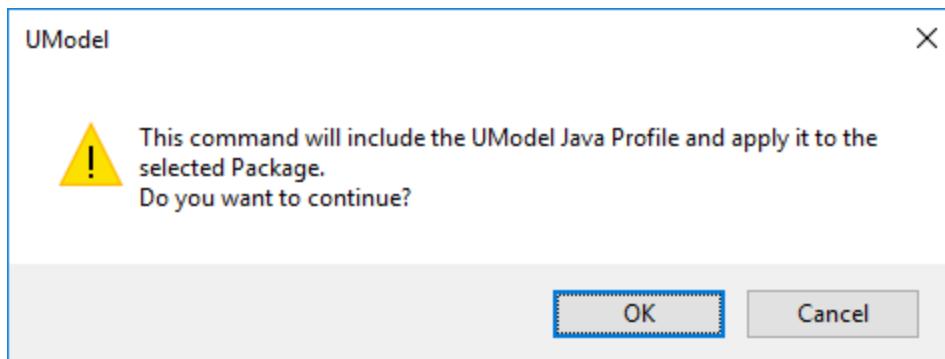
### 6.2.1 Setting a Package as Namespace Root

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

#### To set a package as namespace root:

- Right-click a package in the [Model Tree Window](#)<sup>79</sup> and select **Code Engineering | Set as <...> Namespace Root** from the context menu, where <...> is one of the following: C#, Java, VB.NET, XSD.

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



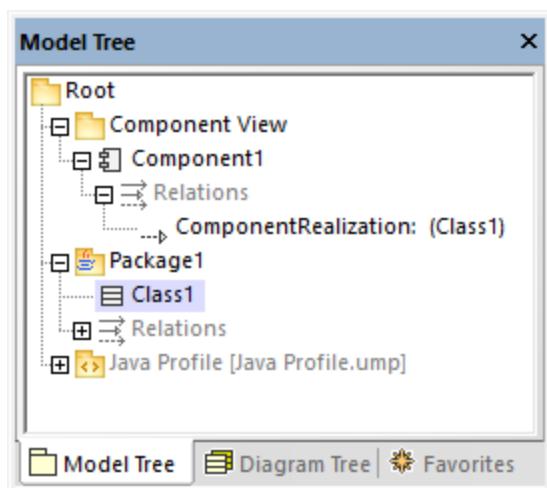
## 6.2.2 Adding a Code Engineering Component

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

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

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

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

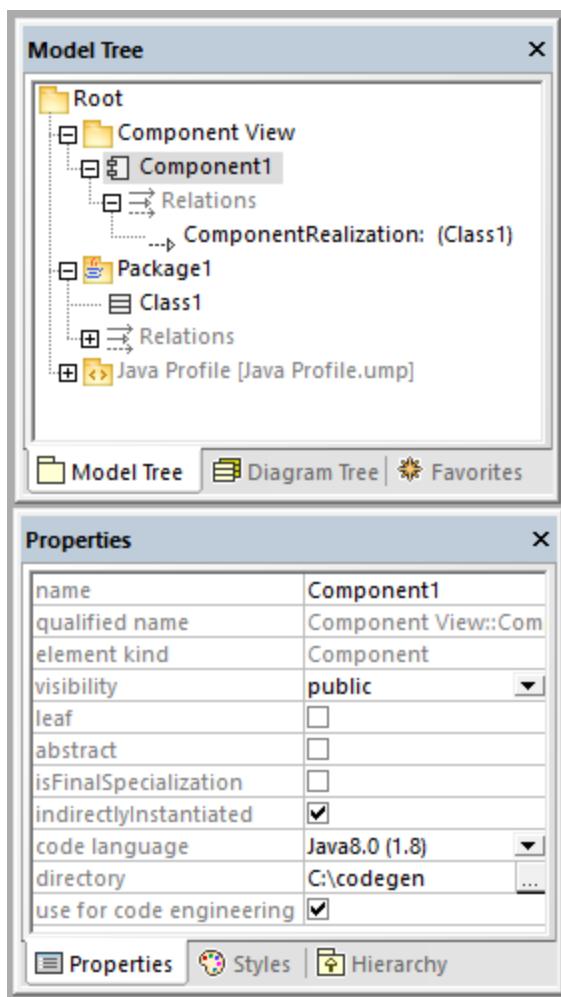


There is also an alternative approach to do this, by creating a Component diagram and then drawing a `ComponentRealization` relation between the component and the classes or interfaces. For more information, see [Component Diagrams](#).

#### To prepare a component for code engineering:

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

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



### 6.2.3 Checking Project Syntax

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

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

- For information about solving common syntax errors, see the [Code generation prerequisites](#) 164.
- For components, the checks below are performed only if the **use for code engineering** property is enabled for the component in the Properties window.
- For classes, interfaces, and enumerations, the checks below are performed only if the class, interface, enumeration is contained in a code language namespace. In other words, it must be under a package which has been defined as namespace root.

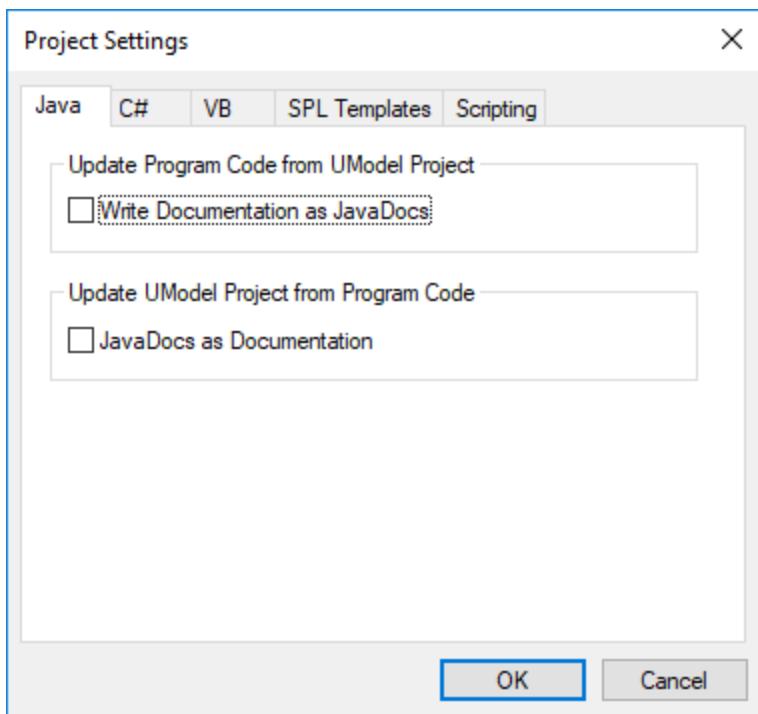
- Constraints on model elements are not checked, as they are not part of the code generation process, see [Constraining Elements](#)<sup>112</sup>.

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

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

## 6.2.4 Code Generation Options

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



The options are grouped into tabs as follows.

Tab	Options
Java	Select the check box <b>Write Documentation as JavaDocs</b> to convert the documentation of UModel elements to equivalent JavaDocs-style documentation in generated code.
C#	Select the check box <b>Write Documentation as DocComments</b> to convert the documentation of UModel elements to comments in generated C# code.
VB	Select the check box <b>Write Documentation as DocComments</b> to convert the documentation of UModel elements to comments in generated VB.NET code.
SPL Templates	If you want to force UModel to read SPL templates from a custom path other than the default one, the custom path must be entered here. See also <a href="#">SPL Templates</a> .

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

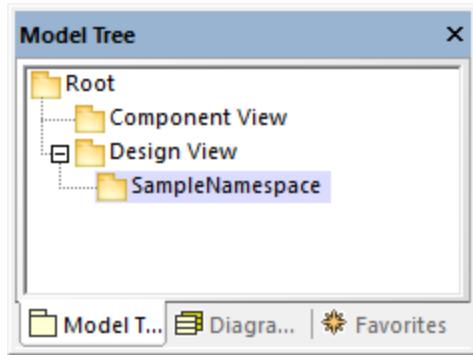
## 6.2.5 Example: Generate C# Code

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

In this example, the target platform is **.NET Standard 2.0 for C# 7.1**. As shown in the instructions below, this is possible thanks to a profile built into UModel that defines all the types of the **.NET Standard 2.0 for C# 7.1**. UModel also includes built-in profiles for specific .NET Framework versions should you need them, see also [Including Subprojects](#)<sup>158</sup>.

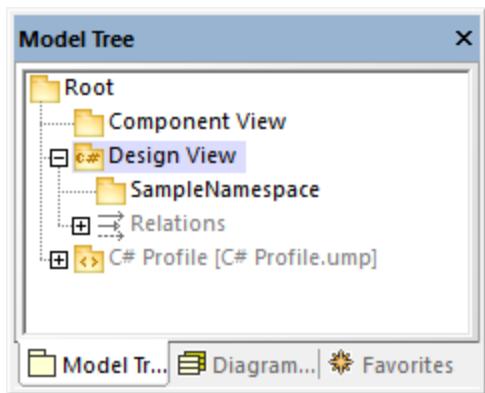
### Create a new UModel project and its structure

On the **File** menu, click **New**. This creates an empty project with two default packages ("Root" and "Component View"). Next, right-click the "Root" package, and create a few more packages, as illustrated below. (If you are completely new to the UModel graphical user interface, see the [UModel Tutorial](#)<sup>14</sup> and [How to Model...](#)<sup>103</sup> chapters to get started.)



In this example, the "Design View" package acts as a container for whatever is going to be the design part of your model (classes and class diagrams, for example), while the "SampleNamespace" package will act as a namespace for all classes that are to be created. In general, however, the package structure is not prescriptive in any way; you may organize your packages in a different way if so required.

Right-click the "Design View" package and select **Code Engineering | Set as C# Namespace Root** from the context menu. When prompted by UModel that the C# profile will be applied to the package, click **OK** to confirm. The C# profile built into UModel is now included to the project.

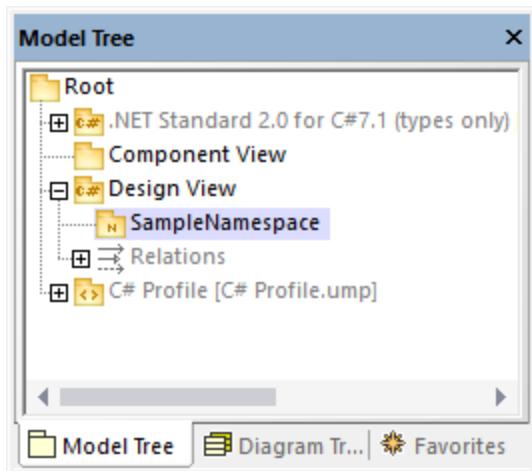


Next, click the "SampleNamespace" package and select the <<namespace>> check box in the Properties window. This applies the "namespace" stereotype to the package and its icon changes to . You can now create classes under this namespace.

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

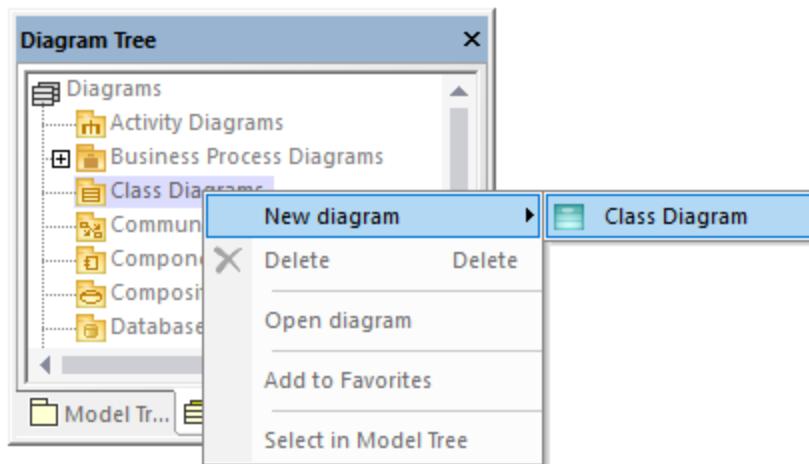
1. On the **Project** menu, click **Include Subproject**.
2. On the **C#** tab, select **.NET Standard 2.0 for C# 7.1 (types only)**.
3. Click **OK**.
4. When prompted to select the include kind, select **Include by reference**.

The additional profile is now added to the project.



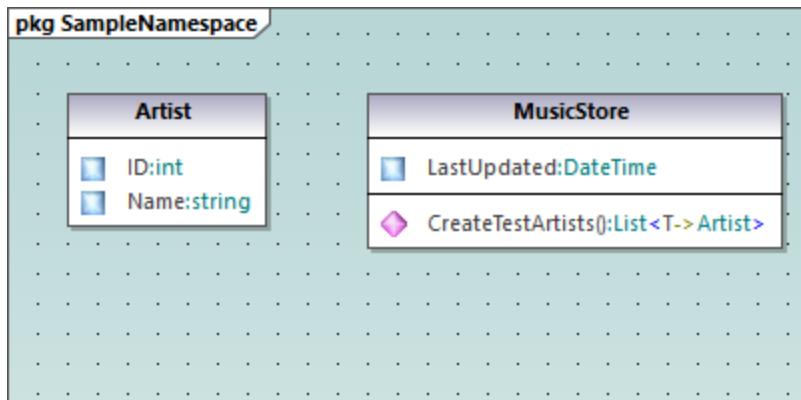
## Create C# classes

You can either create classes directly from the Model Tree window, or from a class diagram. For the scope of this example, create a class diagram from the Diagram Tree window as shown below:



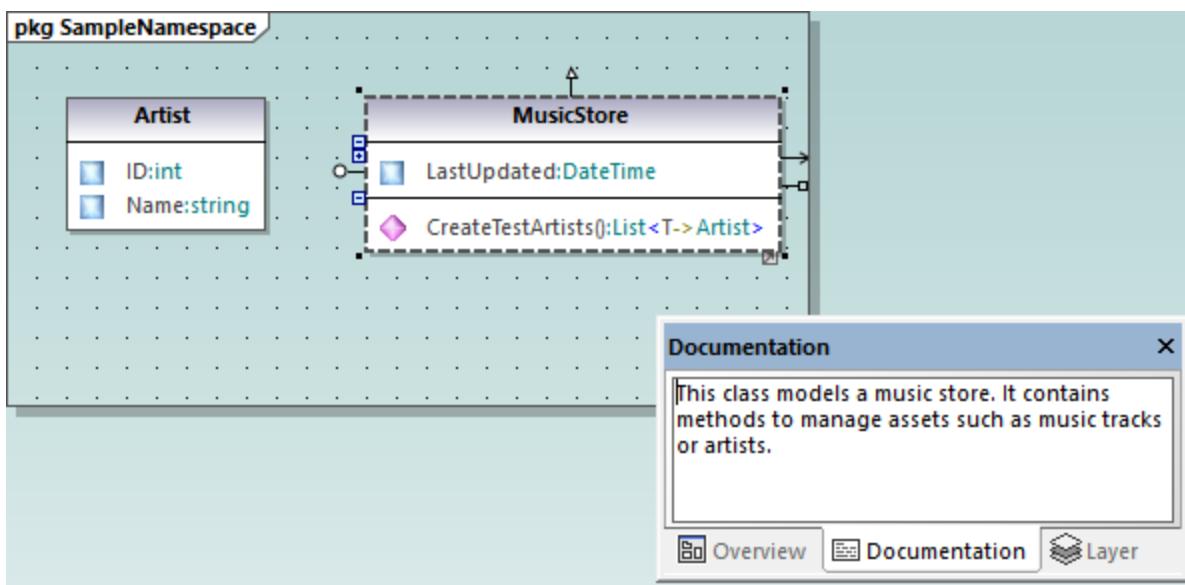
This example assumes that all your classes must be generated under the "SampleNamespace" namespace. Therefore, when prompted to select an owner for the diagram, select the "SampleNamespace" package. If you choose a different package, any elements that you add to the diagram will belong to the same package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

Next, create the classes, types, and other elements required in your model, for example, a simple diagram that contains an `Artist` class and a `MusicStore` class:



In the diagram above, the `Artist` class was created first. That's because the `CreateTestArtists` method of the `MusicStore` class returns a `List<Artist>`, so it's necessary that the `Artist` type already exists. For step-by-step instructions about designing classes and their members, see [Class Diagrams](#)<sup>27</sup>, as well as the [How to Model...](#)<sup>103</sup> chapter.

Optionally, click the `MusicStore` class on the diagram and add some documentation by typing the text in the [Documentation Window](#)<sup>90</sup>. This lets you generate code comments for this class.

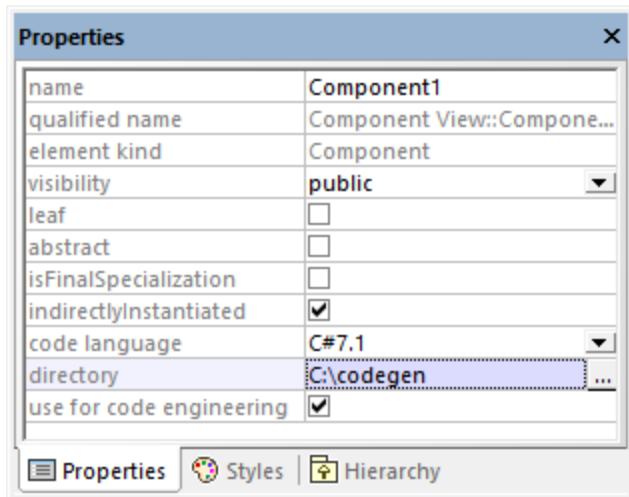


## Configure the project for code engineering

If you haven't done this yet, save the project to a directory. Next, right-click the "Component View" package in the [Model Tree Window](#)<sup>79</sup> and add a new **Component** (that is, a software component) to it. Click the new software component and, in the **Properties** window, set the following properties:

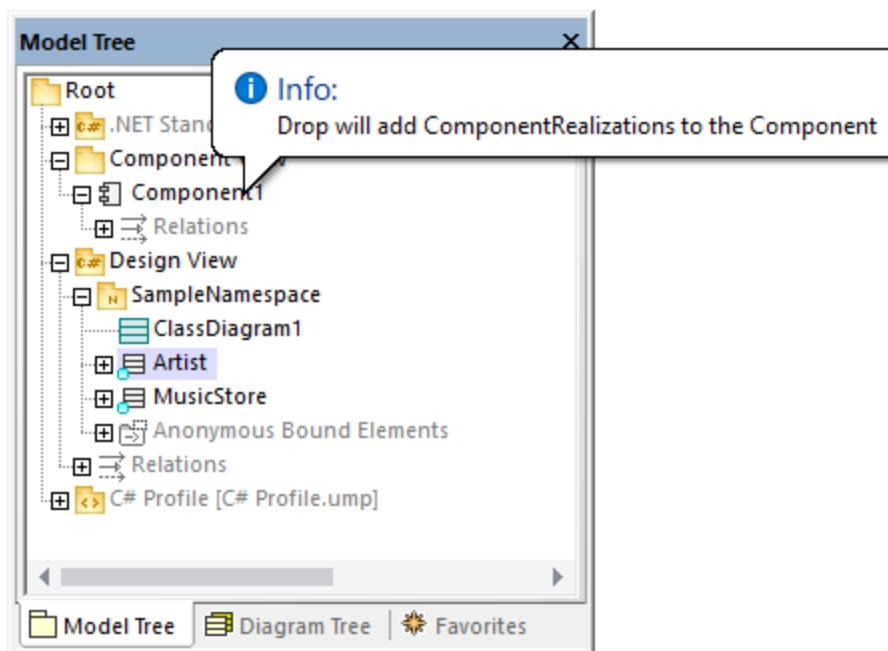
- Code language of the component ("C# 7.1", in this example)
- Code generation directory ("C:\codegen", in this example).

Also, ensure that the "use for code engineering" property is set to **True**.



Next, create a **ComponentRealization** relationship between the classes from which C# code must be generated and the code engineering component. This can be done either from a [Component diagram](#)<sup>49</sup>, or, more simply, as follows:

- In the Model Tree window, click the class to be realized by the component (Artist, in this example) and drag and drop onto the code engineering component (Component1).



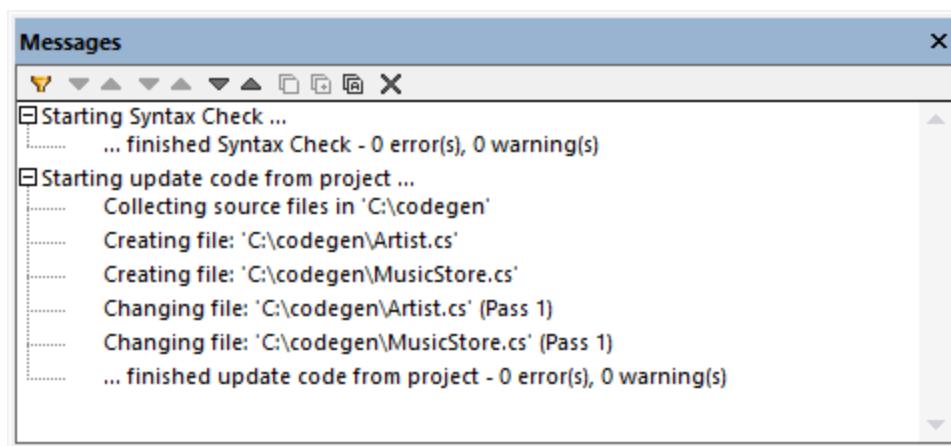
Perform the same step for the `MusicStore` class.

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

## Generate C# code

You can now generate the actual C# code, as follows:

- On the **Project** menu, click **Merge Program Code from UModel Project**. (Alternatively, press **F12**). A dialog box appears where you can adjust whether changes in code should be merged with those in the code, or overwrite them (if applicable). For the scope of this example, you can select **Overwrite...** since a new project is getting generated.
- To include the class documentation as comments in the generated code, click **Project Settings**, and then select the **Write Documentation as DocComments** check box. For more information, see [Code Generation Options](#).
- Click **OK**. The Messages window displays the code engineering result.



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

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

## 6.2.6 Example: Generate Java Code

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

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

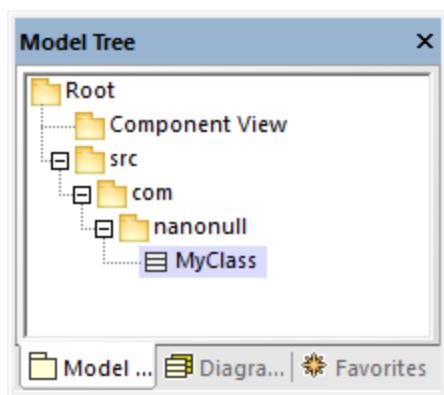
### Creating a new UModel project

You can create a new UModel project as follows:

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

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

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



## Preparing the project for code generation

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

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

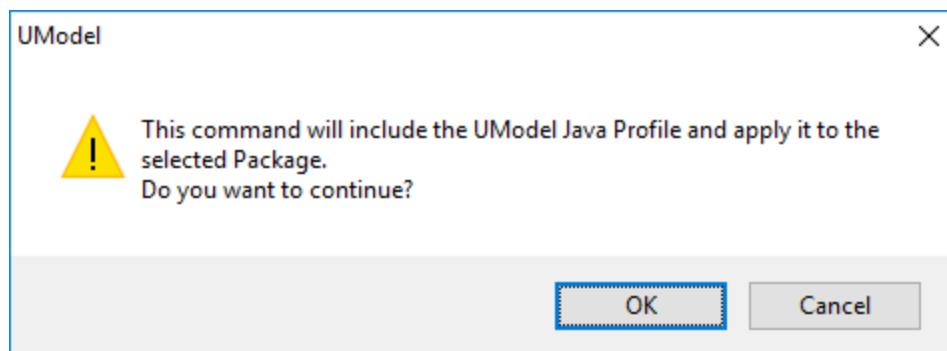
All of these requirements are explained in more detail below. Note that you can always check if the project meets all code generation requirements, by validating it:

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

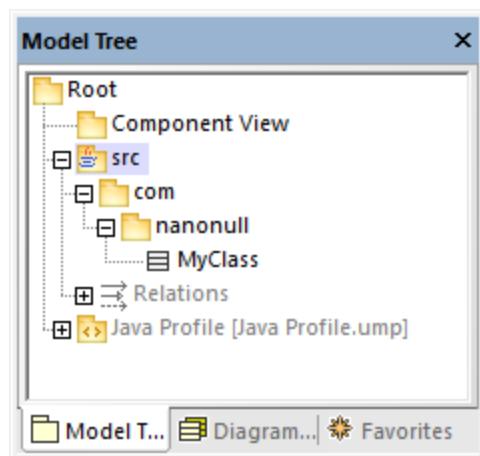
If you validate the project at this stage, the Messages window displays a validation error ("No Namespace Root found! Please use the context menu in the Model Tree to define a Package as Namespace Root"). To resolve this, let's assign the package "src" to be the namespace root:

- Right-click the "src" package and select **Code Engineering | Set As Java Namespace Root** from the context menu.

- When prompted that the UModel Java Profile will be included, click **OK**.

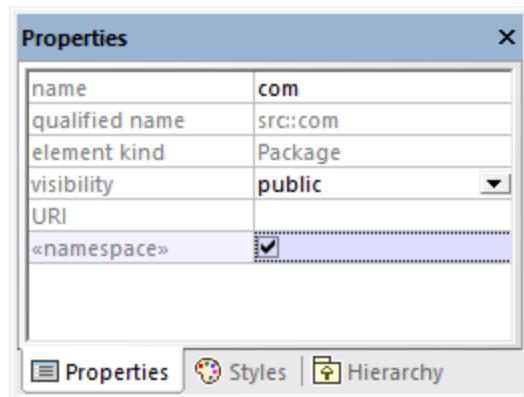


Notice the package icon has now changed to , which signifies that this package is a Java namespace root. Additionally, a Java Profile has been added to the project.



The actual namespace can be defined as follows:

- Select the package "com" in the **Model Tree** window.
- In the **Properties** window, enable the <<namespace>> property.

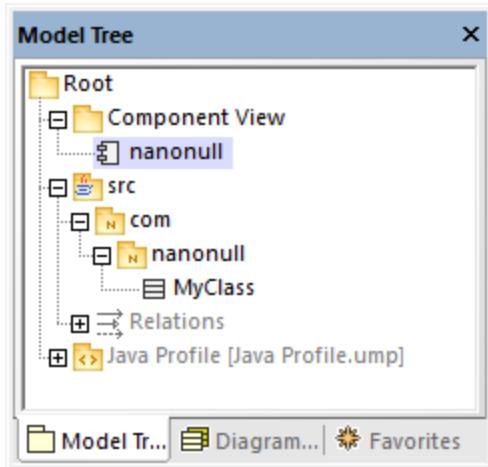


- Repeat the step above for the "nanonull" package.

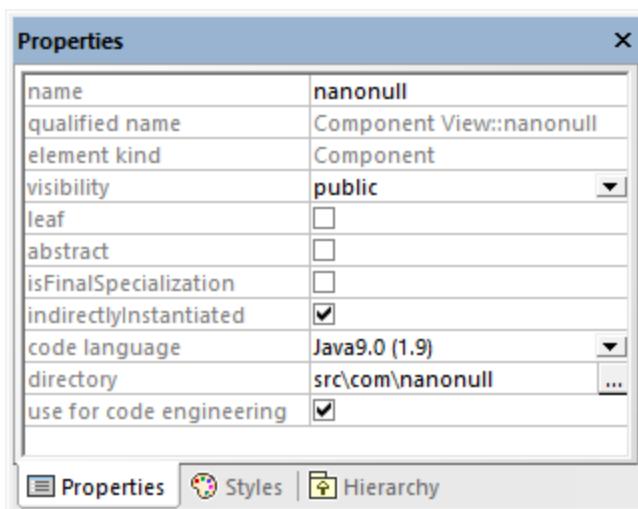
Notice that the icon of both "com" and "nanonull" packages has now changed to , which indicates these are now namespaces.

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

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



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



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

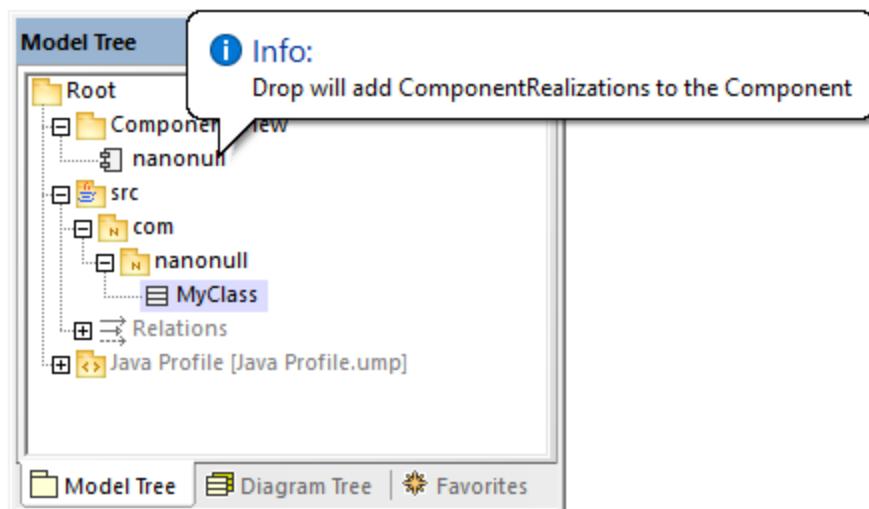
**Note:** The code generation path can be absolute or relative to the .ump project. If it is relative as in this example, a path such as `src\com\nanonull` would create all the directories in the same directory where the UModel project was saved.

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

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

The component realization relationship can be created as follows:

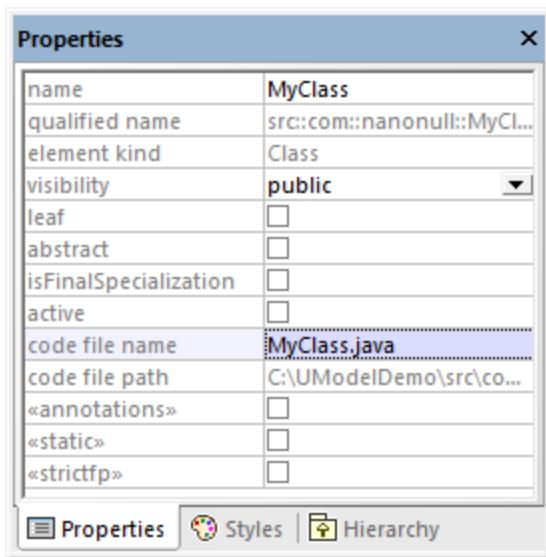
- In the **Model Tree** window, drag from the `MyClass` created previously and drop onto component `nanonull`.



The component is now realized by the project's only class `MyClass`. Note that the approach above is just one of the ways to create the component realization. Another way is to create it from a component diagram, as illustrated in the tutorial section [Component Diagrams](#) 49.

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

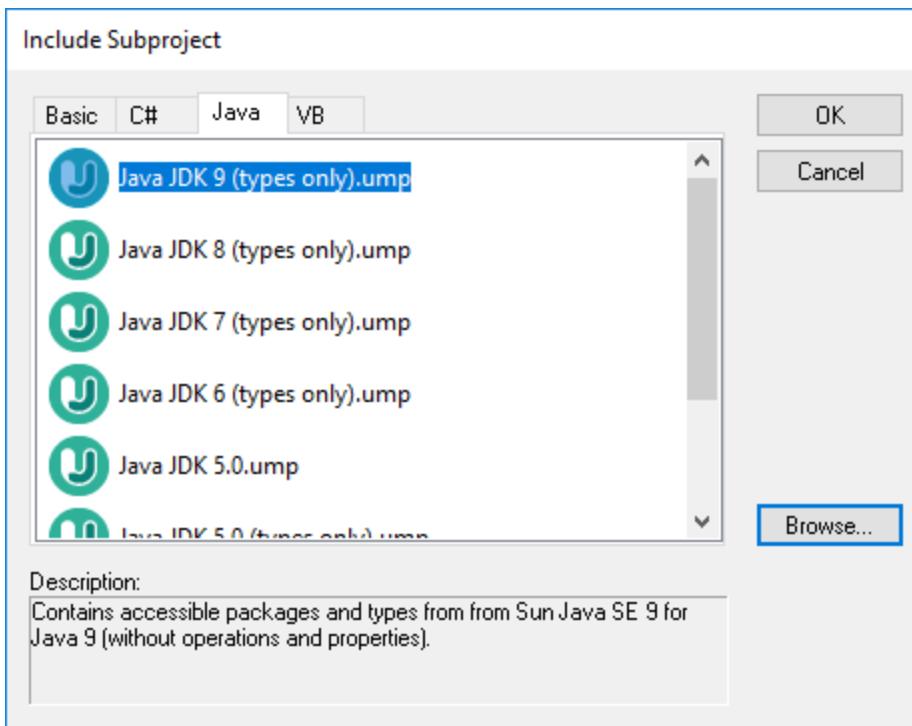
1. Select the class `MyClass` in the **Model Tree** window.
2. In the **Properties** window, change the property **code file name** to the desired file name (in this example, `MyClass.java`).



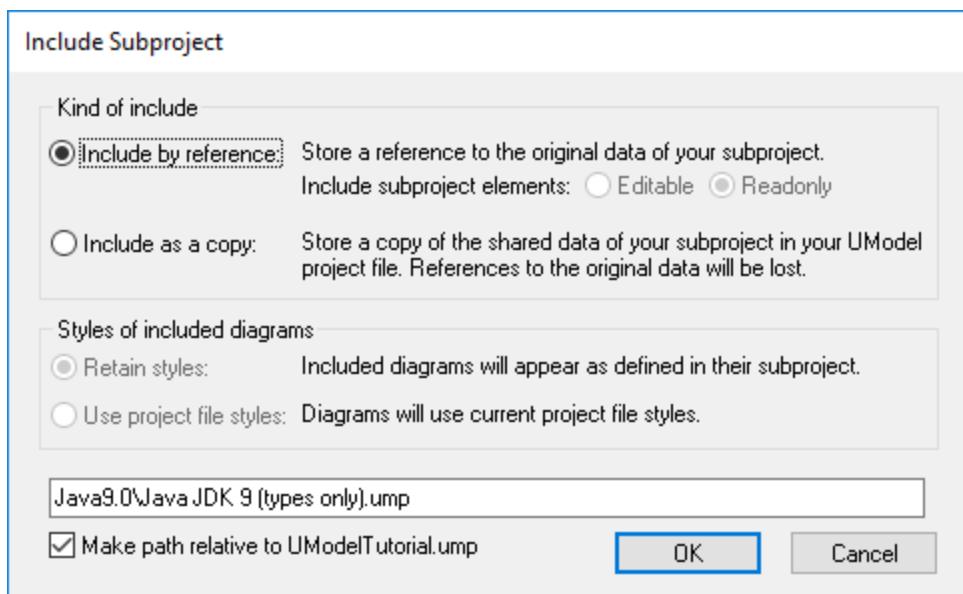
## Including the JDK types

Although this step is optional, it is recommended that you include the Java Development Kit (JDK) language types, as a subproject of your current UModel project. Otherwise, the JDK types will not be available when you create the classes or interfaces. This can be done as follows (the instructions are similar for C# and VB.NET):

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



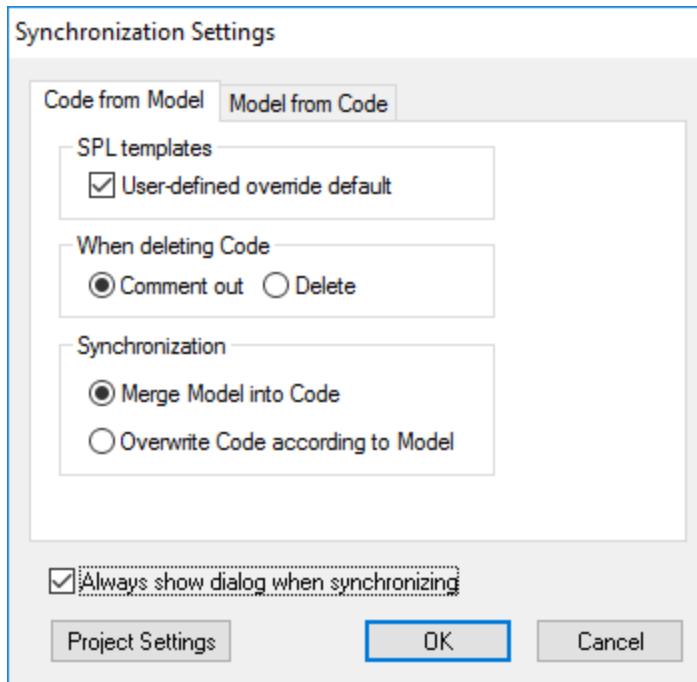
3. When prompted to include by reference or as a copy, select **Include by reference**.



## Generating code

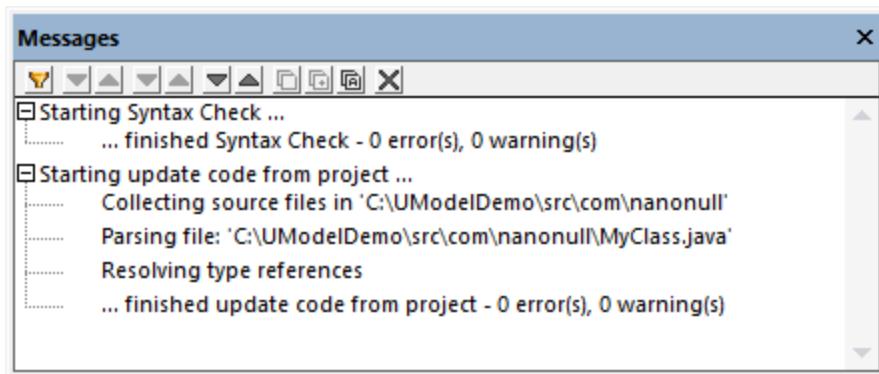
Now that all prerequisites have been met, code can be generated as follows:

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



2. Leave the default synchronization settings as is, and click **OK**. A project syntax check takes place

automatically, and the **Messages** window informs you of the result:



## Modifying code outside of UModel

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

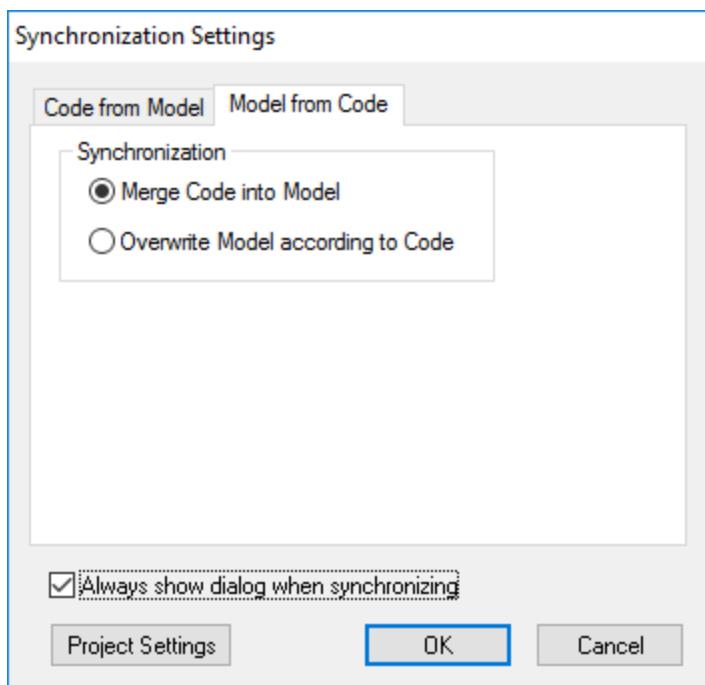
```
package com.nanonull;
public class MyClass{
    public float sum(float num1, float num2) {
        return num1 + num2;
    }
}
```

*MyClass.java*

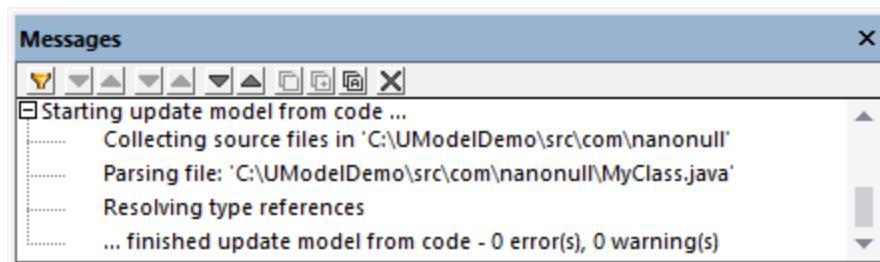
## Merging code changes back into the model

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

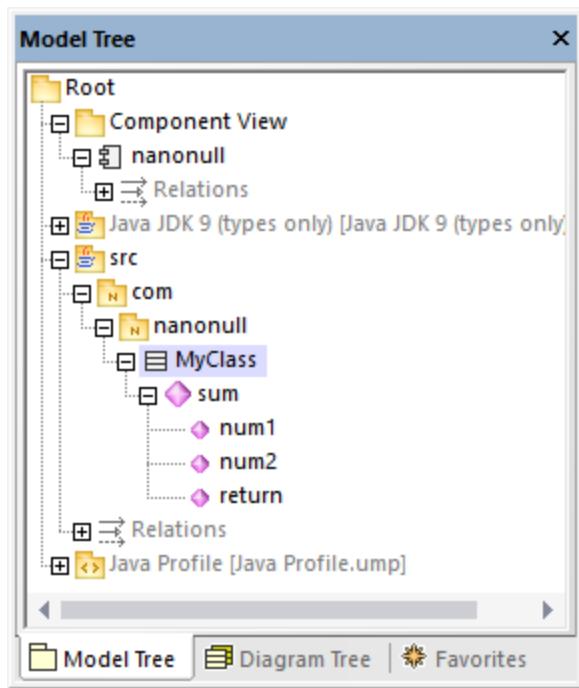
1. On the **Project** menu, click **Merge UModel Project from Program Code** (Alternatively, press **Ctrl + F12**).



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



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



## 6.2.7 SPL Templates

When generating C#, Java, or VB.NET code, as well as XSD schemas, UModel uses a templating language called SPL (Spy Programming Language). The SPL templates dictate the syntax of the generated code files. It is possible to customize the SPL templates, for example, in order to slightly change the syntax of the generated code. Editing SPL templates is meaningful only for languages supported by UModel. If you want to create completely new SPL templates for other languages, it would be possible to generate new code but it would not be possible to update existing code (since the language syntax would be unknown to UModel).

The default SPL templates are available in the **UModelSPL** directory relative to the program installation directory.

Do not modify the existing default SPL templates, since these directly affect the default code generation. Should you need to customize code generation, create custom templates instead, as shown below.

SPL templates are only used when new code is generated (that is, when new classes, operations etc have been added to the model, and then code generation takes place). Any existing code is not affected by the SPL templates.

For an introduction to SPL, see [SPL Reference](#)<sup>521</sup>.

### To modify the provided SPL templates:

1. Locate the provided SPL templates in the UModel installation directory ("Program Files"), for example: ...\\UModel2022\\UModelSPL\\Java\\Default.

2. Copy the SPL files you want to modify into the **parent** directory. For example, if you want to modify the appearance of a Java class in generated code, copy the **Class.spl** file from ...  
**\UModel2022\UModelSPL\Java\Default** to ...**\UModel2022\UModelSPL\Java**.
3. Make the changes to the .spl file(s) and save them.

**To use the custom SPL templates:**

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

## 6.3 Importing Source Code

Existing Java, C#, and VB.NET program code can be imported into UModel (a process also known as "reverse engineering"). The following project types can be imported into UModel:

- Java projects (Eclipse .project files, NetBeans project.xml files, and JBuilder .jpx files)
- C# and VB.NET projects (Visual Studio .sln, .csproj, .csdprj, .vbproj, .vbp as well as Borland .bdsproj project files)

In addition to importing source code from a source project, it is also possible to import code from a source directory. Importing from a source directory works in a similar way, and is particularly useful when your code doesn't use any of the project types listed above. For an example of importing a source directory, see [Reverse Engineering \(from Code to Model\)](#)<sup>69</sup>.

It is possible to import source code either into a new, empty UModel project or into an existing UModel project. During the import, you can specify whether the imported elements should overwrite those in the model (if any), or be merged into the model. Optionally, Class and Package diagrams can be generated automatically as you import code.

The import wizard includes various import options specific to each platform (Java, .NET). For example, if the imported Java/C#/VB.NET code contains comments, these can be optionally converted to UModel documentation. For a complete list of options, see [Code Import Options](#)<sup>189</sup>.

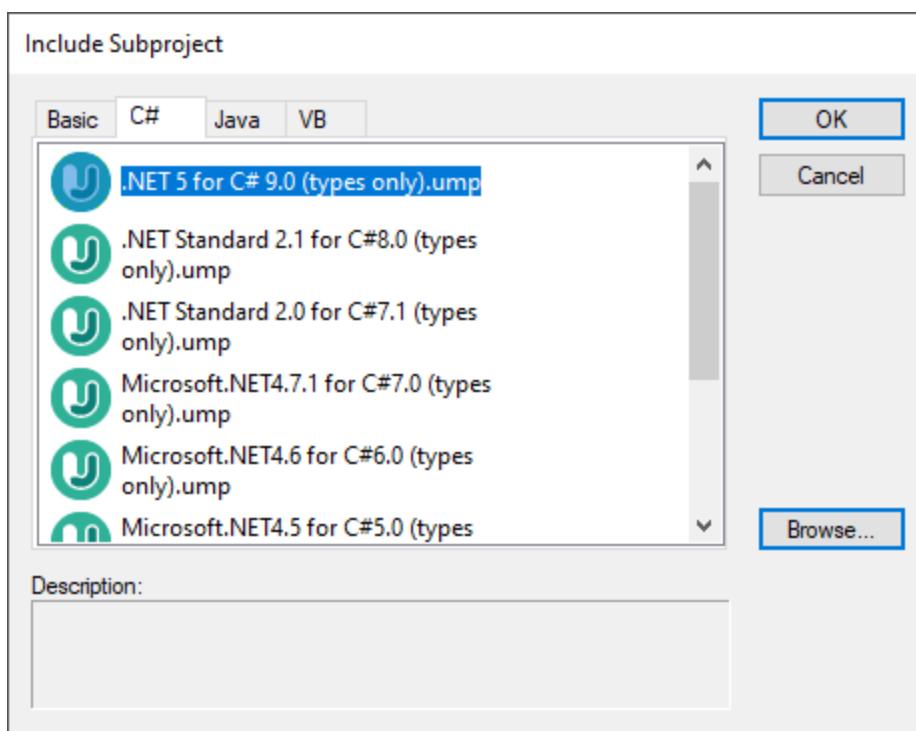
Once your C#, VB.NET, or Java code has been imported into UModel, it is possible to modify the model (for example, add new classes, or rename properties and operations), and optionally synchronize it back with the original code, thus achieving full round-trip engineering, see [Synchronizing the Model and Source Code](#)<sup>212</sup>.

### Prerequisites

UModel includes several built-in sub-projects that were created specifically for code engineering and which include the data types applicable to each supported language and platform. Before attempting to import source code into a UModel project, it is recommended to include the built-in UModel subproject applicable to the corresponding programming language and platform, see [Including Subprojects](#)<sup>158</sup>. Otherwise, certain data types will not be recognized and will be placed after import into a separate package called "Unknown externals".

#### To include a subproject with the required language data types:

1. On the **Project** menu, click **Include Subproject**.
2. Click the tab applicable to the source language and platform (for example, Java 8.0, C# 6.0, VB 9.0), and then click OK.



Note the following:

- When you include a data type subproject for a particular language, UModel also automatically adds the profile of that language to your project. The profile subproject (.ump) contains only the most basic types and is different from the data type subproject (also .ump) which contains more extensive type definitions.
- If you perform the import without including a data type subproject, the import operation will take place nonetheless, and UModel will also automatically include the profile of that language to the project. However, any unknown types will be placed into the "Unknown externals" package. To solve this, make sure to include the data types subproject for the required language and platform, as explained above.

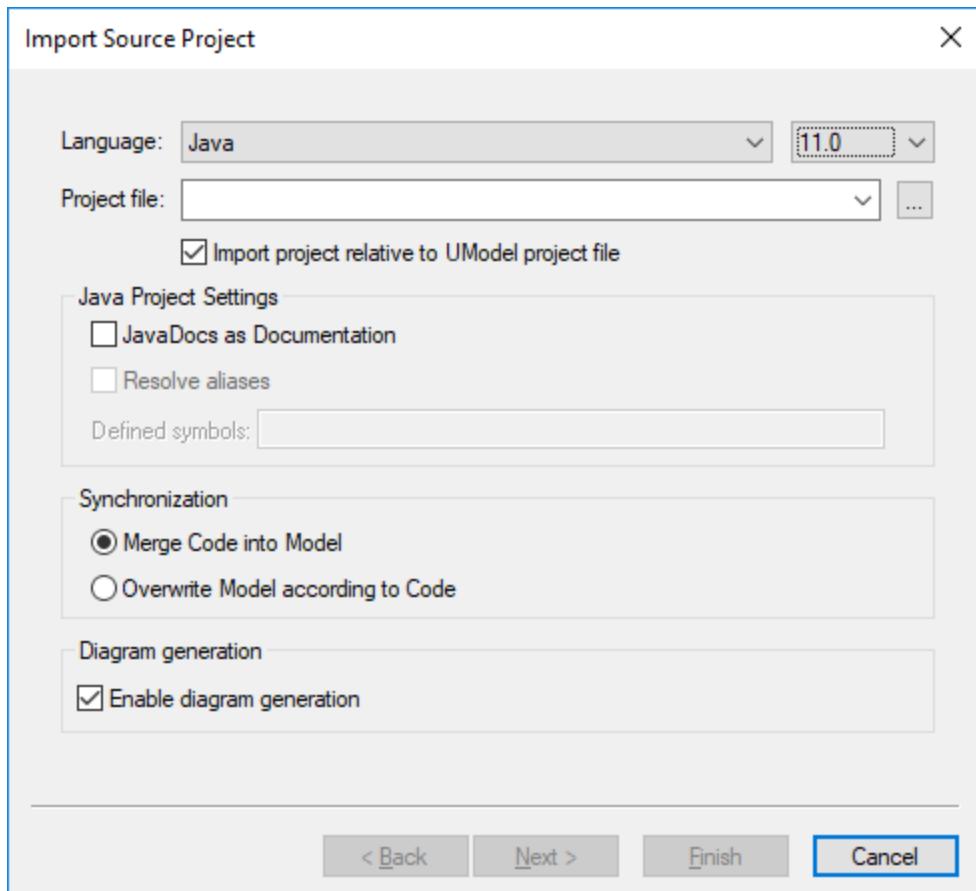
## Importing source code from a project

1. On the **Project** menu, click **Import Source Project**. (Alternatively, if you would like to import code from an existing directory, select **Import Source Directory**.)
2. Select the language version of the source project (for example, Java 8.0, C# 6.0).
3. Click **Browse** and select the source project file.
4. Set or change the required import options, see also [Code Import Options](#) <sup>189</sup> (note that these options depend on the language selected in step 2).
5. Click **Finish** to complete the wizard.

For a step-by-step example, see [Example: Import a C# Project](#) <sup>191</sup>.

### 6.3.1 Code Import Options

When importing program code into a UModel project, you may need to set or change the options listed below. These options are available on the dialog box which appears when you run the menu command **Project | Import Source Project** or **Project | Import Source Directory**.



*Import Source Project dialog box*

Most of the options on the dialog box above can also be changed at any time later, see [Code Synchronization Settings](#)<sup>216</sup>.

The following options are applicable to all project types, regardless of the language or platform:

Option	Description
<i>Import project relative to UModel project file</i>	By default, this option is selected, which means that a relative path dependency will be established between the UModel project and the imported source code project.  After source code is imported, a UML component is generated automatically in the UModel project (it is available in the Model Tree, as a child of "Component View"). This component realizes

Option	Description
	the interfaces or classes to be engineered; it also specifies the code engineering options, including the path to the source project or directory. This will be a relative path if <b>Import project relative to UModel project file</b> is selected; otherwise, it will be an absolute path.
<i>Merge Code into Model / Overwrite Model according to Code</i>	<p>If <b>Merge...</b> is selected, potential name conflicts (such as package or class names) will be resolved by appending a number to the element that is being imported.</p> <p>If <b>Overwrite...</b> is selected, and if there are name conflicts, the imported element will take precedence over (overwrite) the one existing in the project.</p>
<i>Enable diagram generation</i>	Optionally, select this check box if you want to generate Class and Package diagrams from the imported classes. When this check box is selected, the import wizard includes additional steps which enable you to customize the look of the generated diagrams.

The following options are applicable only to C# and VB.NET projects:

Option	Description
<i>DocComments as Documentation</i>	Select this check box to convert comments found in the C# code into UModel element documentation (see also <a href="#">Documentation</a> ).
<i>Resolve aliases</i>	<p>This check box is enabled by default. If your C# or VB.NET code contains namespace or class aliases like in the code listing below, it is recommended to keep this check box selected. Otherwise, associations and dependencies involving aliased classes and namespaces in your code may not be detected automatically by UModel during the import (and thus would not be present in the model).</p> <div data-bbox="556 1374 1274 1438" style="background-color: #f0f0f0; padding: 5px;"> <pre><code>using Q = System.Collections.Generic.Queue&lt;String&gt;; Q myQueue;</code></pre> </div> <p><i>Example of an alias in C# code</i></p> <p>During the source code import, any potentially conflicting aliases are added to the "Unknown externals" package of the UModel project if their use is unclear.</p> <p>When you update the code back from the model (round-trip engineering), aliases will be retained as they exist in the generated code.</p> <p>The <b>Resolve aliases</b> option can be changed at any time later, see <a href="#">Code Synchronization Settings</a>. If you enable this option after (not before) the import operation, UModel prompts you to update the project from the code again, since the option also has consequences for forward engineering.</p>

Option	Description
<p><i>Defined symbols</i></p>	<p>If your C# or VB.NET code includes symbols that are defined through preprocessor directives such as <code>#if</code>, <code>#endif</code>, you can instruct UModel to take them into account while reverse engineering code.</p> <pre data-bbox="556 439 1416 635">#if DEBUG     static void DisplayMessage()     {         Console.WriteLine("Please wait...");     } #endif</pre> <p><i>Example of a conditional compilation symbol in C# code</i></p> <p>For example, if you reverse engineer the code above, the method <code>DisplayMessage()</code> will only be imported into the model if you specified the <code>DEBUG</code> symbol.</p> <p>To specify conditional compilation symbols, enter them in the "Defined symbols" text box, delimited by a semicolon.</p> <p>During the reverse engineering process, UModel outputs all symbols used in the source code in the Messages window.</p>

The following option is applicable only to Java projects:

Option	Description
<p><i>JavaDocs as Documentation</i></p>	<p>Select this check box to convert JavaDocs-style comments found in the code into UModel element documentation (see also <a href="#">Documentation</a> <small>90</small>).</p> <p><b>Note:</b> Only comments applicable for Java classes, interfaces, operations, and properties are converted.</p>

### 6.3.2 Example: Import a C# Project

This example illustrates how to import into UModel a sample C# solution created with Visual Studio. The source solution is available as a .zip archive at the following path: **C:\Users\<username>\Documents\Altova\UModel2022\UModel\Examples\Tutorial\Anagram\_CSharp.zip**.

It is not necessary to compile the solution with Visual Studio before importing it; however, make sure to unzip the **Anagram\_CSharp.zip** archive to a folder of your choice before proceeding to the steps below.

Our goal in this example is to reverse engineer the C# solution and create a UModel project from it. As we import code, we will opt to generate class and package diagrams automatically.

#### Step 1: Create a new project

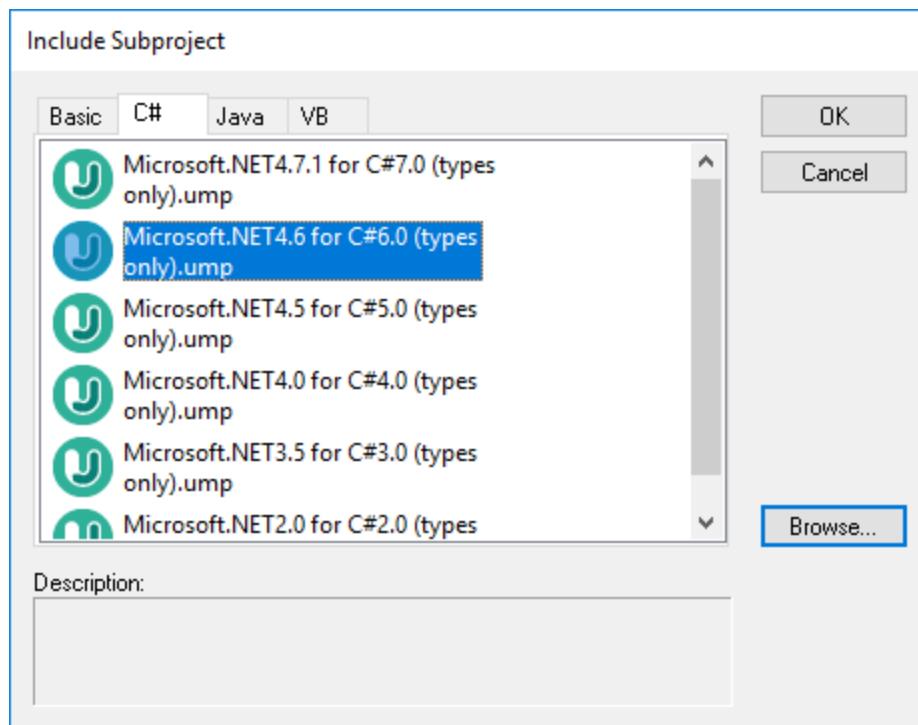
It is possible to import source code either into existing or new UModel projects. For the scope of this example, we will be importing code into a new UModel project.

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

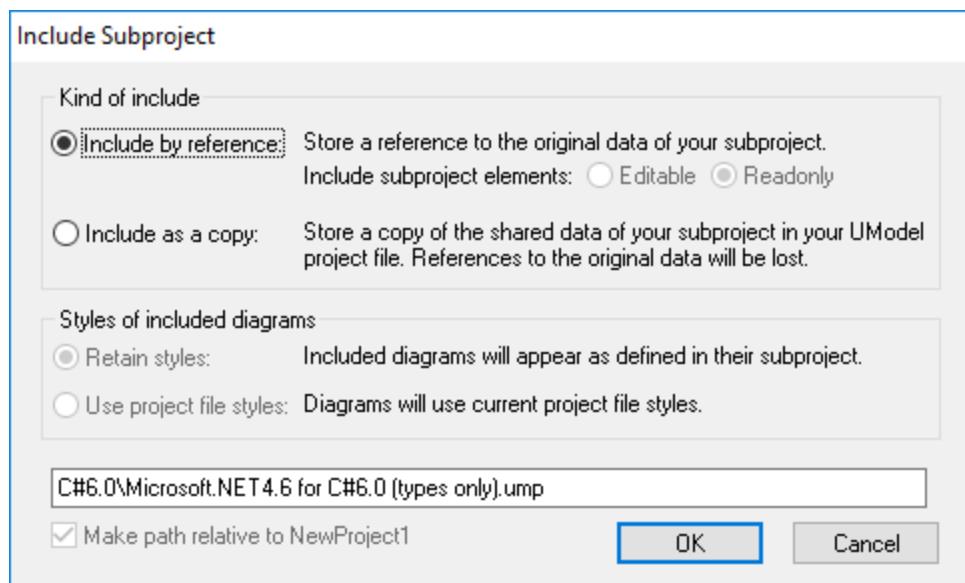
## Step 2: Include the C# language types

The source project was written in C# with Visual Studio 2015, so we will include a built-in UModel project that contains the C# 6.0 language types (since the C# language version corresponding to Visual Studio 2015 is 6.0). Earlier versions of C# are also likely to work with our C# example solution.

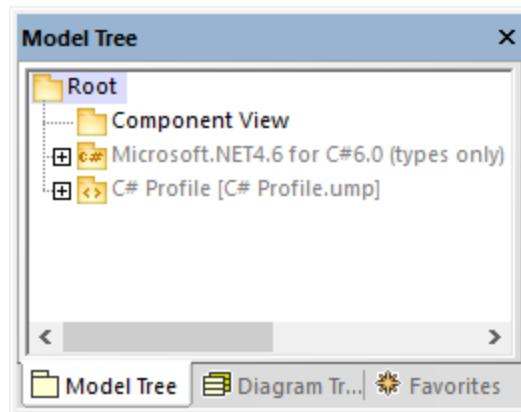
1. On the **Project** menu, click **Include Subproject**.
2. Click the **C#** tab.



3. Select the project **Microsoft .NET 4.6 for C# 6.0 (types only).ump**, and click **OK**.
4. When prompted to select the kind of include (by reference or as a copy), leave the default option as is.

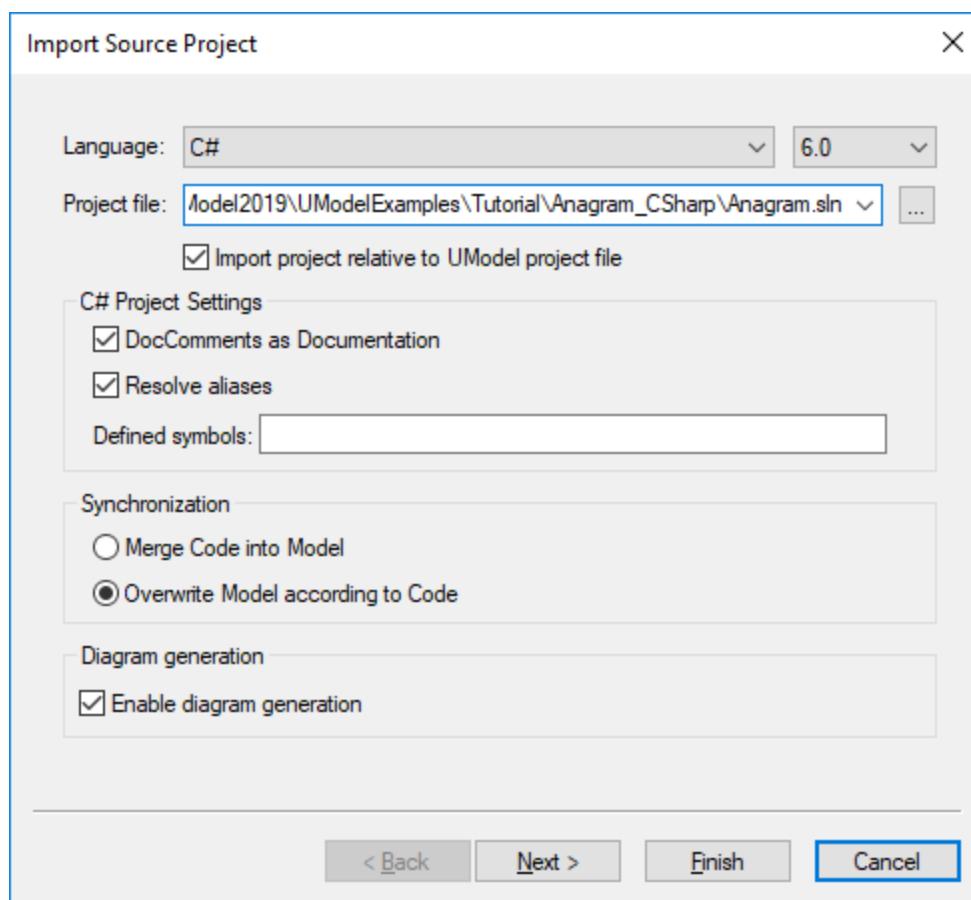


As a result, both the C# language types and the C# language profile are included and visible in the Model Tree:

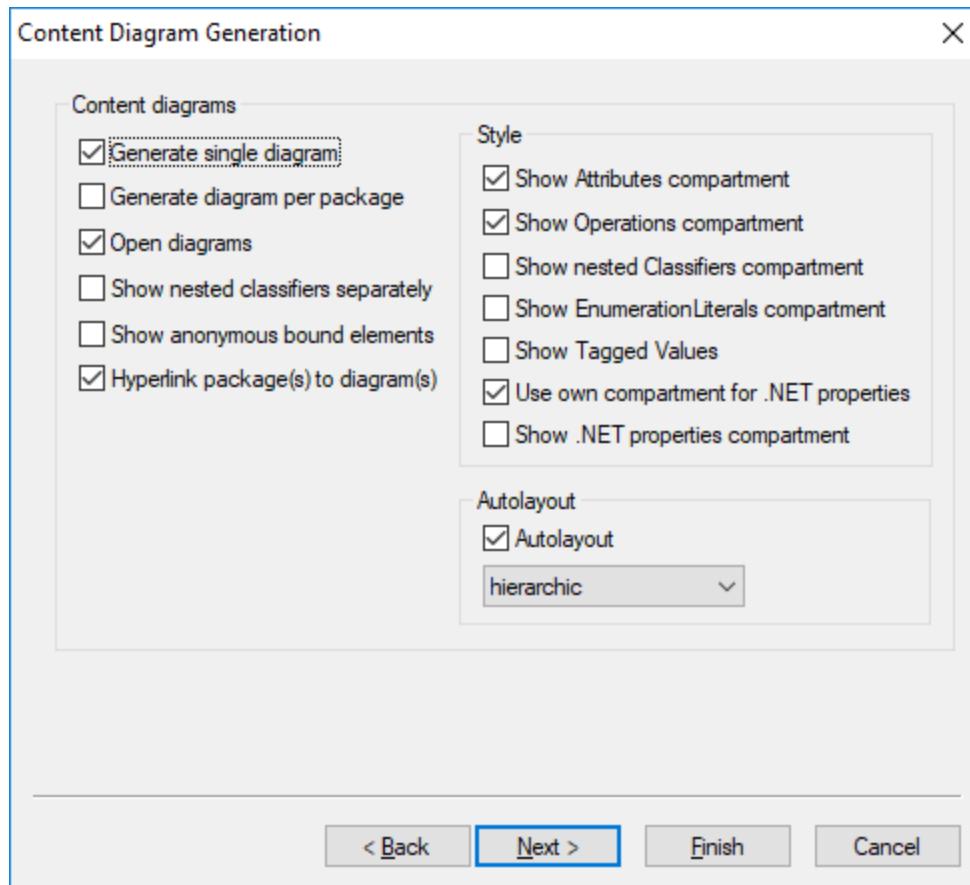


### Step 3: Import the C# solution

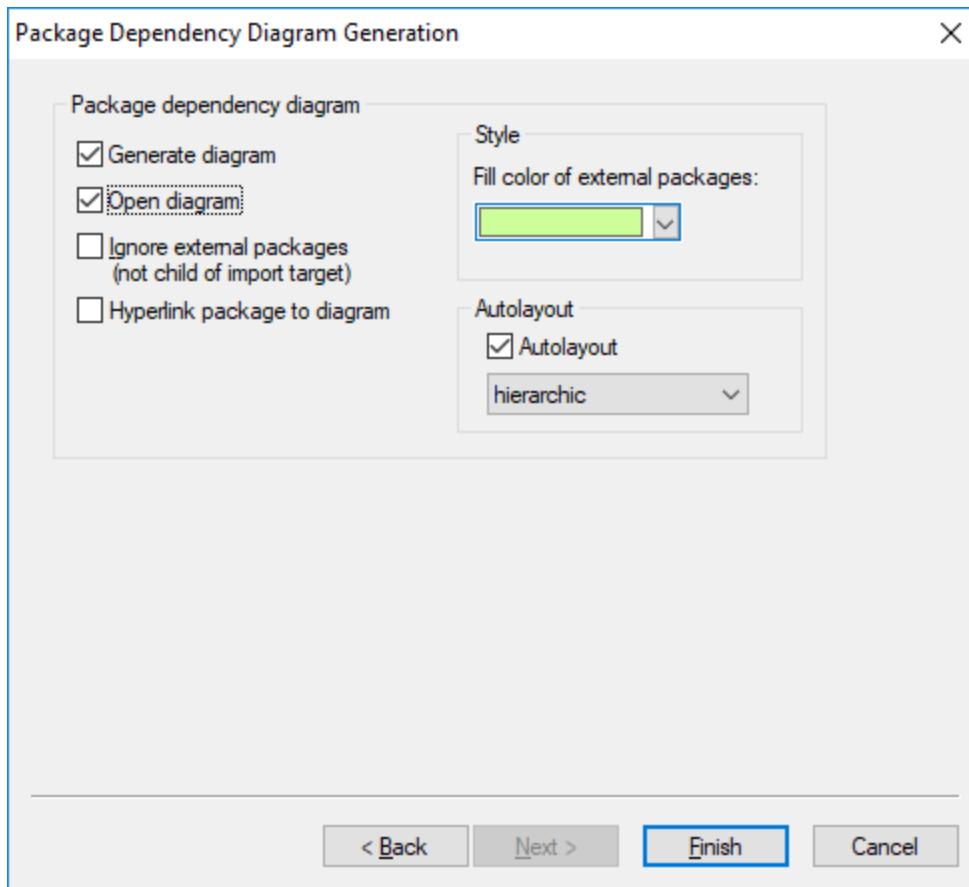
1. On the Project menu, click **Import Source Project**.



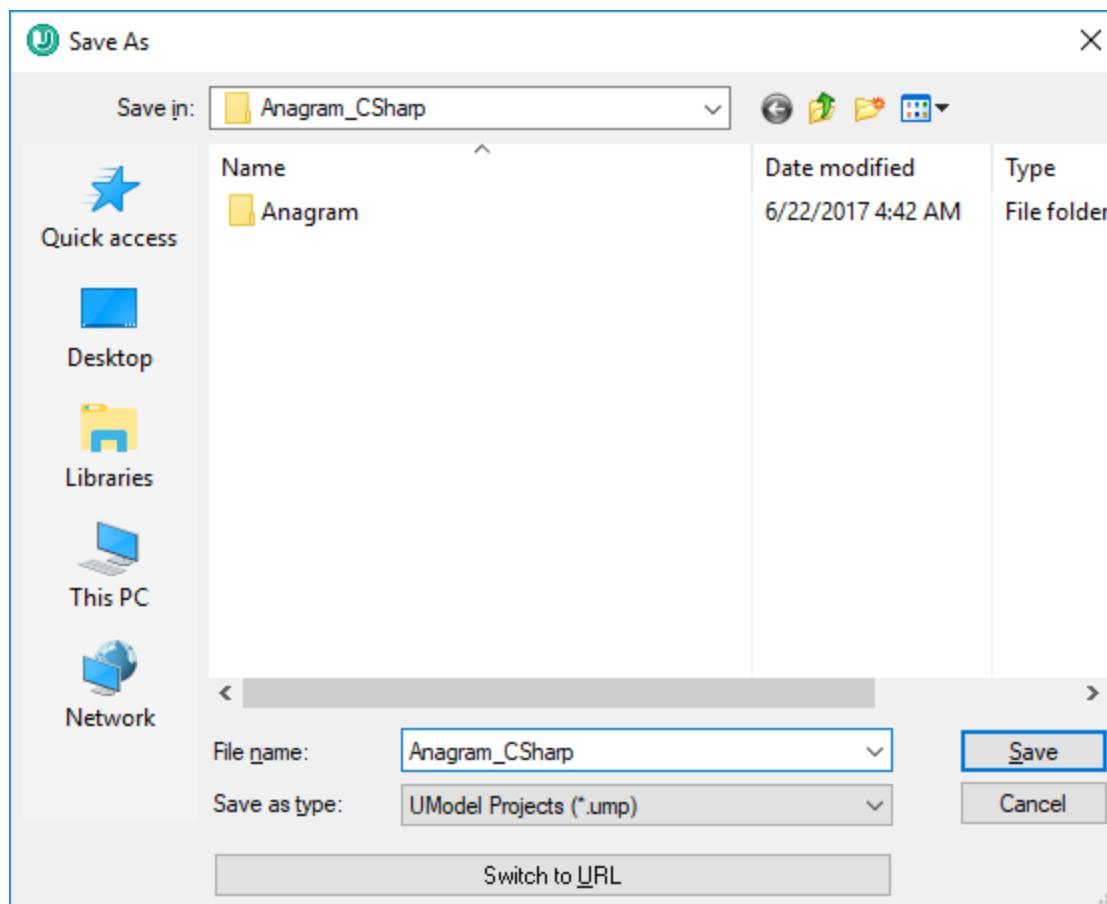
2. Select **C# 6.0** as language.
3. Click **Browse** [...] next to **Project file** and browse for the solution .sln file.
4. Select the **DocComments as Documentation** check box (this will import the code comments found on operations or properties into the model).
5. Since we are importing code into a new UModel project, select the option **Overwrite Model according to Code** (the other option **Merge Code into Model** is preferable when you import into an existing project).
6. Click **Next**.
7. Select the diagram generation options as shown below, and click **Next**. (These options are applicable to Class diagrams generated automatically on code import.)



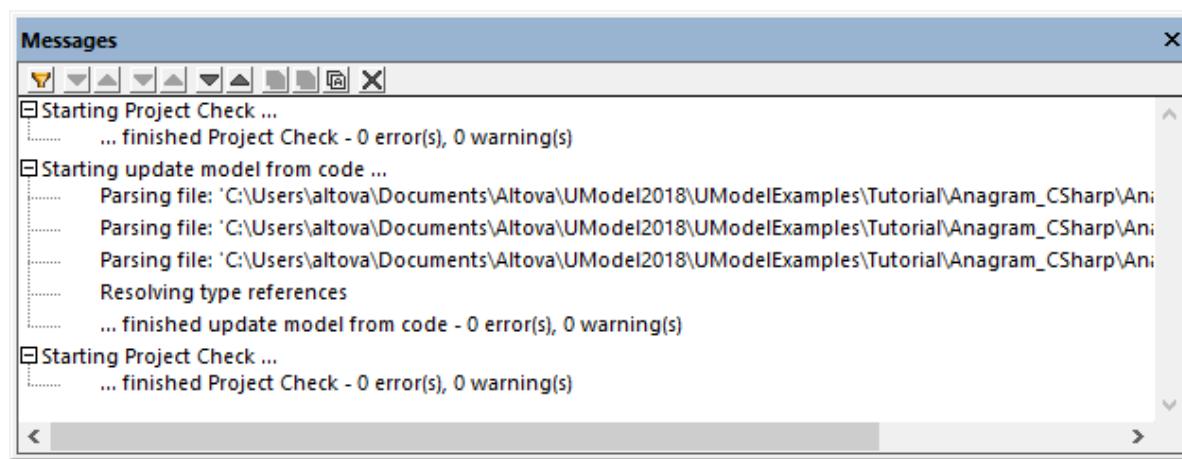
8. Select the diagram generation options as shown below, and click **Finish**. (These options are applicable to Package diagrams generated automatically on code import.)



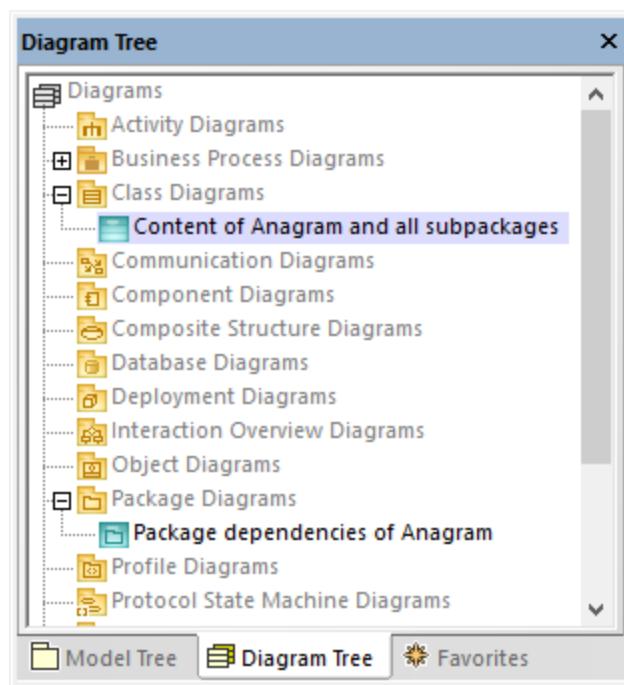
9. Enter a name and select a destination folder for the new UModel project, and click **Save** (by default, this dialog box displays the same folder as the solution you are importing).



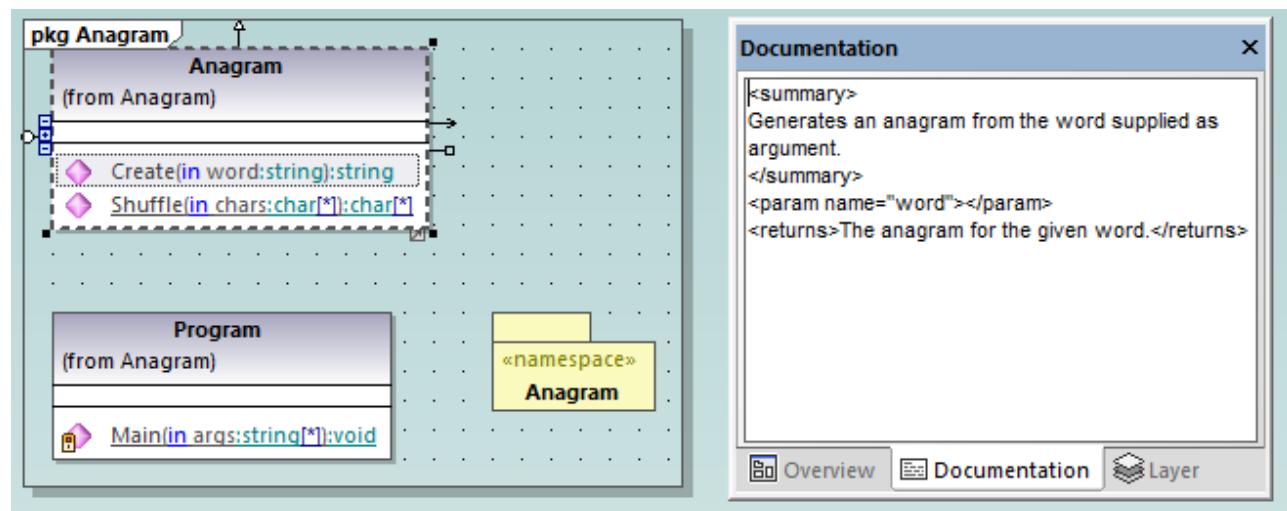
The progress of the reverse engineering operation is shown in the Messages window.



Also, when code import completes, all generated diagrams are opened automatically since this option was selected before code generation. All generated diagrams are available in the Diagram Tree:



Since we opted to generate documentation from the source code, the imported documentation is visible in the **Documentation** window if you click, for example, the `Create` operation of the `Anagram` class:



**Note:** The documentation is added only if the option **DocComments as Documentation** was selected while importing the C# solution (see "Step 3: Import the C# Solution" above).

## 6.4 Importing Java, C# and VB.NET Binaries

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

- To import Java binary files, a [supported version](#) <sup>11</sup> of the Java Runtime Environment (JRE) or Development Kit (JDK) must be installed. Type import is supported for Java .class files or .jar class archives adhering to the Java Virtual Machine Specification. This includes Java Virtual Machines such as OpenJDK, SapMachine, Liberica JDK, and others, see [Adding Custom Java Runtimes](#) <sup>200</sup>.
- To import C# or VB.NET binary files, .NET Framework, .NET Core, .NET 5, or .NET 6 must be installed, as applicable. For best results, select the **any (use disassembler)** option on the import dialog box. After import, any unrecognized types will be placed in the "Unknown externals" package. To prevent (or decrease the number of) unknown externals, apply the UModel profile specific to the version of your code engineering language (for example, ".NET 5 for C# 9.0") *before the import*. See also [Applying UModel Profiles](#) <sup>154</sup>.
- The import of obfuscated binaries is not supported.

The table below lists the available approaches for importing binary types into a UModel project.

C#, VB.NET	Java
Import assembly file (.dll, .exe)	Import class file archive (.jar, .zip)
Import assembly from Global Assembly Cache (GAC)	Import class file (.class) from a package root folder
Import assembly from Visual Studio .NET References	Import class archives from class path
	Import class archives from Java runtime (only for Java versions up to and including Java 8)

You can import binary files by running the **Project | Import Binary Types** menu command. Optionally, you can have UModel generate class and package diagrams from the imported types. For examples, see [Example: Import .NET GAC Assemblies](#) <sup>204</sup> and [Example: Import Java .class Files](#) <sup>206</sup>.

In addition, you can import binary files from the command line (see [UModel Command Line Interface](#) <sup>96</sup>).

When importing binary files into a UModel project, you can specify various import options, including:

- You can import any referencing types, in addition to the types defined in the binary file. In addition, you can restrict importing referencing types to specific Java packages and .NET namespaces.
- You can skip type members while importing. For example, you can import classes and interfaces without their properties and methods.
- You can import types according to their accessibility modifiers (such as private or public). For example, you can import only public classes and skip private, protected, and internal classes.

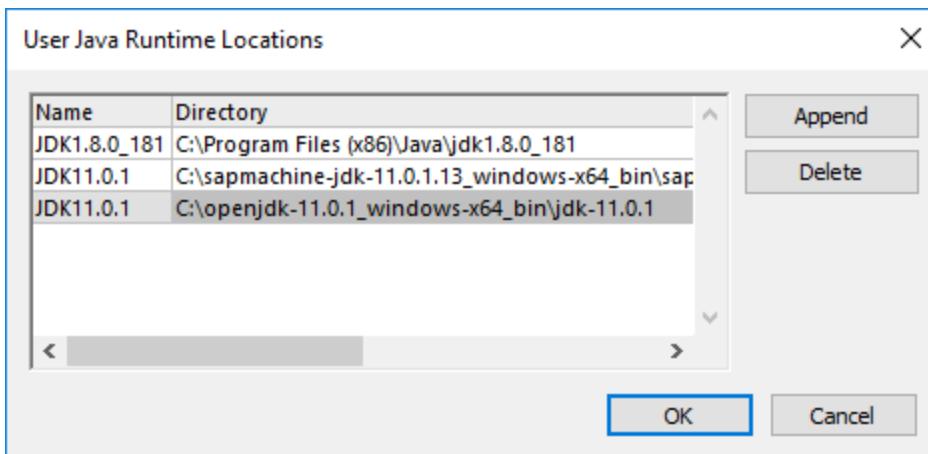
For reference to all options, see [Import Binary Options](#) <sup>200</sup>.

## 6.4.1 Adding Custom Java Runtimes

By default, UModel detects JDKs and JREs if they are *installed* on the local machine. Consequently, these appear in the list of Java runtimes when you start the binary import wizard. This is the case for JDKs and JREs released by Oracle, which come with an installer and register themselves in the system when installed. However, other Java Virtual Machine distributions that do not have an installer must be added manually into UModel. The latter include Oracle OpenJDK, SapMachine, and others.

**To add custom Java runtimes to UModel:**

1. On the **Project** menu, click **Import Binary Types**.
2. Select **Java** as language.
3. Expand the **Runtime** drop-down list, and click **Edit user Java runtime locations**.
4. Click **Append** and browse for the directory of the respective JDK.



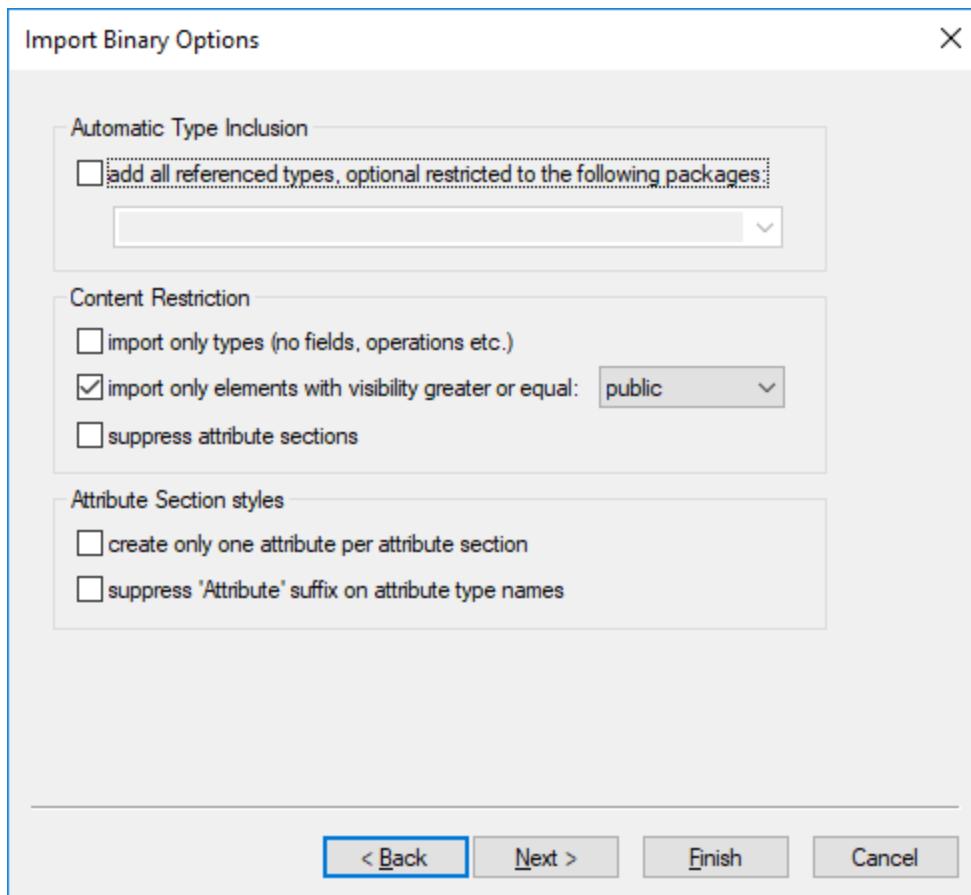
5. Click **OK**.

The selected runtime now appears in the **Runtime** list, and you can select it whenever you need to import binary files targeting that runtime.

Note that these settings affect only the import of binary files. For information about adding a Java Virtual Machine path to be used for JDBC connectivity and Java code generation and import, see [Java Virtual Machine Settings](#)<sup>510</sup>.

## 6.4.2 Import Binary Options

When you run the menu command **Project | Import Binary Types**, one of the wizard steps prompts you to specify the binary import options. The options you can set are described below. Note that the dialog box options may be slightly different, depending on whether you are importing .NET or Java binaries.



*Import Binary Options dialog box*

## Automatic type inclusion

.NET or Java binaries may reference various external assemblies or packages. Select the option **add all referenced types...** if you would like to import all types referenced by the types included in the binary file.

To import referenced types only for specific Java packages or .NET namespaces, enter those packages or namespaces in the adjacent text box. To separate multiple packages or namespaces, use the comma, semi-colon, or space characters.

For example, let's assume that the source .NET .dll file references types from `System.Reflection` and `System.Data` namespaces. If you would like to import types from the `System.Reflection` namespace but not from the `System.Data` namespace, select the option **add all referenced types, optionally restricted to the following packages** and enter "System.Reflection" in the text box.

## Content restriction

Select the option **import only types** to skip members such as fields, operations, properties, and so on.

Select the option **import only elements with visibility greater than or equal to** to import types and type members according to their visibility. The table below lists visibility of types, beginning with types with least

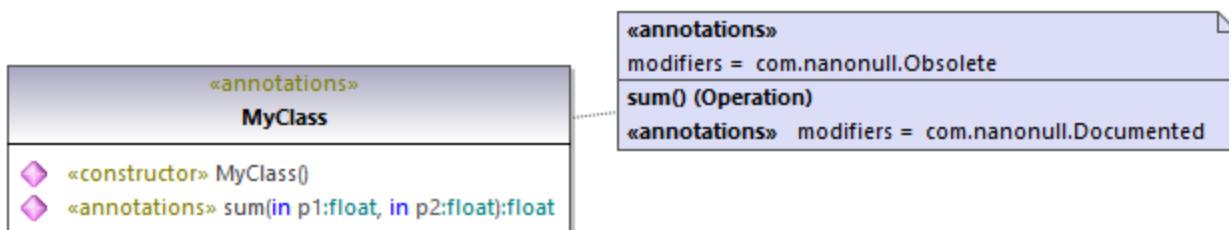
visibility. For example, selecting "private" will import all types, whereas selecting "public" will import only public types and type members.

**Note:** If the check box is not selected, all types will be imported, regardless of their visibility.

.NET	Java
private	private
internal	package (default visibility when no explicit modifier exists)
protected	protected
public	public

The option **suppress attribute sections** is applicable for .NET binaries. By default, UModel imports the C# or VB.NET attributes detected in the binary. Select the **suppress attribute sections** option if you don't want to import attributes. Otherwise, members that were decorated with attributes in the original source code will have the `<<attributes>>` stereotype applied to them after you import the binary into the model. If attributes are imported, you can display them on the diagram as tagged values, by right-clicking the class on the diagram and selecting **Tagged Values | All** from the context menu. For more information, see [Stereotypes and Tagged Values](#)<sup>140</sup>.

The option **suppress annotation modifiers** is applicable for Java binaries. By default, UModel imports Java annotations detected in the binary, provided that their retention policy was defined as `RUNTIME` (not `CLASS` or `SOURCE`). If you don't want to import annotations, select the **suppress annotation modifiers** option. If annotations are imported, members that had annotations in the original source have the `<<annotations>>` stereotype, and annotations appear as tagged values, as illustrated below.



## Attribute section styles

These options are applicable to .NET binaries only. As previously mentioned, if types or type members in the original source code were decorated with attributes, these are imported as tagged values in UModel.

The option **create only one attribute per attribute section** is best illustrated by an example. Let's assume that the original C# source code defined a method with two attributes:

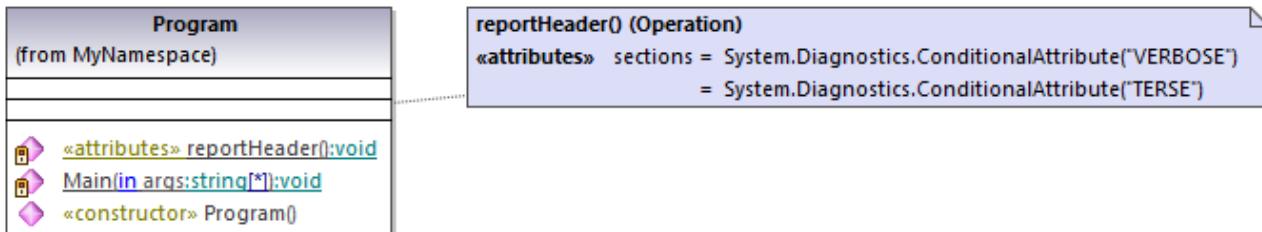
```

using System;
using System.Diagnostics;

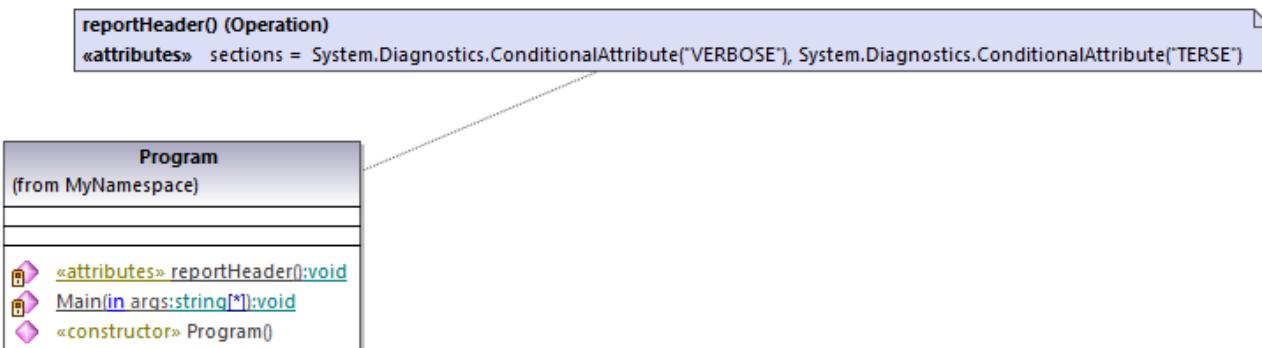
namespace MyNamespace
  
```

```
{  
    class Program  
    {  
        [Conditional("VERBOSE"), Conditional("TERSE")]  
        static void reportHeader()  
        {  
            Console.WriteLine("This is the header");  
        }  
  
        static void Main(string[] args)  
        {  
            reportHeader();  
        }  
    }  
}
```

If the option **create only one attribute per attribute section** is enabled upon importing from the binary file, then each attribute would appear on a separate line inside the "Tagged Values" element :



Otherwise, attributes would appear as comma-separated:



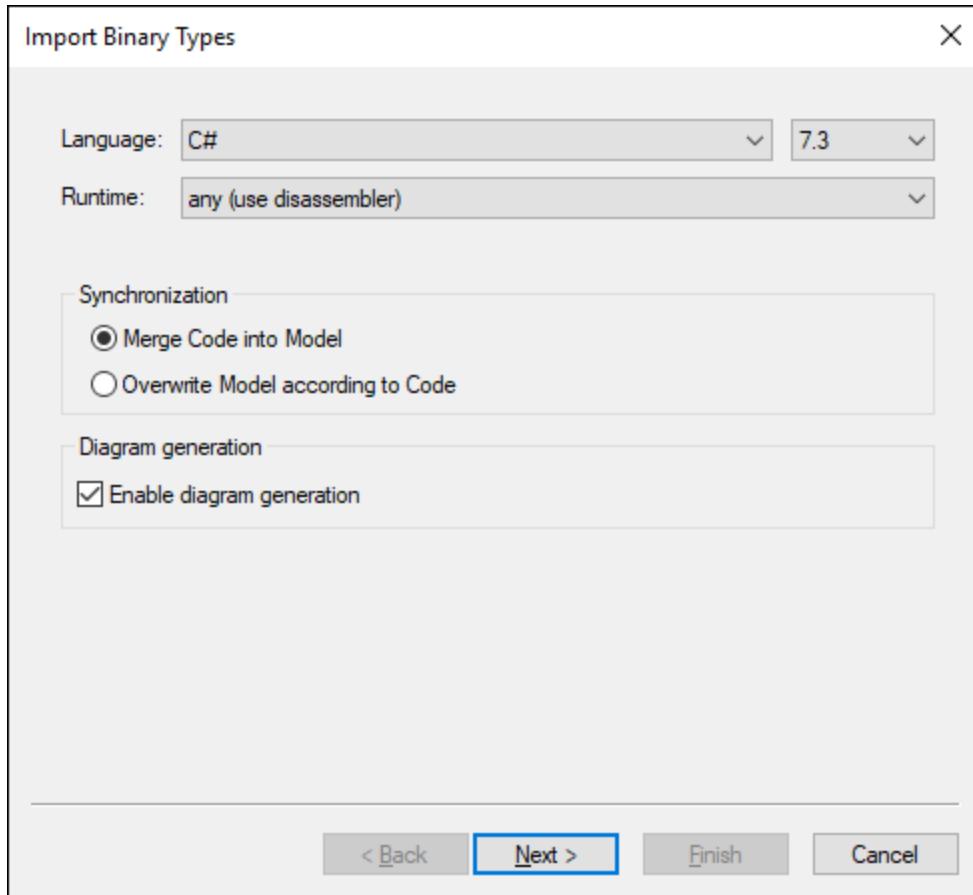
Finally, the option **suppress 'Attribute' suffix on attribute type names** removes the 'Attribute' suffix of an attribute type. For example, if this option is selected, an attribute type defined in the original code as `System.Xml.Serialization.XmlTypeAttribute` would be imported as `System.Xml.Serialization.XmlType`.

### 6.4.3 Example: Import .NET Assemblies

This example shows you how to import binary types from the .NET Global Assembly Cache (GAC) into a UModel C# project. The instructions are similar if you want to import binary types from a standalone .dll or .exe file. To find out how to import Java .class files, see [the next topic](#) 206.

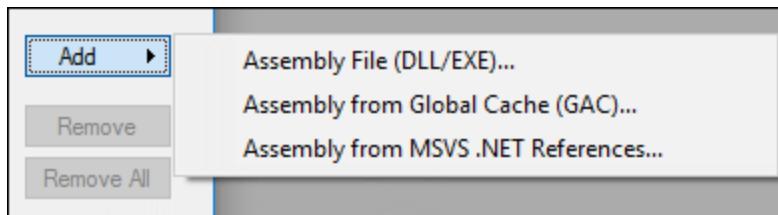
*To import binary files from the .NET Global Assembly Cache:*

1. Go the **Project** menu and click **Import Binary Types** (see screenshot below).



2. Choose the target language of the UModel project (C#, VB.NET, Java). In this example, C# is selected, since we are importing a .NET GAC assembly.
3. If you would like to set a specific language version for the imported UModel project, select it from the adjacent text box. In this example, C# 7.3 is selected.
4. Optionally, select a .NET runtime version from the **Runtime** drop-down list. The default option is *any (use disassembler)*. In this case, UModel will choose a reflection API that is most appropriate for the imported binary.
5. If you import binary types into a new project, select either **Merge Code into Model** or **Overwrite Model according to Code**.
6. Optionally, to generate class diagrams and package diagrams from the imported binary types, select the **Enable diagram generation** check box. If you select this option, more diagram generation options will be available in the next steps. See [Generating Class Diagrams](#) 392 and [Generating Package Diagrams](#) 401.
7. Click **Next**.

8. Click **Add | Assembly from Global Cache (GAC)** (see screenshot below). Note that the option **Assembly from Global Cache (GAC)** is only available for .NET Framework 2.x-4.x. The GAC is not relevant to .NET Core, .NET 5 and later versions. For more information, see [the Microsoft documentation](#). In order to import assembly files for .NET Core, .NET 5 and .NET 6, you will need to [extract the required files from the GAC](#). Then click **Add | Assembly File (DLL/EXE)**, select the assembly files manually and add them to the project.



9. Select an assembly from the dialog box. In this example, the *EventViewer* assembly is selected (see screenshot below).

The dialog box has a title bar 'Select Assemblies from Global Cache (GAC)...'. It contains a table with columns: Component Name, Version, Runtime, and Assembly Name. The 'EventViewer' row is highlighted with a light blue background. The table data is as follows:

Component Name	Version	Runtime	Assembly Name
EnvDTE100	10.0.0.0	v2.0.50727	EnvDTE100, Ve...
EnvDTE80	8.0.0.0	v1.0.3705	EnvDTE80, Ver...
EnvDTE90	9.0.0.0	v1.0.3705	EnvDTE90, Ver...
EnvDTE90a	9.0.0.0	v1.0.3705	EnvDTE90a, Ve...
EventViewer	10.0.0.0	v4.0.30319	EventViewer, V...
EventViewer.Resources	10.0.0.0	v4.0	EventViewer.R...

10. Select the types you would like to import and click **Next**. For more information about other options of the **Import Binary Selection** dialog box, see the notes below.
11. Select the import options as applicable. For more information, see [Import Binary Options](#).
12. If you enabled diagram generation in Step 6, click **Next** and configure the options applicable to diagram generation. Otherwise, click **Finish**.

UModel performs the conversion and displays a progress log in the **Messages** window. If the conversion of binary types is not possible, the error text may provide additional information. For example, the binary file you are trying to import is targeting a runtime newer than the one selected in the **Import Binary Types** dialog box. In this case, select a newer runtime version and try again.

#### Notes:

- The text box **Override of PATH variable...** in the **Import Binary Selection** dialog box is applicable only to Java. Optionally, paste here any Java class paths that must be queried in addition to those read from the `CLASSPATH` environment variable. Alternatively, click **Add** and browse for the required folders.
- The check box **use 'reflection only' context...** in the **Import Binary Selection** dialog box is applicable only when you import a C# or VB.NET binary. This is useful when importing a library which has dependencies that cannot be resolved or loaded. Selecting this check box will not execute any static initializer code, which might cause errors when importing.

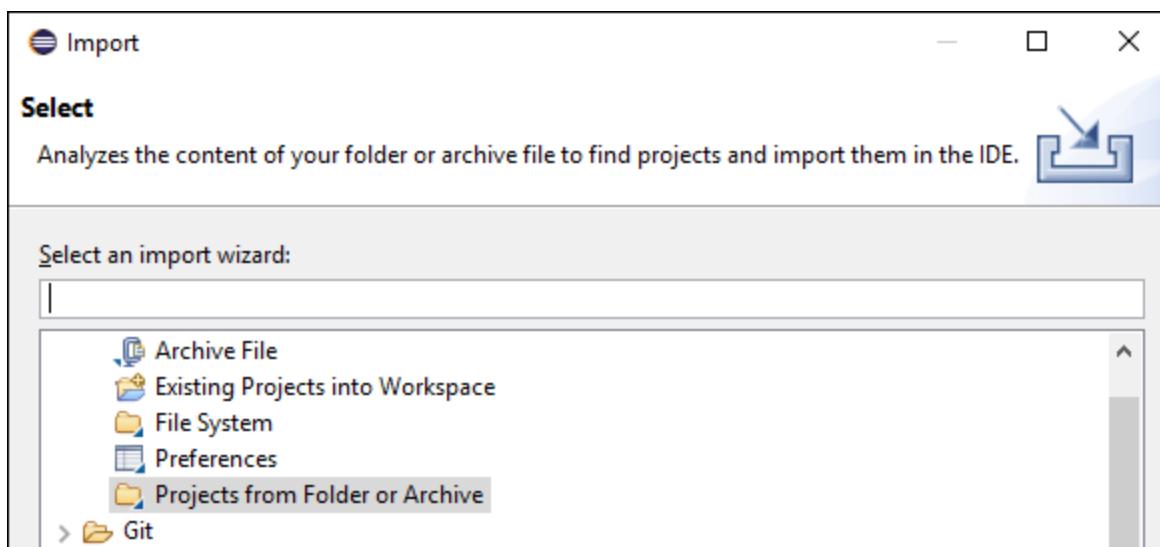
#### 6.4.4 Example: Import Java .class Files

This example shows you how to import compiled Java .class files into UModel. In this example, the source Java .class files originate from a tutorial Java project that was created with UModel, but you can also use other .class files as an alternative.

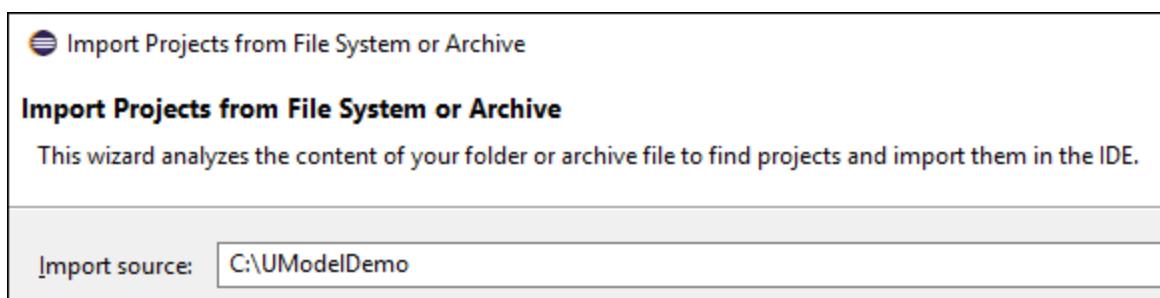
##### Compiling UModel-generated Java code (optional)

This section shows you how to compile a demo UModel-generated Java project with Eclipse. Note that this step is purely optional, the goal here is to obtain some compiled .class files. You can skip it if you already have readily available Java .class files. In this example, Eclipse is chosen as compilation environment for convenience; however, you can use the Java command line or some other Java development environment to achieve the same result.

1. If you haven't done that already, create a simple Java project with UModel, as shown in [Example: Generate Java Code](#). This is a very simple example consisting of a Java package with only one class. When you complete the example, the directory **C:\UModelDemo\src** will contain the required Java source code.
2. Run Eclipse. On the **File** menu, click **Import**.

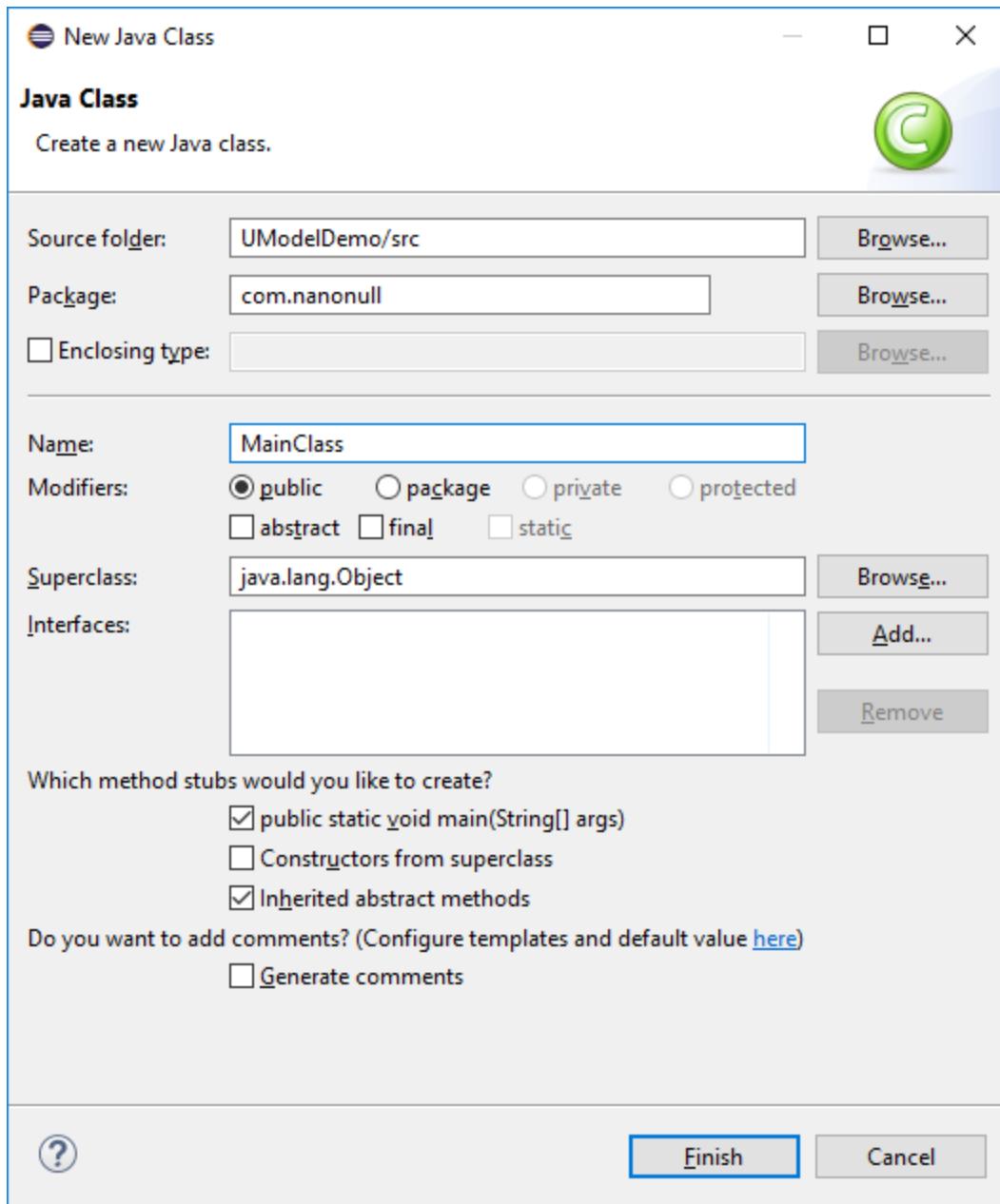


3. Select **Projects from Folder or Archive**, and click **Next**.



4. Enter **C:\UModelDemo** as directory, and click **Finish**.

5. Right-click the **com.nanonull** package in Eclipse's Package Explorer and select **New | Class** from the context menu.
6. Enter a class name ("MainClass", in this example), and select the **public static void main...** check box.



7. On the **Run** menu, click **Run**.

You have now finished compiling the UModel-generated Java project. The compiled .class files should now be available in the **bin** sub-directory of your project's directory.

Finally, take note of the Java version used for compilation—this is important if you intend to import binary types later. By default, if you did not modify your Eclipse project properties, it is likely that it was compiled with the default Java version available to Eclipse. To view the default Java version, do the following in Eclipse:

1. On the **Window** menu, click **Preferences**.
2. Click **Java**, and then click **Installed JREs**.

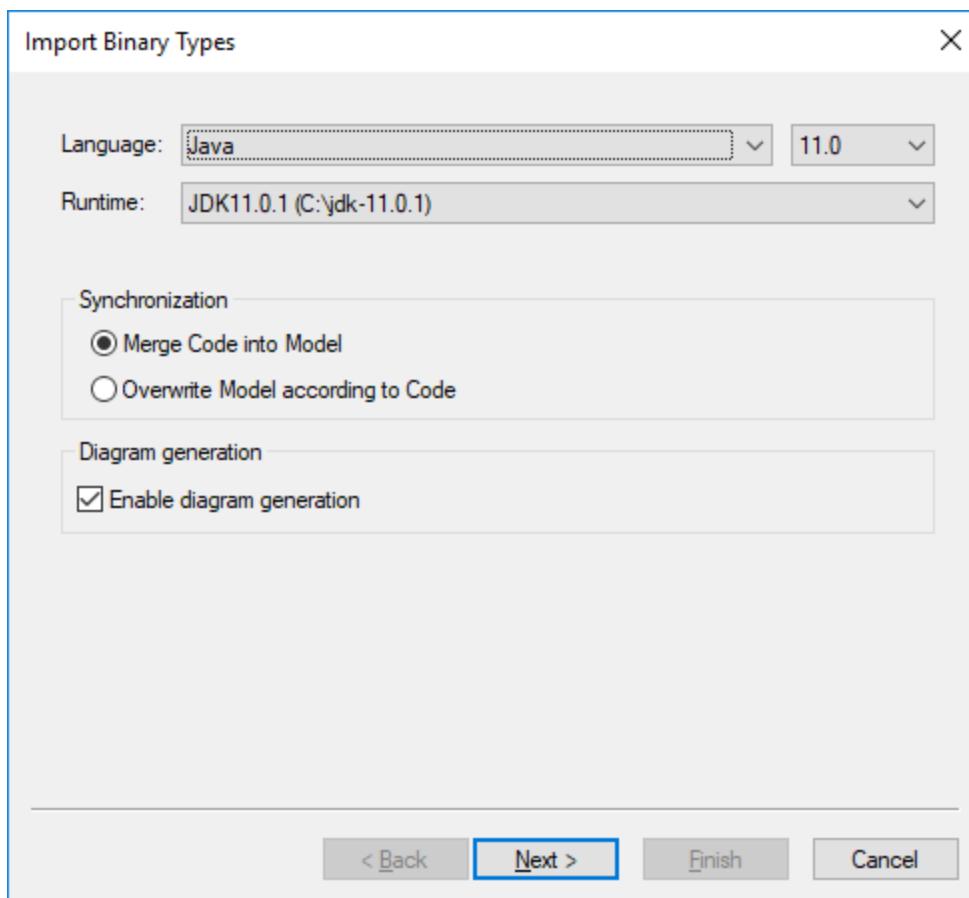
## Importing Java .class files

If you already have binary .class files such as the ones compiled previously, you can now proceed to importing them into UModel.

1. Create a new UModel project, or open an existing one. In this example, we are importing binary types into a new project.
2. If your project does not contain the Java JDK types already, do the following:
  - a. On the **Project** menu, click **Include subproject**.
  - b. Click the **Java** tab and select **Java JDK (types only)**.
  - c. Select **Include by reference** when prompted.

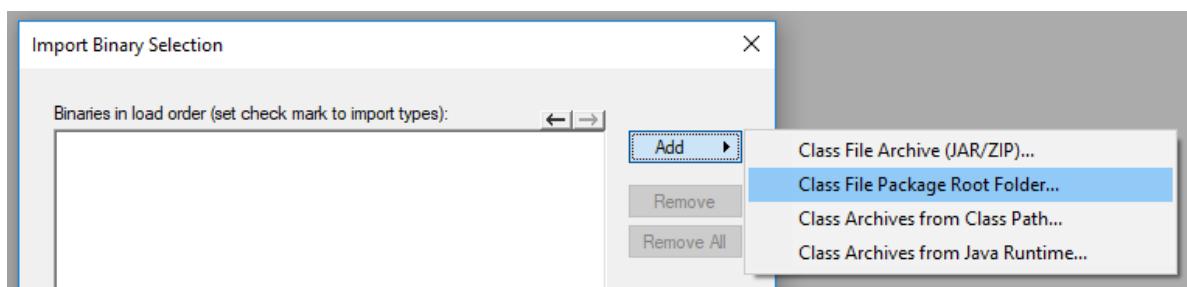
**Note:** This is an optional step which normally prevents the "Unknown externals" package from appearing in the project after the import is complete.

3. On the **Project** menu, click **Import Binary Types**.
4. Select **Java** as language, and the Java version in which the Java code was compiled (for example, 11.0).
5. Select the Java runtime to be used by UModel for extracting information from the binary files (the so-called "reflection"). The runtime version must be equal or newer than the Java version selected in the previous step.

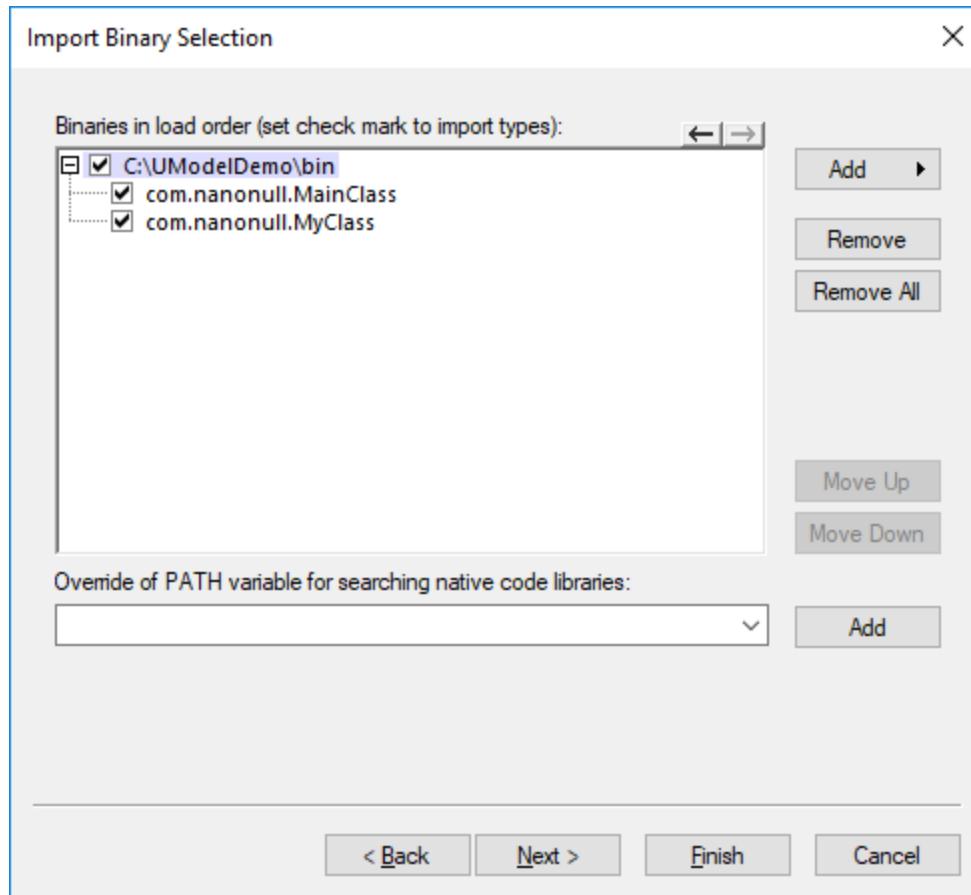


**Note:** The **Runtime** drop-down list contains only Java JDKs and JREs detected automatically. If your JDK or JRE is not listed, select the entry **Edit user java runtime locations** and browse for the directory where the respective distribution is installed on your machine, see [Adding Custom Java Runtimes](#) <sup>200</sup>.

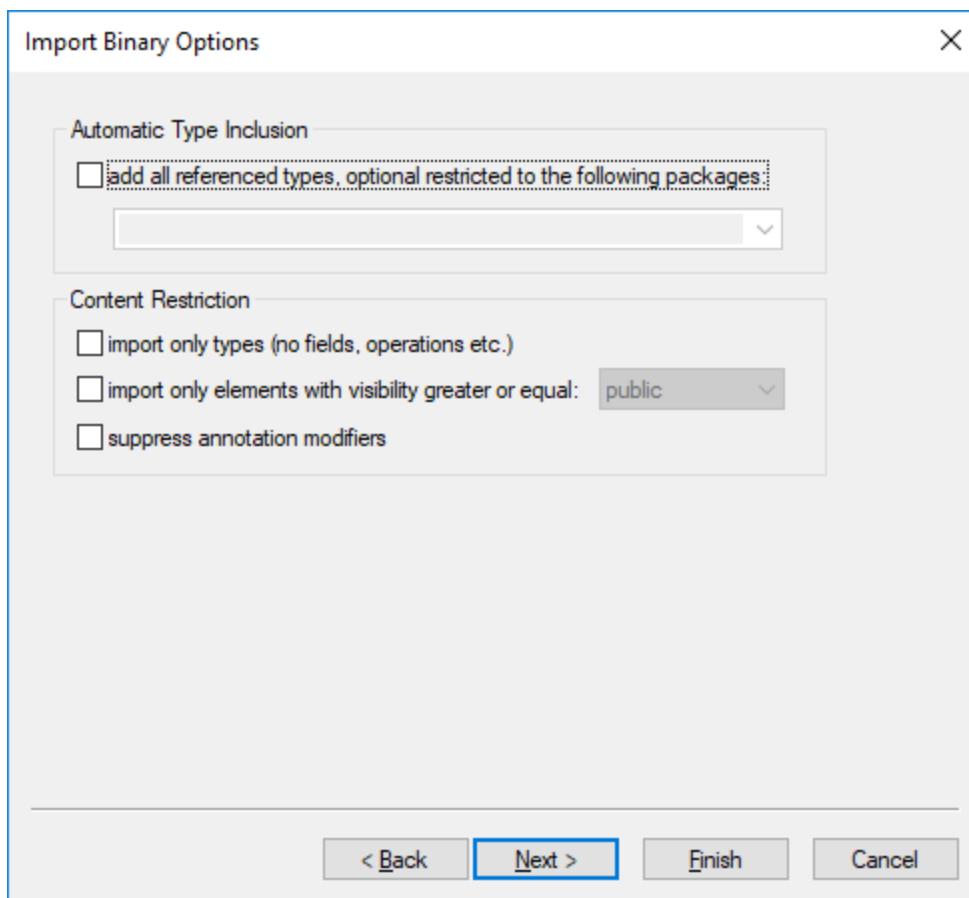
6. If you import binary types into a new project, select either **Merge Code into Model** or **Overwrite Model according to Code**. Otherwise, select **Merge code into Model**.
7. Optionally, to generate class diagrams and package diagrams from the imported binary types, select the **Enable diagram generation** check box. If you select this option, more diagram generation options are available in subsequent steps, see also [Generating Class Diagrams](#) <sup>392</sup> and [Generating Package Diagrams](#) <sup>401</sup>.
8. Click **Next**.



9. In this example, we are importing Java .class files from a package root. Select **Add | Class File Package Root Folder**, and browse for the **C:\UModelDemo\bin** directory. If this directory does not exist, make sure to compile the project first, as shown in the first part of this tutorial.



10. Select the classes to be imported, and click **Next**.



11. Select the import options as applicable, see [Import Binary Options](#)<sup>200</sup>.
12. If you enabled diagram generation in an earlier step, click **Next** and configure the options applicable to diagram generation. Otherwise, click **Finish**.

UModel performs the conversion and displays a progress log in the **Messages** window. If the conversion of binary types is not possible, the error text may provide additional information. For example, the binary file you are trying to import is targeting a runtime newer than the one selected in the **Import Binary Types** dialog box. In this case, select a newer runtime version and try again.

## 6.5 Synchronizing the Model and Source Code

You can synchronize the model and code in either direction, and at different levels (for example, project, package or class).

When UModel (Enterprise or Professional) runs as an Eclipse or Visual Studio plug-in, synchronization between model and code takes place automatically. Manual synchronization is possible at the project level; the option to update individual classes or packages is not available.

When you right-click an element in the Model Tree (for example, a class), the context menu displays the code synchronization or merging commands under the **Code Engineering** menu item:

- **Merge Program Code from UModel \*\*\***
- **Merge UModel \*\*\* from Program Code**

\*\*\* is a Project, Package, Component, Class, and so on, depending on your current selection.

Depending on the settings you have defined from **Project | Synchronization Settings**, the alternative name of these two commands may be:

- **Overwrite Program Code from UModel \*\*\***
- **Overwrite UModel \*\*\* from Program Code**

To update the entire project (but not classes, packages, or other local elements), you can also use the following commands on the **Project** menu of UModel:

- **Merge (or Overwrite) Program Code from UModel Project**
- **Merge (or Overwrite) UModel Project from Program Code**

For convenience, any of the commands listed above will be generically referred to as "code synchronization commands" further in this topic.

**To synchronize at the project or Root package level, do one of the following:**

- Right-click the Root package in the Model Tree, and select the required code synchronization command.
- On the **Project** menu, click the required code synchronization command.

**To synchronize at package level:**

1. Use **Shift**, or **Ctrl + Click** to select the package(s) you want to merge.
2. Right-click the selection, and select the required code synchronization command.

**To synchronize at class level:**

1. Use **Shift**, or **Ctrl + Click** to select the classes(s) you want to merge.
2. Right-click the selection, and click the required code synchronization command.

To avoid undesired results when synchronizing the model and code, consider the following scenarios:

On the <b>Project</b> menu, click <b>Overwrite UModel Project from Program Code</b> .	<ul style="list-style-type: none"><li>• This checks all directories (project files) of all different code languages you have defined in your project.</li><li>• New files are identified and added to the project.</li><li>• An entry "Collecting source files in (...)" appears in the Messages window.</li></ul>
Right-click a class or interface in the Model Tree and select <b>Code Engineering   Overwrite UModel Class from Program Code</b> .	<ul style="list-style-type: none"><li>• This updates only the selected class (interface) of your project.</li><li>• If the source code contains classes that are new or modified classes since the last synchronization, those changes will not be added to the model.</li></ul>
Right-click a Component in the Model Tree (within the Component View package) and select <b>Code Engineering   Overwrite UModel Component from Program Code</b> .	<ul style="list-style-type: none"><li>• This updates the corresponding directory (or project file) only.</li><li>• New files in the directory (project file) are identified and added to the project.</li><li>• An entry "Collecting source files in (...)" appears in the Message window.</li></ul>

**Note:** When synchronizing code, you might be prompted to update your UModel project before synchronization. This occurs when you open UModel projects created before the latest release. Click **Yes** to update your project to the latest release format, and save your project file. The notification message will not occur once this has been done.

## 6.5.1 Synchronization Tips

### Renaming of classifiers and reverse engineering

The process described below applies to the standalone application as well as to the plug-in versions (Visual Studio or Eclipse) when reverse engineering or automatic synchronization takes place.

Renaming a classifier in the code window of your programming application causes it to be deleted and re-inserted as new classifier in the **Model Tree**.

The new classifier is only re-inserted in those modeling diagrams that are automatically created during the reverse-engineering process, or when generating a diagram using the **Show in new Diagram | Content** option. The new classifier is inserted at a default position on the diagram, that will likely differ from the previous location.

See also [Refactoring code and synchronization](#)<sup>215</sup>.

## Automatic generation of ComponentRealizations

UModel is capable of automatically generating ComponentRealizations during the code engineering process. ComponentRealizations are only generated where it is absolutely clear to which component a class should be assigned:

- Only one Visual Studio project file exists in the .ump project.
- Multiple Visual Studio projects exist but their classes are completely separate in the model.

### To enable automatic generation of ComponentRealizations:

1. Open the menu item **Tool | Options**.
2. Click the **Code Engineering** tab and activate the **Generate missing ComponentRealizations** option.

Automatic ComponentRealizations are created for a **Classifier** that can be assigned one (and only one) Component

- without any ComponentRealizations, or
- contained in a code language namespace.

The way the Component is found differs for the two cases.

Component representing a code project file (property "**projectfile**" set)

- if there is ONE Component having/realizing classifiers in the containing package
- if there is ONE Component having/realizing classifiers in a subpackage of the containing package (top down)
- if there is ONE Component having/realizing classifiers in one of the parent packages (bottom up)
- if there is ONE Component having/realizing classifiers in a subpackage of one of the parent packages (top down)

Component representing a directory (property "**directory**" set)

- if there is ONE Component having/realizing classifiers in the containing package
- if there is ONE Component having/realizing classifiers in one of the parent packages (bottom up)

Notes:

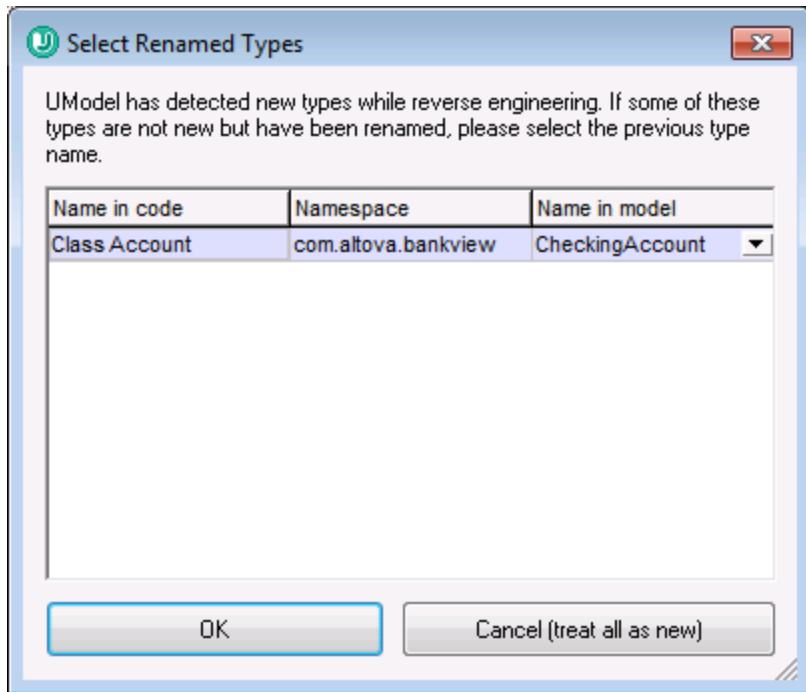
- The option "**Code Engineering | Generate missing ComponentRealizations**" has to be set.
- As soon as ONE viable Component is found during one of the above steps, this Component is used and the remaining steps are ignored.

Error/Warnings:

- If no viable Component was found, a warning is generated (message log)
- If more than one viable Component was found, an error is generated (message log)

## 6.5.2 Refactoring Code and Synchronization

When refactoring code, it is often the case that class names are changed or updated in the code. If it detects that new types have been added or renamed during reverse engineering, UModel (version 2009 or later) displays a dialog box. The new types are listed in the "Name in code" column while the assumed original type name is listed in the "Name in model" column. UModel attempts to determine the original name by relying on namespace, class content, base classes and other data.



If a class was renamed, select the previous class name using the combo box in the "Name in model" column, e.g. C1. This ensures that all related data are retained and the code engineering process remains accurate.

### Changing class names in the model and regenerating code

Having created a model and generated code from it, it is possible that you might want to make changes to the model again before going through the synchronization process.

E.g. You decide that you want to change the class names before generating code the second time round. As you previously assigned a file name to each class, in the "code file name" field of the Properties window, the new class and file name would now be out of sync.

UModel prompts if you want the code file name to agree with the new class name, when you start the synchronization process. Note that you also have the option to change the class constructors as well.

### Round-trip engineering and relationships between modeling elements

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

dialog box. Associations are displayed for those elements where the attributes type is set, and the referenced "type" modeling element is in the same diagram.

InterfaceRealizations as well as Generalizations are all automatically shown in the diagram when updating model from code.

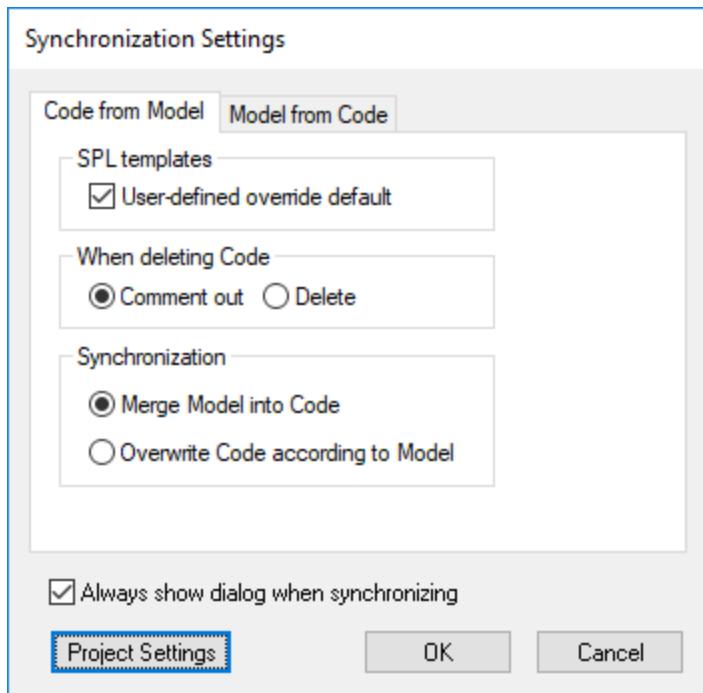
### 6.5.3 Code Synchronization Settings

The code synchronization settings are relevant in the following scenarios:

- When program code is generated from the model (that is, when either the command **Project | Merge Program Code from UModel Project** or the command **Project | Overwrite Program code from UModel Project** is run)
- When source code is imported into the model (that is, when either the command **Project | Merge UModel Project from Program Code** or the command **Project | Overwrite UModel Project from Program Code** is run)
- When automatic synchronization takes place in either direction (this applies to UModel Enterprise and Professional Editions when UModel runs as a Visual Studio or Eclipse plug-in).

To change the code synchronization settings:

- On the **Project** menu, click **Synchronization Settings**.



Synchronization Settings dialog box

By default, the Synchronization Settings dialog box will be displayed automatically every time when you initiate any of the code synchronization commands. To disable this behaviour, clear the check box **Always show dialog when synchronizing**.

The available options are grouped into two tabs:

- **Code from Model** (options in this tab are applicable when program code is generated from the model)
- **Model from Code** (options in this tab are applicable when program code is imported into the model).

Option	Description
<b>SPL templates</b>	<p>This option is applicable only when generating program code. Select the check box <b>User-defined override default</b> check box if you have created custom Spy Programming Language (SPL) templates that should override the default ones supplied with UModel (see also <a href="#">SPL Templates</a><sup>185</sup>).</p>
<b>When deleting code</b>	<p>This option is applicable only when generating program code. Select whether program code should be deleted or commented out during synchronization (assuming the relevant objects no longer exist in the model).</p>
<b>Synchronization</b>	<p>This option is applicable both when generating and importing program code. It lets you specify whether changes should be merged as opposed to being overwritten. Assuming that code has been generated once from a model, and changes have since been made to both model and code, for example:</p> <ul style="list-style-type: none"> <li>• A new class X has been added in UModel</li> <li>• A new class Y has been added to the external code,</li> </ul> <p><b>Merge Model into Code</b> means that:</p> <ul style="list-style-type: none"> <li>• The newly added class Y in the external code is retained</li> <li>• The newly added class X, from UModel, is added to the code.</li> </ul> <p><b>Overwrite Code according to Model</b> means that:</p> <ul style="list-style-type: none"> <li>• The newly added class Y in the external code is deleted (or commented out, depending on the current settings)</li> <li>• The newly added class X, from UModel, is added to the code.</li> </ul> <p><b>Merge Code into Model</b> means that:</p> <ul style="list-style-type: none"> <li>• The newly added class X in UModel is retained</li> <li>• The newly added class Y, from the external code, is added to the model</li> </ul> <p><b>Overwrite Model according to Code</b> means that:</p> <ul style="list-style-type: none"> <li>• The newly added class X in UModel is deleted (or commented out, depending on the current settings)</li> <li>• The newly added class Y, from the external code, is added to the model.</li> </ul>
<b>Project settings</b>	<p>Opens the Project Settings dialog box, where you can modify the code engineering settings applicable to each language. For reference to all settings, see <a href="#">Code Import Options</a><sup>189</sup> and <a href="#">Code Generation Options</a><sup>169</sup>, respectively.</p>

Option	Description
	<p>The Project Settings dialog box can also be triggered from the menu command <b>Project   Project Settings</b>. Note that the project settings in this dialog box are global (they are saved together with the project and are applicable on any workstation where the UModel project is open) whereas the options you define from <b>Tools   Options</b> are local (they are applicable only to the current installation of UModel).</p>

## 6.6 UModel Element Mappings

This section illustrates how UModel elements map to elements (constructs) in various programming languages (C#, Java, VB.NET), as well as to databases and XML schemas. The mappings are grouped by language, and are applicable when importing code into model, or when generating code from model.

- [C# Mappings](#)<sup>219</sup>
- [VB.NET Mappings](#)<sup>239</sup>
- [Java Mappings](#)<sup>253</sup>
- [XML Schema Mappings](#)<sup>259</sup>

### 6.6.1 C# Mappings

The table below shows the one-to-one correspondence between:

- UModel elements and C# code elements, when outputting model to code
- C# code elements and UModel model elements, when inputting code into model

#### C# Project

C#		UModel	
Project	projectfile	projectfile	Component
	directory	directory	

#### C# Namespace

C#		UModel	
Namespace	name	name	Package <<namespace>>

#### C# Class

C#		UModel			
Class	name		name		Class
	modifiers	internal	visibility	package	
		protected internal		protected <<internal>>	
		public		public	
		protected		protected	
		private		private	
	sealed		leaf		

C#			UModel				
Field	modifiers	abstract	abstract			Property visibility	
		static	<<static>>				
		unsafe	<<unsafe>>				
		partial	<<partial>>				
		new	<<new >>				
		filename	code file name				
		associated projectfile/directory	ComponentRealization				
		base types	Generalization, InterfaceRealization(s)				
		attribute sections	<<attributes>>				
		doc comments	Comment(->Documentation)				
	name	internal	visibility	name		Property	
		protected internal		package			
		public		protected <<internal>>			
		protected		public			
		private		protected			
		static		private			
		readonly		static			
		volatile		readonly			
		unsafe		<<volatile>>			
		new		<<unsafe>>			
	type	type	type			Property <<const>>	
		type dimensions	multiplicity				
		type pointer	type modifier				
		nullable	<<nullable>>				
		default value	default				
	attribute sections	attribute sections	<<attributes>>			Property <<const>>	
		doc comments	Comment(->Documentation)				
		name	name				
	Constant	modifiers	internal	visibility	package		

C#			UModel								
		protected internal		protected <>internal>>							
		public		public							
		protected		protected							
		private		private							
		new	<>new >>								
	type		type								
	type dimensions		multiplicity								
	type pointer		type modifier								
	nullable		<>nullable>>								
	default value		default								
	attribute sections		<>attributes>>								
	doc comments		Comment(->Documentation)								
Method	name		name		Operation						
	modifiers	internal	visibility	package							
		protected internal		protected <>internal>>							
		public		public							
		protected		protected							
		private		private							
	static		static								
	abstract		abstract								
	sealed		leaf								
	override		<>override>>								
	partial		<>partial>>								
	virtual		<>virtual>>								
	new		<>new >>								
	unsafe		<>unsafe>>								
	attribute sections		<>attributes>>								
	doc comments		Comment(->Documentation)								
	implemented interfaces		implements								
	type		direction	return	Parameter						

C#				UModel										
<b>Parameter</b>	<b>Parameter</b>	name		name		<b>Template Parameter</b>								
		modifiers	ref	direction	inout									
		out			out									
		varArgList												
		type		type										
		type dimensions		multiplicity										
	<b>Type Parameter</b>	type pointer		type modifier										
		this		<<this>>										
		nullable		<<nullable>>										
		predefine d constraint	struct	<<ValueTypeConstraint >>										
				<<ReferenceTypeConst raint>>										
			new ()	<<ConstructorConstrain t>>										
		attribute sections		<<attributes>>										
<b>Construct or</b>	name			name		<b>Operation &lt;&lt;constructor&gt;&gt;</b>								
	<b>modifiers</b>	internal		visibility	package									
		protected internal			protected <<internal>>									
		public			public									
		protected			protected									
		private			private									
		static			static									
		unsafe			<<unsafe>>									
	attribute sections				<<attributes>>									
	doc comments				Comment(->Documentation)									
	<b>Parameter</b>	name		name		<b>Parameter</b>								
		modifiers	ref	direction	inout									

C#				UModel									
			params	varArgList									
		type		type									
		type dimensions		multiplicity									
		type pointer		type modifier									
		nullable		<<nullable>>									
Destructor	name			name			Operation <<destructor>>						
	modifiers	private		visibility	private								
		unsafe		<<unsafe>>									
	attribute sections			<<attributes>>									
	doc comments			Comment(->Documentation)									
Property	name			name			Operation <<property>>						
	modifiers	internal		visibility	package								
		protected internal			protected <<internal>>								
		public			public								
		protected			protected								
		private			private								
		static			static								
		abstract			abstract								
		sealed			leaf								
		override			<<override>>								
	attribute sections	virtual			<<virtual>>								
		new			<<new >>								
		unsafe			<<unsafe>>								
		attribute sections			<<attributes>>								
		doc comments			Comment(->Documentation)								
		type			direction	return	Parameter						
		type dimensions			multiplicity								
	nullable			<<nullable>>									
	Get	modifiers	internal	visibility	internal	<<GetAcc							

C#					UModel															
Set Accessor	modifiers	protected internal	internal	visibility	internal	<<SetAcc essor>>														
			protected		protected															
			private		private															
			Set Accessor		internal															
	modifiers	protected internal	protected		protected															
			internal		internal															
			protected		protected															
			private		private															
Operator	name			name			Operation <<operato r>>	Parameter												
	modifiers	public		visibility	public															
		static		static																
		unsafe		<<unsafe>>																
	attribute sections			<<attributes>>																
	doc comments			Comment(->Documentation)																
	type			direction	return	Parameter														
	Parameter	name		name																
		modifier	params	varArgList																
		type		type																
		type dimensions		multiplicity																
		type pointer		type modifier																
		nullable		<<nullable>>																
Indexer	name ("this")			name ("this")			Operation <<indexer >>													
	modifiers	internal		visibility	package															
		protected internal			protected <<internal>>															
		public			public															
		protected			protected															
		private			private															
		static			static															
		abstract			abstract															
		sealed			leaf															

C#				UModel				
		override virtual new unsafe	override	<<override>>				
			virtual	<<virtual>>				
			new	<<new >>				
			unsafe	<<unsafe>>				
		attribute sections			<<attributes>>			
		doc comments			Comment(->Documentation)			
		type			direction	return	Parameter	
	Parameter	name modifier params type type dimensions type pointer nullable	name		name			
			modifier	params	varArgList			
			type		type			
			type dimensions		multiplicity			
			type pointer		type modifier			
			nullable		<<nullable>>			
	Get Accessor	modifiers internal protected internal protected private	internal protected internal protected private	visibility	internal protected internal protected private	<<GetAccesso r>>	Operation <<event>>	
	Set Accessor	modifiers internal protected internal protected private	internal protected internal protected private	visibility	internal protected internal protected private	<<SetAccesso r>>		
	Event	name		name				
		modifiers internal protected internal public protected private static	internal protected internal public protected private static	visibility	package protected <<internal>> public protected private	<<event>>		

C#			UModel				
			sealed	leaf			
			override	<<override>>			
			virtual	<<virtual>>			
			new	<<new >>			
			unsafe	<<unsafe>>			
		attribute sections		<<attributes>>			
		doc comments		Comment(->Documentation)			
		type		direction	return	Parameter	
		type dimensions		multiplicity			
		nullable		<<nullable>>			
		Add Accessor		<<AddRemoveAccessor>>			
		Remove Accessor					
	Type Parameter	name		name		Template Parameter	
		constraint		constraining classifier			
		predefine d constraint	struct	<<ValueTypeConstraint>>			
			class	<<ReferenceTypeConstraint>>			
		new ()		<<ConstructorConstraint>>			
		attribute sections		<<attributes>>			

## C# Struct

C#			UModel				
Struct	name		name				
	modifiers	internal		visibility	package		
		protected internal			protected <<internal>>		
		public			public		
		protected			protected		
		private			private		
		unsafe			<<unsafe>>		
		partial			<<partial>>		
		new			<<new >>		

C#			UModel				
Field	filename		code file name				
	associated projectfile/directory		ComponentRealization				
	base types		InterfaceRealization(s)				
	attribute sections		<<attributes>>				
	doc comments		Comment(>->Documentation)				
	modifiers	name		name			
		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 >>				

C#			UModel				
		type	type				
		type dimensions	multiplicity				
		type pointer	type modifier				
		nullable	<<nullable>>				
		default value	default				
		attribute sections	<<attributes>>				
		doc comments	Comment(->Documentation)				
FixedSize Buffer	name		name		Property <<fixed>>		
	modifiers	internal	visibility	package			
		protected internal		protected <<internal>>			
		public		public			
		protected		protected			
		private		private			
	unsafe		<<unsafe>>				
	new		<<new >>				
	type		type				
	type pointer		type modifier				
	nullable		<<nullable>>				
	buffer size		default				
	attribute sections		<<attributes>>				
	doc comments		Comment(->Documentation)				
Method	name		name		Operation		
	modifiers	internal	visibility	package			
		protected internal		protected <<internal>>			
		public		public			
		protected		protected			
		private		private			
	static		static				
	abstract		abstract				
	sealed		leaf				

C#			UModel							
			override	<<override>>						
			partial	<<partial>>						
			virtual	<<virtual>>						
			new	<<new >>						
			unsafe	<<unsafe>>						
			attribute sections	<<attributes>>						
			doc comments	Comment(->Documentation)						
			implemented interfaces	implements						
			type	direction	return	Parameter				
Parameter	name		name							
	modifiers	ref	direction	inout						
		out		out						
		params	varArgList							
	type		type							
	type dimensions		multiplicity							
	type pointer		type modifier							
	this		<<this>>							
	nullable		<<nullable>>							
	Type Parameter	name		name		Template Parameter				
		constraint		constraining classifier						
		predefined constraint	struct	<<ValueTypeConstraint>>						
			class	<<ReferenceTypeConstraint>>						
		new()		<<ConstructorConstraint>>						
	attribute sections			<<attributes>>						
Constructor	name			name		Operation <<constructor>>				
	modifiers	internal		visibility	package					
		protected internal			protected <<internal>>					
		public		public						

C#				UModel						
<b>Parameter</b>	protected private static unsafe				protected					
					private					
		static		static						
		unsafe		<<unsafe>>						
	attribute sections			<<attributes>>						
	doc comments			Comment(->Documentation)						
	name modifiers ref out params type type dimensions type pointer nullable	name	name		Parameter					
			direction	inout						
		ref				out				
		varArgList								
			type			type				
			type dimensions			multiplicity				
			type pointer			type modifier				
<b>Destructor</b>	name		name		Operation <<destructor>>					
	modifiers	private		visibility	private					
		unsafe		<<unsafe>>						
	attribute sections			<<attributes>>						
	doc comments			Comment(->Documentation)						
<b>Property</b>	name			name		Operation <<property>>				
	modifiers	internal		visibility	package					
		protected internal			protected <<internal>>					
		public			public					
		protected			protected					
		private			private					
	static		static							
	abstract		abstract							
	sealed		leaf							
	override		<<override>>							
	virtual		<<virtual>>							

C#				UModel							
				<<new >>							
		unsafe		<<unsafe>>							
attribute sections				<<attributes>>							
doc comments				Comment(->Documentation)							
type				direction	return	Parameter					
type dimensions				multiplicity							
nullable				<<nullable>>							
Get Accessor	modifiers	internal		visibility	internal		<<GetAccessor>>				
		protected internal			protected internal						
		protected			protected						
		private			private						
		Set Accessor	modifiers	internal		<<SetAccessor>>					
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>>						

C#				UModel								
<b>Indexer</b>	<b>Indexer</b>	name ("this")		name ("this")		<b>Operation &lt;&lt;indexer &gt;&gt;</b>						
		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	<b>Parameter</b>					
		<b>Parameter</b>	name		name							
			modifier	params	varArgList							
			type		type							
			type dimensions		multiplicity							
			type pointer		type modifier							
			nullable		<<nullable>>							
		Get Accessor	modifiers	internal	visibility	internal	<<GetAccess or>>					
				protected internal		protected internal						
				protected		protected						
				private		private						
		Set Accessor	modifiers	internal	visibility	internal	<<SetAccess or>>					
				protected internal		protected internal						
				protected		protected						

C#				UModel									
			private		private								
Event	name		name				Operation "><<event>>						
	modifiers	internal		visibility	package								
		protected internal			protected <<internal>>								
		public			public								
		protected			protected								
		private			private								
		static			static								
		abstract			abstract								
		sealed			leaf								
		override			<<override>>								
Type Parameter	virtual		<<virtual>>				Template Parameter						
	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												
	name			name									
	constraint			constraining classifier									
	predefine d constraint	struct		<<ValueTypeConstraint>>									
		class		<<ReferenceTypeConstraint>>									
		new ()		<<ConstructorConstraint>>									
	attribute sections			<<attributes>>									

## C# Interface

C#			UModel				
Interface	name		name		Interface		
	modifiers	internal		visibility			
		protected internal					
		public					
		protected					
		private					
	unsafe			<>unsafe>>			
	new		<>new >>				
	filename			code file name			
	associated projectfile/directory			ComponentRealization			
	base types			Generalization(s)			
	attribute sections			<>attributes>>			
	doc comments			Comment(->Documentation)			
Method	name		name		Operation		
	modifiers	public		visibility	public		
		new					
		unsafe					
	attribute sections			<>attributes>>			
	doc comments			Comment(->Documentation)			
	type		direction	return	Parameter		
	Parameter	name		name			
		modifiers	ref	direction	inout		
			out		out		
			params	varArgList			
		type		type			
		type dimensions		multiplicity			
		type pointer		type modifier			

C#				UModel									
<b>Type Parameter</b>	this	<<this>>		name	<<name>>		<b>Template Parameter</b>						
		nullable			<<nullable>>								
	constraint	name			name								
		constraint			constraining classifier								
		predefine d constraint	struct		<<ValueTypeConstraint >>								
			class		<<ReferenceTypeConst raint>>								
		new ()			<<ConstructorConstrain t>>								
	attribute sections				<<attributes>>								
	<b>Property</b>	name			name		<b>Operation &lt;&lt;property&gt;&gt;</b>						
		modifiers	public		visibility	public							
			new			<<new >>							
			unsafe		<<unsafe>>								
		attribute sections			<<attributes>>								
		doc comments			Comment(->Documentation)								
		type			direction	return	<b>Parameter</b>						
		type dimensions											
		nullable			<<nullable>>								
	<b>Get Accessor</b>	modifiers	internal	visibility	internal	<<GetAcc essor>>							
			protected internal		protected internal								
	<b>Set Accessor</b>	modifiers	internal	visibility	internal	<<SetAcc essor>>							
			protected		protected								
			private		private								
<b>Indexer</b>	name ("this")			name ("this")			<b>Operation &lt;&lt;indexer &gt;&gt;</b>						
	modifiers	public		visibility	public								

C#				UModel						
			new	<<new >>						
			unsafe	<<unsafe>>						
		attribute sections		<<attributes>>						
		doc comments		Comment(->Documentation)						
		type		direction	return	Parameter				
		Parameter	name		name					
			modifier	params	varArgList					
			type		type					
			type dimensions		multiplicity					
			type pointer		type modifier					
		nullable		<<nullable>>						
	Get Accessor	modifiers	internal	visibility	internal	<<GetAccesso r>>				
			protected internal		protected internal					
			protected		protected					
			private		private					
	Set Accessor	modifiers	internal	visibility	internal	<<SetAccesso r>>				
			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		<<AddRemoveAccesso r>>						

C#			UModel			
Type Parameter	Remove Accessor				Template Parameter	
	name		name			
	constraint		constraining classifier			
	predefine d constraint	struct	<<ValueTypeConstraint>>			
		class	<<ReferenceTypeConstraint>>			
		new()	<<ConstructorConstraint>>			
	attribute sections		<<attributes>>			

## C# Delegate

C#			UModel						
Delegate	name		name		Class <<delegat e>>				
	modifiers	internal		visibility	package				
		protected internal			protected <<internal>>				
		public			public				
		protected			protected				
		private			private				
		unsafe			<<unsafe>>				
		new			<<new>>				
	filename			code file name					
	associated projectfile/directory			ComponentRealization					
	attribute sections			<<attributes>>					
	doc comments			Comment(->Documentation)					
	type			direction	return	Parameter			
	Parameter	name		name					
		modifiers	ref	direction	inout				
			out		out				
			params	varArgList					
		type		type					
		type dimensions		multiplicity					
		type pointer		type modifier					

C#				UModel			
Type Parameter	nullable			<<nullable>>			
	name			name			Template Parameter
	constraint			constrainting classifier			
	predefined constraint	struct		<<ValueTypeConstraint>>			
		class		<<ReferenceTypeConstraint>>			
		new()		<<ConstructorConstraint>>			
	attribute sections			<<attributes>>			

## C# Enum

C#			UModel				
Enum	name		name		Enumeration		
	modifiers	internal	visibility	package			
		protected internal		protected <<internal>>			
		public		public			
		protected		protected			
		private		private			
		new	<<new>>				
	filename		code file name				
	associated projectfile/directory		ComponentRealization				
	base type		type	<<BaseType>>			
	attribute sections		<<attributes>>				
	doc comments		Comment(<->Documentation)				
	Enum Constant	name	name	Enumeration Literal			
		default value	default				
		attribute sections	<<attributes>>				

C#			UModel		
		doc comments	Comment(->Documentation)		

### C# Parameterized Type

C#		UModel	
Parameterized Type		Anonymous Bound Element	

## 6.6.2 VB.NET Mappings

The table below shows the one-to-one correspondence between:

- UModel elements and VB.NET code elements, when outputting model to code
- VB.NET code elements and UModel model elements, when inputting code into model

VB.NET			UModel			
Project	projectfile		projectfile		Component	
	directory		directory			
NameSpace	name			name		Package <<nameSpace>>
Class	name		name		Class	
	modifiers	Friend	visibility	package		
		Protected Friend		protected <<Friend>>		
		Public		public		
		Protected		protected		
		Private		private		
		NotInheritable		leaf		
		MustInherit		abstract		
		Partial		<<Partial>>		
	Shadow s		<<Shadow s>>			
	filename		code file name			
	associated projectfile/directory		ComponentRealization			
	base types		Generalization, InterfaceRealization(s)			

VB.NET			UModel			
Field	attribute sections		<<Attributes>>			
	doc comments		Comment(->Documentation)			
	name		name		Property	
	modifiers	Friend	visibility	package		
		Protected Friend		protected <<Friend>>		
		Public		public		
		Protected		protected		
		Private		private		
		Shared	static			
		ReadOnly	readonly			
		Shadow s	<<Shadow s>>			
	type		type			
	type dimensions		multiplicity			
	nullable		<<Nullable>>			
	default value		default			
	attribute sections		<<Attributes>>			
	doc comments		Comment(->Documentation)			
Constant	name		name		Property <<Const>>	
	modifiers	Friend	visibility	package		
		Protected Friend		protected <<Friend>>		
		Public		public		
		Protected		protected		
		Private		private		
		Shadow s	<<Shadow s>>			
	type		type			
	type dimensions		multiplicity			
	nullable		<<Nullable>>			
	default value		default			
	attribute sections		<<Attributes>>			
	doc comments		Comment(->Documentation)			

VB.NET			UModel				
Method	name	name		Operation			
		modifiers	Friend				
			Protected Friend				
			Public				
			Protected				
			Private				
			Shared				
			MustOverride				
			NotOverridable				
			Overrides				
			Overridable				
			Partial				
			Shadows				
			Overloads				
		attribute sections		<>Attributes>>			
		doc comments		Comment(->Documentation)			
		implemented interfaces		implements			
		type (function)		direction	return		
	Parameter	modifiers	name		Parameter		
			ByRef				
			ByVal				
			ParamArray				
			Optional				
		type		type			
		type dimensions		multiplicity			
		nullable		<>Nullable>>			
	Type Parameter	name		name			
		constraint		constraining classifier			
		predefined	Structure	<>ValueTypeConstraint>>			

VB.NET					UModel														
			constraint	Class	<<ReferenceTypeConstraint>>														
			New		<<ConstructorConstraint>>														
			attribute sections		<<Attributes>>														
Constructor	name			name			Operation <<Constructor>>	Parameter											
	modifiers	Friend		visibility	package														
		Protected Friend			protected <<Friend>>														
		Public			public														
		Protected			protected														
		Private			private														
	Shared		static																
	attribute sections			<<Attributes>>															
	doc comments			Comment(->Documentation)															
	Parameter	name		name															
		modifiers	ByRef	direction	inout		Parameter												
			ByVal		in														
			ParamArray	varArgList															
			Optional	default															
		type		type															
		type dimensions		multiplicity															
		nullable		<<Nullable>>															
Property	name			name			Operation <<Property>>	Parameter											
	modifiers	Friend		visibility	package														
		Protected Friend			protected <<Friend>>														
		Public			public														
		Protected			protected														
		Private			private														
	Default		<<Property>> ( Default <= IsDefault )																
	Shared		static																

VB.NET				UModel									
		MustOverride		abstract									
		NotOverridable		leaf									
		Overrides		<>Overrides>>									
		Overridable		<>Overridable>>									
		Shadow s		<>Shadow s>>									
		Overloads		<>Overloads>>									
		ReadOnly		<>GetAccessor>> ( without <>SetAccessor>> )									
		WriteOnly		<>SetAccessor>> ( without <>GetAccessor>> )									
	attribute sections			<>Attributes>>									
	doc comments			Comment(->Documentation)									
	type			direction	return	Parameter							
	type dimensions			multiplicity									
	nullable			<>Nullable>>									
	Get Accessor	modifiers	Friend Protected Friend Protected Private	visibility	Friend Protected Friend Protected Private	<>GetAccessor>>							
	Set Accessor	modifiers	Friend Protected Friend Protected Private	visibility	Friend Protected Friend Protected Private	<>SetAccessor>>							
Operator	name			name			Operation <>Operator>>						
	modifiers	Public		visibility	Public								
		Shared		static									
		Narrow ing		name <= Narrow ing									
		Widening		name <= Widening									
	attribute sections			<>Attributes>>									
	doc comments			Comment(->Documentation)									

VB.NET				UModel					
		type		direction	return	Parameter			
Parameter		name		name					
		modifier	ByVal	direction	in				
		type		type					
		type dimensions		multiplicity					
		nullable		<<Nullable>>					
Event		name		name		Operation <<Event>>			
		modifiers	Friend	visibility	package				
			Protected Friend		protected <<Friend>>				
			Public		public				
			Protected		protected				
			Private		private				
		Shared		static					
		MustOverride		abstract					
		NotOverridable		leaf					
		Overrides		<<Overrides>>					
		Overridable		<<Overridable>>					
		Shadows		<<Shadows>>					
		Overloads		<<Overloads>>					
Type Parameter		kind	without specifying a delegate type		<<Event>> ( Type <= Simple )				
			with specifying a delegate type		<<Event>> ( Type <= Regular )				
			with custom accessors		<<Event>> ( Type <= Custom )				
		attribute sections		<<Attributes>>		Parameter			
		doc comments		Comment(->Documentation)					
		type		direction	return				
		type dimensions		multiplicity					
		nullable		<<Nullable>>		Template Parameter			
		name		name					
		constraint		constraining classifier					

VB.NET				UModel					
		predefined constraint	Structure	<<ValueTypeConstraint>>					
			Class	<<ReferenceTypeConstraint>>					
			New	<<ConstructorConstraint>>					
		attribute sections		<<Attributes>>					
Structure	name			name					
	modifiers	Friend		visibility	package				
		Protected Friend			protected <<Friend>>				
		Public			public				
		Protected			protected				
		Private			private				
		Partial			<<Partial>>				
		Shadows			<<Shadows>>				
	filename			code file name					
	associated projectfile/directory			ComponentRealization					
	base types			InterfaceRealization(s)					
	attribute sections			<<Attributes>>					
	doc comments			Comment(->Documentation)					
	Field	name		name		Property			
		modifiers	Friend	visibility	package				
			Public		public				
			Private		private				
			Shared		static				
			ReadOnly		readonly				
			Shadows	<<Shadows>>					
		type		type					
		type dimensions		multiplicity					
		nullable		<<Nullable>>					
		default value		default					
		attribute sections		<<Attributes>>					
		doc comments		Comment(->Documentation)					

VB.NET				UModel							
Constant	Property <<Const>>	name		name							
		modifiers	Friend		visibility	package					
			Public			public					
			Private			private					
		Shadows		<<Shadows>>							
		type		type							
		type dimensions		multiplicity							
		nullable		<<Nullable>>							
		default value		default							
		attribute sections		<<Attributes>>							
Method	Operation	doc comments		Comment(->Documentation)							
		name		name							
		modifiers	Friend		visibility	package					
			Public			public					
			Private			private					
		Shared		static							
		MustOverride		abstract							
		NotOverridable		leaf							
		Overrides		<<Overrides>>							
		Overridable		<<Overridable>>							
		Partial		<<Partial>>							
		Shadows		<<Shadows>>							
		Overloads		<<Overloads>>							
		attribute sections		<<Attributes>>							
		doc comments		Comment(->Documentation)							
		implemented interfaces		implements							
		type (function)		direction	return	Parameter					
		Parameter	name	name							
			modifiers	ByRef	direction	inout					
				ByVal	in						

VB.NET					UModel																
Type Parameter				ParamArray	varArgList	Template Parameter	Operation <>Constructor>>	Parameter	Operation <>Property>>												
				Optional	default																
				type																	
				type dimensions																	
				nullable																	
				name																	
				constraint																	
				predefined constraint	Structure																
					Class																
					New																
Constructor	attribute sections				<>Attributes>>																
	name				name																
	modifiers	Friend		visibility	package		Parameter	Operation <>Property>>	Operation <>Constructor>>												
		Public			public																
		Private			private																
		Shared			static																
	attribute sections				<>Attributes>>																
	doc comments				Comment(>->Documentation)																
	Parameter	name		name		Parameter															
		modifiers	ByRef	direction	inout																
			ByVal		in																
			ParamArray	varArgList																	
			Optional	default																	
		type		type																	
		type dimensions		multiplicity																	
		nullable		<>Nullable>>																	
	Property	name			name																

VB.NET				UModel				
		modifiers	Friend Public Private Shared Default MustOverride NotOverridable Overrides Overridable Shadow s Overloads ReadOnly WriteOnly	visibility	package public private static <>Property>> ( Default <= IsDefault ) abstract leaf <>Overrides>> <>Overridable>> <>Shadow s>> <>Overloads>> <>GetAccessor>> ( without <>SetAccessor>> ) <>SetAccessor>> ( without <>GetAccessor>> )	y>>		
		attribute sections			<>Attributes>>			
		doc comments			Comment(->Documentation)			
		type			direction	return	Parameter	
		type dimensions			multiplicity			
		nullable			<>Nullable>>			
	Get Accessor	modifiers	Friend Private	visibility	Friend Private	<>GetAccessor>>	Operation <>Operator>>	
	Set Accessor	modifiers	Friend Private	visibility	Friend Private	<>SetAccessor>>		
	Operator	name			name			
		modifiers	Public Shared Narrow ing Widening	visibility	Public static name <= Narrow ing name <= Widening		Operation <>Operator>>	
		attribute sections			<>Attributes>>			
		doc comments			Comment(->Documentation)			

VB.NET				UModel							
		type		direction	return	Parameter					
	Parameter	name		name		Parameter					
		modifier	ByVal	direction	in						
		type		type							
		type dimensions		multiplicity							
		nullable		<<Nullable>>							
	Event	name		name		Operation <<Event>>					
		modifiers	Friend		visibility	package					
			Public			public					
			Private			private					
		Shared		static							
		MustOverride		abstract							
		NotOverridable		leaf							
		Overrides		<<Overrides>>							
		Overridable		<<Overridable>>							
		Shadows		<<Shadows>>							
		Overloads		<<Overloads>>							
	kind	without specifying a delegate type		<<Event>> ( Type <= Simple )		Template Parameter					
		with specifying a delegate type		<<Event>> ( Type <= Regular )							
		with custom accessors		<<Event>> ( Type <= Custom )							
		attribute sections		<<Attributes>>							
		doc comments		Comment(->Documentation)							
		type		direction	return	Parameter					
		type dimensions		multiplicity							
	Type Parameter	nullable		<<Nullable>>		Template Parameter					
		name		name							
		constraint		constraining classifier							
		predefine d constraint	Structure	<<ValueTypeConstraint>>							
			Class	<<ReferenceTypeConstraint>>							

VB.NET				UModel								
			New	<<ConstructorConstraint>>								
			attribute sections	<<Attributes>>								
Interface	name			name			Interface					
	modifiers	Friend		visibility	package							
		Protected Friend			protected <<Friend>>							
		Public			public							
		Protected			protected							
		Private			private							
	Shadows			<<Shadows>>								
	filename			code file name								
	associated projectfile/directory			ComponentRealization								
	base types			Generalization(s)								
Method	attribute sections			<<Attributes>>								
	doc comments			Comment(->Documentation)								
	modifiers	name		name		Operation	Parameter					
		Public		visibility	public							
	Shadows		<<Shadows>>									
	attribute sections			<<Attributes>>								
	doc comments			Comment(->Documentation)								
	type (function)			direction	return		Parameter					
	Parameter	name		name								
		modifiers	ByRef	direction	inout							
			ByVal		in							
			ParamArray	varArgList								
			Optional	default								
		type		type								
		type dimensions		multiplicity								
		nullable		<<Nullable>>								
	Type Parameters	name		name		Template Parameters						

VB.NET				UModel							
Property	constraint	constraint		constraining classifier							
		predefined constraint	Structure	<<ValueTypeConstraint>>							
			Class	<<ReferenceTypeConstraint>>							
		New		<<ConstructorConstraint>>							
attribute sections				<<Attributes>>							
Event	name			name							
	modifiers	Public		visibility	public						
		Default		<<Property>> ( Default <= IsDefault )							
		Shadows		<<Shadows>>							
		ReadOnly		<<GetAccessor>> ( without <<SetAccessor>> )							
		WriteOnly		<<SetAccessor>> ( without <<GetAccessor>> )							
	attribute sections			<<Attributes>>							
	doc comments			Comment(->Documentation)							
	type			direction	return						
	type dimensions			multiplicity							
	nullable			<<Nullable>>							
	name			name							
	modifiers	Public		visibility	public						
		Shadows		<<Shadows>>							
	kind	without specifying a delegate type		<<Event>> ( Type <= Simple )							
		with specifying a delegate type		<<Event>> ( Type <= Regular )							
	attribute sections			<<Attributes>>							
	doc comments			Comment(->Documentation)							
	type			direction	return						
	type dimensions			multiplicity							
	nullable			<<Nullable>>							

VB.NET			UModel								
Type Parameter	name constraint predefined constraint attribute sections	name	name	Template Parameter							
		constraint	constraining classifier								
		Structure Class New	<<ValueTypeConstraint>>								
			<<ReferenceTypeConstraint>>								
			<<ConstructorConstraint>>								
		attribute sections									
Delegate	name			name							
	modifiers	Friend		visibility	package	Class <<Delegate>>					
		Protected Friend			protected <<Friend>>						
		Public			public						
		Protected			protected						
		Private			private						
	Shadows			<<Shadows>>							
	filename			code file name							
	associated projectfile/directory			ComponentRealization							
	attribute sections			<<Attributes>>							
	doc comments			Comment(>-Documentation)							
	type			direction	return	Parameter	Operation				
	Parameter	name		name							
		modifiers	ByRef	direction	inout						
			ByVal		in						
		type		type							
		type dimensions		multiplicity							
		nullable		<<Nullable>>							
Type Parameter	name			name		Template Parameter					
	constraint			constraining classifier							
	predefined constraint	struct		<<ValueTypeConstraint>>							
		class		<<ReferenceTypeConstraint>>							
		new()		<<ConstructorConstraint>>							
	attribute sections			<<Attributes>>							

VB.NET		UModel			
Enum	name	name			
	modifiers	Friend	visibility	package	
		Protected Friend		protected <>Friend>>	
		Public		public	
		Protected		protected	
		Private		private	
	Shadow s		<>Shadow s>>		
	filename		code file name		
	associated projectfile/directory		ComponentRealization		
	base type		type	<>BaseType>>	
	attribute sections		<>Attributes>>		
	doc comments		Comment(->Documentation)		
Enum Constant	name	name		Enumeration Literal	
	default value	default			
	attribute sections doc comments		<>Attributes>>		
			Comment(->Documentation)		
Parameterized Type		Anonymous Bound Element			

### 6.6.3 Java Mappings

The table below shows the one-to-one correspondence between:

- UModel elements and Java code elements, when outputting model to code
- Java code elements and UModel model elements, when inputting code into model

Java		UModel	
Project	projectfile	projectfile	Component
	directory	directory	
Package	name	name	Package <>nameSpace>>
Class	name	name	Class

Java			UModel			
modifiers	package		visibility	package		
	public			public		
	protected			protected		
	private			private		
	abstract		abstract			
	strictfp		<<strictfp>>			
	final		<<final>>			
	filename		code file name			
	associated projectfile/directory		ComponentRealization			
	extends clause		Generalization			
Field	implements clause		InterfaceRealization(s)			
	java docs		Comment(->Documentation)			
	name		name		Property	
		modifiers	package	package		
			public	public		
			protected	protected		
			private	private		
		static	static			
		transient	<<transient>>			
		volatile	<<volatile>>			
		final	<<final>>			
Method	type		type		Operation	
	type dimensions		multiplicity			
	default value		default			
	java docs		Comment(->Documentation)			
	name		name			
Modifiers		modifiers	package	package	Operation	
			public	public		
			protected	protected		
			private	private		

Java				UModel						
			static	static						
			abstract	abstract						
			final	<<final>>						
			native	<<native>>						
			strictfp	<<strictfp>>						
			synchronized	<<synchronized>>						
		throws clause		raised exceptions						
		java docs			Comment(->Documentation)					
		type			direction	return	Parameter			
	Parameter	name	name		name					
			modifier	final	<<final>>					
			...		varArgList					
			type		type					
			type dimensions		multiplicity					
	Type Parameter	name	name		name		Template Parameter			
			bound		constraining classifier					
	Constructor	name			name		Operation <<constructor>>			
		modifiers	public		visibility	public				
			protected			protected				
			private			private				
		throws clause			raised exceptions					
		java docs			Comment(->Documentation)					
		Parameter	name		name		Parameter			
			modifier	final	<<final>>					
			...		varArgList					
			type		type					
			type dimensions		multiplicity					
	Type Parameter	name	name		name		Template Parameter			
			bound		constraining classifier					

Java				UModel								
	Type Parameter	name		name		Template Parameter						
		bound		constraining classifier								
Interface	name			name			Interface					
	modifiers	package		visibility	package							
		public			public							
		protected			protected							
		private			private							
		abstract		abstract								
	strictfp			<<strictfp>>								
	filename			code file name								
	associated projectfile/directory			ComponentRealization								
	extends clause			Generalization(s)								
	java docs			Comment(->Documentation)								
Field	name			name		Property						
	modifiers	public		visibility	public							
		static		static								
		final		<<final>>								
	type			type								
	type dimensions			multiplicity								
	default value			default								
	java docs			Comment(->Documentation)								
	Method			name		Operation						
Method	modifiers	public		visibility	public							
		abstract		abstract								
		throws clause			raised exceptions							
	java docs			Comment(->Documentation)								
	type			direction	return	Parameter						
	Parameter	name		name								
		modifier	final	<<final>>								
	...			varArgList								

Java				UModel									
Type Parameter			type	type									
			type dimensions	multiplicity									
	Type Parameter	name	name	Template Parameter									
		bound	constraining classifier										
	Type Parameter	name		name									
		bound		constraining classifier									
Enum	name			name			Enumeration						
	modifiers	package		visibility	package								
		public			public								
		protected			protected								
		private			private								
	filename			code file name									
	associated projectfile/directory			ComponentRealization									
	java docs			Comment(->Documentation)									
	Enum Constant	name		name		Enumeration Literal							
	Field	name		name			Property						
		modifiers	package	visibility	package								
			public		public								
		protected	protected		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							

Java				UModel				
		modifiers	package	visibility	package			
			public		public			
			protected		protected			
			private		private			
		static		static				
		abstract		abstract				
		final		<<final>>				
		native		<<native>>				
		strictfp		<<strictfp>>				
		synchronized		<<synchronized>>				
		throws clause		raised exceptions				
		java docs		Comment(->Documentation)				
		type		direction	return	Parameter		
		Parameter	name		name			
			modifier	final	<<final>>			
			...		varArgList			
			type		type			
		type dimensions		multiplicity				
		Type Parameter	name		name	Template Parameter		
			bound		constraining classifier			
	Construct or	name		name		Operation <<constructor>>		
		modifiers	public		visibility			
			protected					
			private					
		throws clause		raised exceptions				
		java docs		Comment(->Documentation)				
		Parameter	name		name	Parameter		
			modifier	final	<<final>>			
			...		varArgList			
			type		type			

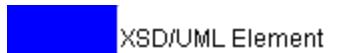
Java				UModel						
		Type Parameter	type dimensions	multiplicity		Template Parameter				
			name	name						
			bound	constraining classifier						
Parameterized Type				Anonymous Bound Element						
Annotation				<>annotations> modifiers						

## 6.6.4 XML Schema Mappings

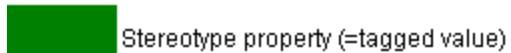
The table below shows the one-to-one correspondence between:

- UModel elements and XML Schema elements, when outputting model to code
- XML Schema elements and UModel model elements, when inputting code into model

Legend:



XSD/UML Element



Stereotype property (=tagged value)

XSD		UModel	
file path		projectfile	Component
schema	target namespace	name	Package <>namespac<>
	attributeFormDefault	attributeFormDefault	Class <>schema<>
	blockDefault	blockDefault	
	elementFormDefault	elementFormDefault	
	finalDefault	finalDefault	
	version	version	
	xml:lang	xml:lang	
	xmlns	xmlns	
	annotation	source	Comment <>appinfo<>
	appinfo		

XSD				UModel						
						>>				
		document ation	xml:lang		xml:lang		Comment <<docume ntation>>			
attributeGr oup	attribute	name		name						
		annotation	appinfo			Comment <<appinfo >>				
		document ation				Comment <<docume ntation>>				
		attribute	name		name		Property <<attribute >>			
			form		form					
			use		use					
			ref		type					
			type		default	fixed				
			default							
		attributeGr oup	ref		type		Property <<attribute Group>>			
		anyAttribu te	namespace		namespace		Property <<anyAttri bute>>			
			processContents		processContents					
attribute	annotation	name		name			Class <<attribute >>			
		form		form						
		use		use						
		type		type		Property				
		default		default	fixed					
		fixed								
		appinfo				Comment <<appinfo >>				
		documentation				Comment <<docume ntation>>				

XSD			UModel		
		simpleType	name (= name of Class + "_anonymousType[n]")	DataType <<simpleT ype>>	
element		name	name	Class <<element >>	
		abstract	abstract		
		block	block		
		final	final		
		form	form		
		nillable	nillable		
		type	type		Property
		default	default		
		fixed	fixed		
		substitutionGroup	general	Generaliz ation <<substi tution>>	
	annotation	appinfo		Comment <<appinfo >>	
		document ation		Comment <<docume ntation>>	
	simpleTyp e		name (= name of Class + "_anonymousType[n]")	DataType <<simpleT ype>>	
	complexT ype		name (= name of Class + "_anonymousType[n]")	Class <<comple xType>>	
	group	name	name	Class <<group> >	
	annotation	appinfo			
		document ation			
	all		name (= "_all")		
			name (= "mg" + "all")	Class <<...>>	

XSD				UModel						
			annotation	appinfo		Comment <<appinfo>>				
			document ation			Comment <<docume ntation>>				
			element	name	name	Property <<element>>				
			ref	type						
			type							
			choice			name (= "_choice")	Property			
						name (= "mg" + "choice")	Class <<choice> >			
				annotation	appinfo					
				document ation		Comment <<docume ntation>>				
				name	name	Property <<element>>				
				ref	type					
				type						
			group			Property <<group> >				
			any	namespac e	namespac e	Property <<any>>				
				processC ontents	processC ontents					
			choice			Property				
						Class <<choice> >				
			sequence			Property				
						Class <<sequen ce>>				
			sequence			name (= "_sequence")	Property			

XSD				UModel				
				name (= "mg" + "sequence")	Class <<sequence>>			
				annotation	appinfo		Comment <<appinfo>>	
				document	ation		Comment <<documentation>>	
				element	name	name	Property <<element>>	
					ref	type		
					type			
				group			Property <<group>>	
				any	namespac	namespac	Property <<any>>	
					processC	ontents		
				choice			Property  Class <<choice>>	
				sequence				
			name	name			DataType <<notation>>	
			system	system				
			public	public				
			annotation	appinfo				
				document	ation			
			name	name			Class <<complexType>>	
			abstract	abstract				
			block	block				

XSD				UModel			
final				final			
mixed				mixed			
annotation	source		source				
	appinfo				Comment <<appinfo>>		
group	document ation	xml:lang	xml:lang		Comment <<docume ntation>>		
			name (= "_ref[n]")		Property <<group> >		
	maxOccurs		multiplicity				
	minOccurs						
all	ref		type				
			name (= "mg" + "all")		Class <<all>>		
			name (= "_all")		Property		
	maxOccurs		multiplicity				
choice	minOccurs						
			name (= "mg" + "choice[n]")		Class <<choice> >		
			name (= "_choice[n]")		Property		
	maxOccurs		multiplicity				
sequence	minOccurs						
			name (= "mg" + "sequence[n]")		Class <<sequen ce>>		
			name (= "_sequence[n]")		Property		
	maxOccurs		multiplicity				
attribute	minOccurs						
	name		name		Property <<attribute> >		
ref			type				

XSD				UModel						
simpleType	complexType		type							
		attributeGroup	ref	type		Property <<attribute Group>>				
		anyAttribute	namespace	namespace		Property <<anyAttribute>>				
			processContents	processContents						
		complexContent	restriction	base	general	Generalization <<restriction>>				
			extension			Generalization <<extension>>				
	list	name		name		DataType <<simpleType>> Enumeration <<simpleType>>				
		final		final						
		annotation	source	source						
			appinfo			Comment <<appinfo>>				
		documentation	xml:lang	xml:lang		Comment <<documentation>>				
	union	itemType		name (= "_itemType")	Property <<itemType>>	<<list>>				
		simpleType		DataType <<simpleType>>						
	minExclusive	memberTypes		name (= "memberType[n]")	Property <<memberType>>	<<union>>				
		simpleType		DataType <<simpleType>>						
	minInclusive	value		value		<<minExclusive>>				
		fixed		fixed						
	maxExclusive	value		value		<<minInclusive>>				
		fixed		fixed						
	maxInclusive	value		value		<<maxExclusive>>				

XSD				UModel				
XSD element mapping	UModel element mapping		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	UModel element mapping	name			name		<<complexType>><<simpleContent>>	
			annotation		source			
		appinfo						
			documentation	xml:lang	xml:lang	Comment<<appinfo>>		
						Comment<<documentation>>		

XSD				UModel					
		minExclusive	value	value	<<minExclusive>>				
			fixed	fixed					
		minInclusive	value	value	<<minInclusive>>				
			fixed	fixed					
		maxExclusive	value	value	<<maxExclusive>>				
			fixed	fixed					
		maxInclusive	value	value	<<maxInclusive>>				
			fixed	fixed					
		totalDigits	value	value	<<totalDigits>>				
			fixed	fixed					
		fractionDigits	value	value	<<fractionDigits>>				
			fixed	fixed					
		length	value	value	<<length>>				
			fixed	fixed					
		minLength	value	value	<<minLength>>				
			fixed	fixed					
		maxLength	value	value	<<maxLength>>				
			fixed	fixed					
		whiteSpace	value	value	<<whiteSpace>>				
			fixed	fixed					
		pattern	value	value	<<whiteSpace>>				
		attribute	name	name	Property <<attribute>>				
			ref	type					
			type						
		attributeGroup	ref	type	Property <<attribute Group>>				
		anyAttribute	namespace	namespace					

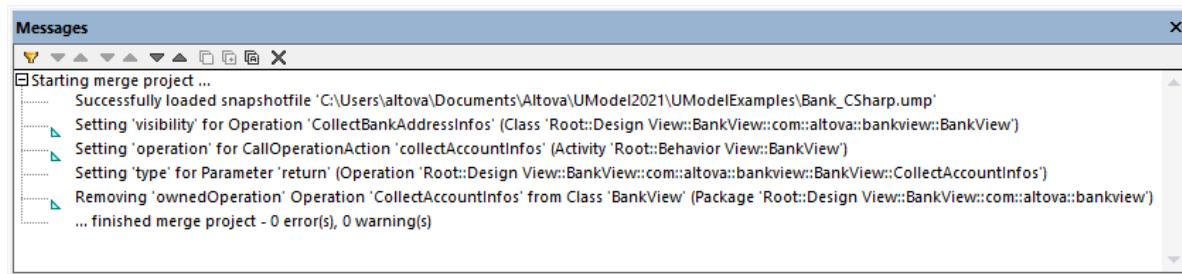
XSD				UModel					
			processC ontents		processC ontents				
		simpleTyp e					<b>DataType</b> <b>&lt;&lt;simpleT ype&gt;&gt;</b>		
		restriction	base		general		<b>Generaliz ation</b> <b>&lt;&lt;restricti on&gt;&gt;</b>		
		extension	base		general		<b>Generaliz ation</b> <b>&lt;&lt;extensi on&gt;&gt;</b>		
	<b>import</b>	schemaLocation		schemaLocation			<b>ElementIm port</b> <b>&lt;&lt;import&gt; &gt;</b>		
		namespace		namespace					
	<b>include</b>	schemaLocation		schemaLocation			<b>ElementIm port</b> <b>&lt;&lt;include &gt;&gt;</b>		
	<b>redefine</b>	schemaLocation		schemaLocation			<b>ElementIm port</b> <b>&lt;&lt;redefin e&gt;&gt;</b>		
		simpleTyp e		<<redefine>>			<b>DataType</b> <b>&lt;&lt;simpleT ype&gt;&gt;</b>		
		complexT ype					<b>Class</b> <b>&lt;&lt;comple xType&gt;&gt;</b>		
		attributeGr oup					<b>Class</b> <b>&lt;&lt;attribute Group&gt;&gt;</b>		
		group					<b>Class</b> <b>&lt;&lt;group&gt; &gt;</b>		

## 6.7 Merging UModel Projects

It is possible to perform a two-way or three-way project merge in UModel. Both operations merge different UModel project files into a common UModel \*.ump model. This option is useful if multiple persons are working on the same project at the same time, or you just want to consolidate your work into one model.

### To merge two UML projects:

1. Open the UML file that is to be the target of the merge process, i.e. the file into which the second model will be merged - the merged project file.
2. Select the menu option **Project | Merge Project...**
3. Select the second UML project that is to be merged into the first one. The Messages window reports on the merge process, and logs the relevant details.



**Note:** Clicking on one of the entries in the Messages window displays that modeling element in the Model Tree.

### Merging results:

- New modeling elements i.e. those that do not exist in the source, are added to the merged model.
- Differences in the same modeling elements; the elements from the second model take precedence, e.g. there can only be one default value of an attribute, the default value of the second file is used.
- Diagram differences: UModel first checks to see if there are differences between diagrams of the two models. If there are, then the new/different diagram is added to the merged model (with a running number suffix, activity1 etc.) and the original diagram is retained. If there are no differences, then identical diagrams(s) are ignored, and nothing is changed. You can then decide which of the diagrams you want to keep or delete, you can of course keep both of them if you want.
- The whole merge process can be undone step-by-step by clicking the **Undo** toolbar button, or pressing **Ctrl+Z**.
- Clicking an entry in the message window displays that element in the Model Tree.
- The file name of the merged file (the first file you opened) is retained.

### 6.7.1 3-Way Project Merge

UModel supports the merging of multiple UModel projects that have been simultaneously edited by different developers, in a 3-way project merge. The 3-way project merge works with top-level UModel projects, i.e. main projects that may contain subprojects, it does not support individual file merging, when these files have unresolved references to other files.

When merging main projects, any editable subprojects are automatically merged as well. There is no need for a separate subproject merging process. For an example, see [Example: Manual 3-Way Project Merge](#)<sup>271</sup>. Note the following:

- The whole merge process can be undone step-by-step by clicking the **Undo** toolbar button, or pressing **Ctrl+Z**.
- Clicking an entry in the message window displays that element in the Model Tree.
- The file name of the merged file, the first file you opened, is retained.

## Merging results

In the following text, "source" means the initial/first project file you open before starting the merge process.

- New modeling elements in the second file i.e. that do not exist in the source, are added to the merged model.
- New modeling elements in the source file i.e. that do not exist in the second file, remain in the merged model.
- Deleted modeling elements from the second file i.e. those that still exist in the source, are removed from the merged model.
- Deleted modeling elements from the source file i.e. that still exist in the second file, remain deleted from the merged model.

Differences to the same modeling elements:

- If a property (e.g. the visibility of a class) is changed in either the source, or second file, the updated value is used in the merged model.
- If a property (e.g. the visibility of a class) is changed in both source and second file, the value of the second file is used (and a warning is shown in the messages window).

Moved elements:

- If an element is moved in the source, or second file, then the element is moved in the merged model.
- If an element is moved (to different parents) in both the source and second file, a prompt appears, and you have to manually select the parent element in the merged model.

Diagram differences:

UModel first checks to see if there are differences between diagrams of the two models. If yes, then the new/different diagram is added to the merged model (with a running number suffix, activity1 etc.) and the original diagram is retained. If there are no differences, then identical diagrams(s) are ignored, and nothing is changed. You can then decide which of the diagrams you want to keep or delete, you can of course keep both of them if you want.

## Source control systems support for 3-way merging

When checking in/out project files, UModel automatically generates "Common ancestor" (or snapshot) files which are then used for the 3-way merge process. This enables a much finer merge result than the normal 2-way merge.

The specific source control system you use, determines if the automatic snapshot 3-way merge process is supported by UModel. A manual 3-way merge is however, always possible.

- Source control systems that perform automatic file merging without user intervention, will probably not support an automatic 3-way merge.
- Source control systems that prompt you to choose between Replace or Merge, when a project file has been changed, will generally support a 3-way merge. After the source control plug-in has replaced the file, selecting the Replace command activates the UModel file alert which then allows you to do a 3-way merge. UModel must be used for the check in/out process.
- Main projects as well as subprojects can be placed under source control. Changing data in a subproject automatically prompts you if the subproject(s) should be checked out.
- Each check in/out action, creates a Common ancestor, or a snapshot, file which is then used during the 3-way project merge process.

**Note:** Snapshot files are automatically created and used only with the standalone versions of UModel, i.e. these functions are not available in the Eclipse or Visual Studio plug-in versions.

### Example

User A edits a UModel project file and changes the name of a class in the BankView Main diagram. User B opens the same project file and changes the visibility of the same class.

As snapshot files are created for each user, the snapshot editing history allows the individual changes to be merged into the project. Both the name and visibility changes are merged into the project file during the 3-way merge process.

## 6.7.2 Example: Manual 3-Way Project Merge

This example illustrates a simple 3-way project merge. Let's suppose that two users, Tom and Alice, created their own copies of a UModel project and made changes to them. There are now three versions of the same project: the original one, Tom's copy, and Alice's copy. In the context of 3-way merging, the original project represents the "common ancestor file".

For the scope of this example, let's assume that the common ancestor file is **Bank\_CSharp.ump** project, available in the folder **C:\Users\<username>\Documents\Altova\UModel2022\UModelExamples**. The copies of Tom and Alice must be created manually. Therefore, let's first create two copies of the **Bank\_Csharp.ump** project in child folders below the ...\\UModelExamples folder. Let's call the child folders **Alice** and **Tom**; the project name can remain as is.

Use the **File | Save Project As** command to create the copies of Tom and Alice. When prompted to adjust the relative paths, click **Yes**. This way you will avoid introducing syntax errors in the project copies.

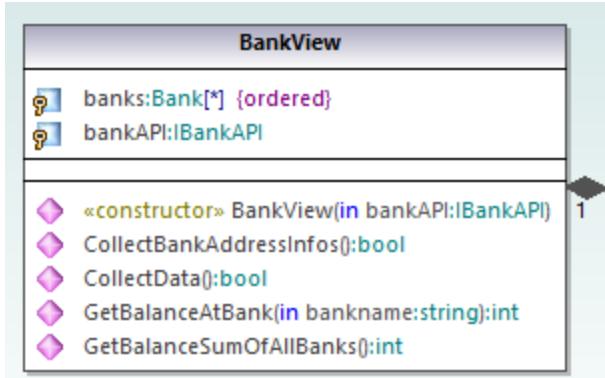
The goal of the example is to show how Alice should merge changes not only from the original **Bank\_CSharp.ump**, but also from Tom's project into a new merged model (a so-called "3-way merge").

### Step 1: Prepare Tom's project

Tom opens the **Bank\_CSharp.ump** project file in folder **Tom**, opens the "BankView Main" diagram, and makes changes to the **BankView** class.

1. Operation `CollectAccountInfos () :bool` is deleted from the `BankView` class.

2. The visibility of the `CollectBankAddressInfos() :bool` operation is changed from "protected" to "public".

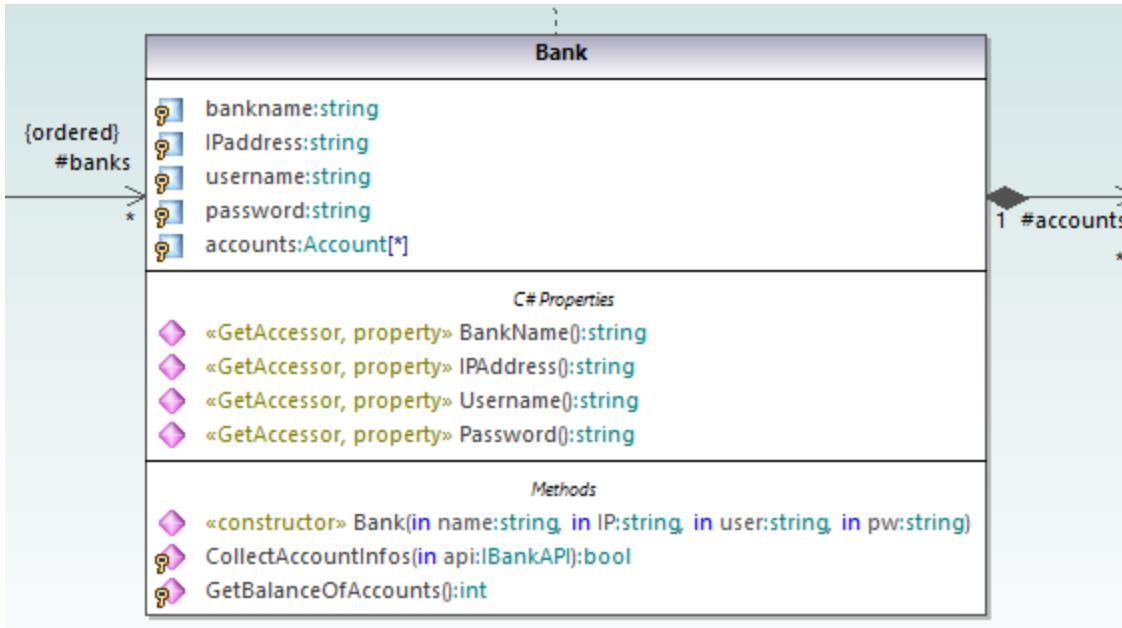


3. The project is then saved.

## Step 2: Prepare Alice's project

Alice opens the **Bank\_CSharp.ump** project file in folder **Alice**, opens the "BankView Main" diagram, and makes changes to the **Bank** class.

1. The operations `CollectAccountInfos` and `GetBalanceOfAccounts` are both changed from "public" to "protected".



2. The project is then saved.

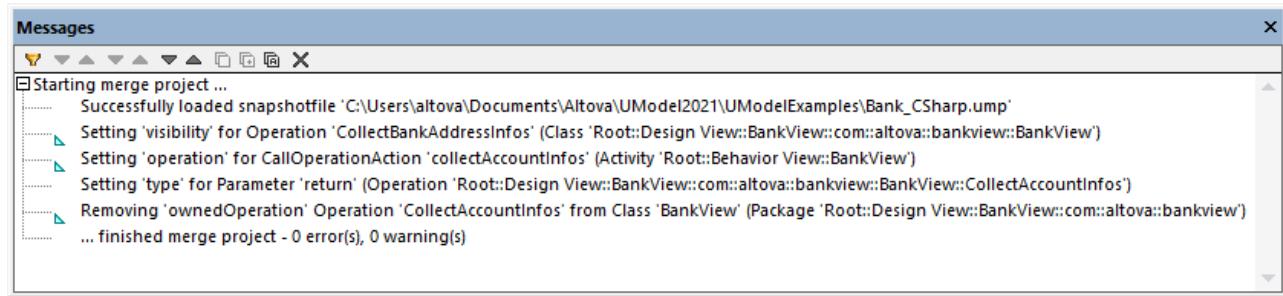
## Step 3: Perform the 3-way merge

Alice now starts a 3-way project merge:

1. Open Alice's project from **Alice** folder.

2. On the **Project** menu, click **Merge Project (3-way)**, and select the project file changed by Tom from **Tom** folder.
3. You are now prompted to open the common ancestor file. Select the original **Bank\_CSharp.ump** project file from the ...**UModelExamples** folder.

The 3-way merge process is started and you return to the project file from which you started the 3-way merge process, i.e. from the project file in the **Alice** folder. The Messages window shows you the merge process in detail.



The outcome of the 3-way merge is as follows:

- The changes made to the project by Tom are replicated in Alice's project.
- The changes made to the project by Alice are retained in the project file.

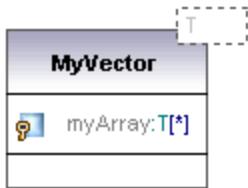
**Note:** The project file in the **Alice** folder should now be used as the common ancestor file for future 3-way merges between the project files in folders **Tom** and **Alice**.

## 6.8 UML Templates

UModel supports the use of UML templates and their mapping to or from Java, C# and Visual Basic generics.

- Templates are "potential" model elements with unbound formal parameters.
- These parameterized model elements, describe a group of model elements of a particular type: classifiers, or operations.
- Templates cannot be used directly as types, the parameters have to be bound.
- Instantiate means binding the template parameters to actual values.
- Actual values for parameters are expressions.
- The binding between a template and model element, produces a new model element (a bound element) based on the template.
- If multiple constraining classifiers exist in C#, then the template parameters can be directly edited in the Properties tab, when the template parameter is selected.

Template **signature** display in UModel:



- Class template called **MyVector**, with formal template parameter "T", visible in the dashed rectangle.
- Formal parameters without type info (T ) are implicitly classifiers: Class, Datatype, Enumeration, PrimitiveType, Interface. All other parameter types must be shown explicitly e.g. Integer.
- Property **myArray** with unbounded number of elements of type T.

Right clicking the template and selecting **Show | Bound elements**, displays the actual bound elements.

Template **binding** display:



- A bound named template **intvector**
- Template of type, **MyVector**, where
- Parameter **T** is substituted/replaced by **int**.
- "**Substituted by**" is shown by - >.

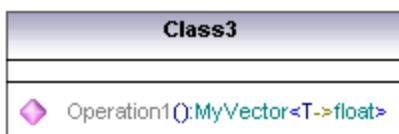
Template **use** in properties/operations:



An anonymous template binding:

- Property **MyFloatVector** of type **MyVector<T>float**

Templates can also be defined when defining properties or operations. The autocomplete function helps you with the correct syntax when doing this.



- Operation1 returns a vector of floats.

## 6.8.1 Template Signatures

A Template signature is a string that specifies the formal template parameters. A template is a parameterized element that is used to generate new model elements by substituting/binding the formal parameters to actual parameters (values).

### Formal template parameter

T

Template with a single untyped formal parameter  
(stores elements of type T)

### Multiple formal template parameters

KeyType:DateTime, ValueType

### Parameter substitution

T>aBaseClass

The parameter substitution must be of type "aBaseClass", or derived from it.

### Default values for template parameters

T=aDefaultValue

### Substituting classifiers

T>{contract}aBaseClass

allowsSubstitutable is true

Parameter must be a classifier that may be substituted for the classifier designated by the classifier name.

### Constraining template parameters

T:Interface>anInterface

When constraining to anything other than a class, (interface, data type), the constraint is displayed after the colon ":" character. E.g. T is constrained to an interface (T:Interface) which must be of type "anInterface" (>anInterface).

### Using wildcards in template signatures

T>vector<T>-?<aBaseClass>

Template parameter T must be of type "vector" which contains objects which are a supertype of aBaseClass.

**Extending template parameters**

T&gt;Comparable&lt;T&gt;T&gt;

## 6.8.2 Template Binding

Template binding involves the substitution of the formal parameters by actual values, i.e. the template is instantiated. UModel automatically generates anonymously bound classes, when this binding occurs. Bindings can be defined in the class name field as shown below.

**Substituting/binding formal parameters**

vector &lt;T&gt;int&gt;

**Create bindings using the class name**

a\_float\_vector:vector&lt;T&gt;float&gt;

**Binding multiple templates simultaneously**

Class5:vector&lt;T&gt;int, map&lt;KeyType&gt;int, ValueType&lt;T&gt;int&gt;

**Using wildcards ? as parameters (Java 5.0)**

vector&lt;T&gt;?&gt;

**Constraining wildcards - upper bounds (UModel extension)**

vector&lt;T&gt;?&gt;aBaseClass&gt;

**Constraining wildcards - lower bounds (UModel extension)**

vector&lt;T&gt;?&lt;aDerivedClass&gt;

## 6.8.3 Template Usage in Operations and Properties

**Operation returning a bound template**

Class1

Operation1():vector&lt;T&gt;int&gt;

Parameter T is bound to "int". Operation1 returns a vector of ints.

**Class containing a template operation**

Class1

Operation1&lt;T&gt;(in T):T

**Using wildcards**

Class1

Property1:vector&lt;T&gt;?&gt;

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

**Typed properties can be displayed as associations as follows:**

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

## 7 Generating UML Documentation

Altova website:  [UML project documentation](#)

Run the **Project | Generate Documentation** menu command to generate detailed documentation about your UML project in HTML, Microsoft Word, RTF or PDF format. The documentation generated by this command can be freely altered and used; permission from Altova to do so is not required.

### Notes

- To generate documentation in PDF format or to customize the generated documentation, Altova StyleVision (<https://www.altova.com/stylevision>) must be installed and licensed.
- To generate documentation in Microsoft Word format, Microsoft Word 2000 or later is required.

Documentation is generated for the modeling elements you select in the Generate Documentation dialog box. You can either use the fixed design, or specify a custom StyleVision Power Stylesheet (SPS). Using a StyleVision Power Stylesheet enables you to customize the output of the generated documentation, see [Customizing Output with StyleVision](#) <sup>287</sup>.

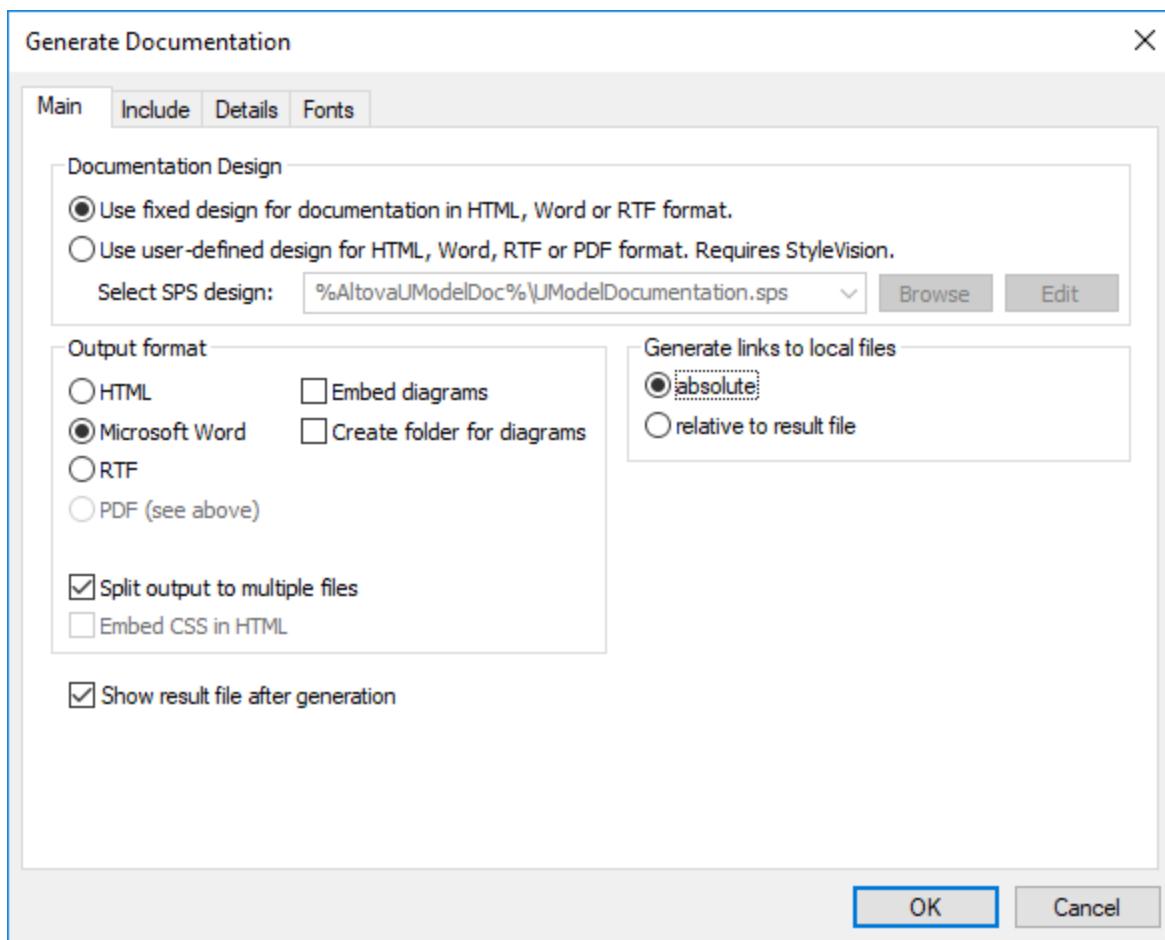
You can also create partial documentation of modeling elements. To do this, right-click an element (or multiple elements using **Ctrl+Click**) in the Model Tree and select **Generate Documentation**. The element can be a folder, class, interface, and so on. The documentation options are the same in both cases.

Related elements are hyperlinked in the generated output, enabling you to navigate from component to component. All manually created hyperlinks also appear in the documentation.

If your project contains UModel profiles (such as C#, Java, VB.NET, and so on), the generated documentation will include these if the **Included subprojects** option is enabled in the **Include** tab, see [Documentation Generation Options](#) <sup>282</sup>.

### To generate documentation:

1. Open a project (for example, C:\Users\<username>\Documents\Altova\UModel2022\UModel\Examples\Bank\_Java.ump).
2. On the **Project** menu, click **Generate Documentation**.



3. Select an output format (HTML, Word, RTF, PDF).
4. Optionally, customize the generation options, see [Documentation Generation Options](#) 282.
5. Click **OK** and choose a target output folder.

The following image shows a fragment of UModel fixed-design documentation generated from the **Bank\_Java.ump** project file.

**Bank\_Java.ump**

project location [C:\Users\UModelExamples\Bank\\_Java.ump](C:\Users\UModelExamples\Bank_Java.ump)

## Index of diagrams:

Activity Diagram	<a href="#">collectData Draft</a>
Class Diagram	<a href="#">BankView Main</a>
	<a href="#">Hierarchy of Account</a>
Component Diagram	<a href="#">BankView realization</a>
	<a href="#">Overview</a>
Composite Structure Diagram	<a href="#">Account Transfer</a>
Deployment Diagram	<a href="#">Deployment</a>
Object Diagram	<a href="#">Sample Accounts</a>
Profile Diagram	<a href="#">Apply Java Profile</a>
Sequence Diagram	<a href="#">Collect Account Information</a>
	<a href="#">Connect to BankAPI</a>
State Machine Diagram	<a href="#">BankAPI Draft</a>
	<a href="#">Query BankServer Draft</a>
UseCase Diagram	<a href="#">Overview Account Balance</a>

## Index of elements:

Actor	<a href="#">Bank</a>	<a href="#">Standard User</a>	
Class	<a href="#">Account</a>	<a href="#">Bank</a>	<a href="#">BankView</a>
	<a href="#">CreditCardAccount</a>	<a href="#">SavingsAccount</a>	
Component	<a href="#">Bank API client</a>	<a href="#">BankView</a>	<a href="#">BankView GUI</a>
Interface	<a href="#">IBankAPI</a>		

As illustrated above, the generated documentation includes an index of diagrams and elements (with links) at the top of the HTML file.

The image below shows a fragment of the generated documentation for the `Account` class. Note that the individual members in class diagrams are also hyperlinked to their definitions. For example, clicking a property or operation takes you to its definition. The hierarchy classes, as well as all underlined text, are also hyperlinked.

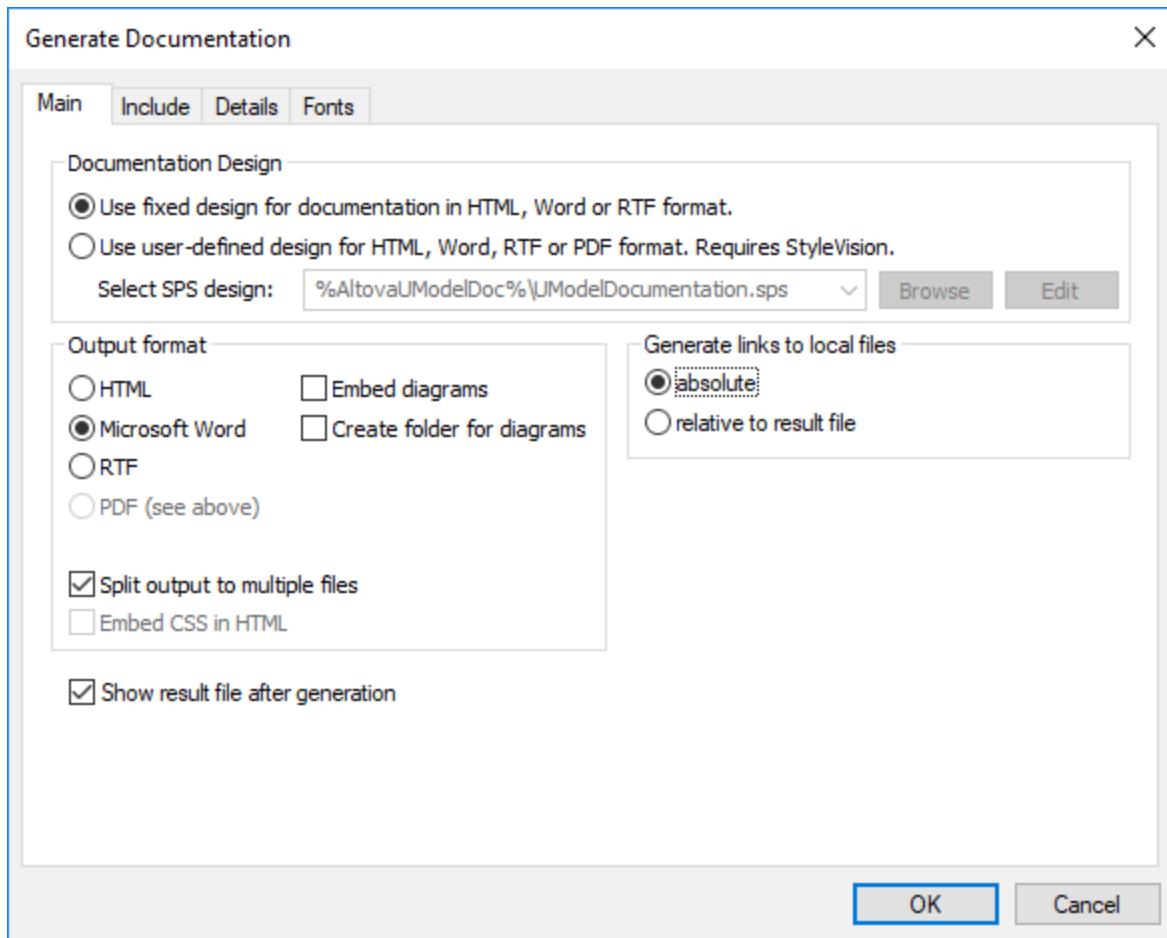
Class Account							
diagram	<p style="text-align: center;"><b>Account</b></p> <table border="1"><tr><td>balance:float=0</td></tr><tr><td>id:String</td></tr><tr><td>«constructor» Account()</td></tr><tr><td>getBalance():float</td></tr><tr><td>getId():String</td></tr><tr><td>collectAccountInfo(in bankAPI:IBankAPI):boolean</td></tr></table>	balance:float=0	id:String	«constructor» Account()	getBalance():float	getId():String	collectAccountInfo(in bankAPI:IBankAPI):boolean
balance:float=0							
id:String							
«constructor» Account()							
getBalance():float							
getId():String							
collectAccountInfo(in bankAPI:IBankAPI):boolean							
hierarchy	<pre>classDiagram     class Account {         &lt;&lt;abstract&gt;&gt;         balance:float=0         id:String         &lt;&lt;constructor&gt;&gt; Account()         getBalance():float         getId():String         collectAccountInfo(in bankAPI:IBankAPI):boolean     }     class CheckingAccount {         &lt;&lt;Account&gt;&gt;     }     class SavingsAccount {         &lt;&lt;Account&gt;&gt;     }     class CreditCardAccount {         &lt;&lt;Account&gt;&gt;     }</pre>						
owner	<a href="#">bankview</a>						
properties	<pre>qualified name Design View::BankView::com::altova::bankview::Account visibility public leaf false abstract true isFinalSpecialization false active false code file name Account.java code file path C:\UML_Bank_Sample\JavaCode\com\altova\bankview\Account.java «annotations» false «static» false «strictfp» false</pre>						

## 7.1 Documentation Generation Options

When generating documentation from UModel projects, you can set various options as described below. The options are organized by the tab in which they appear in the "Generate Documentation" dialog box.

### Main tab

The **Main** tab includes the general documentation generation options.



#### Documentation Design:

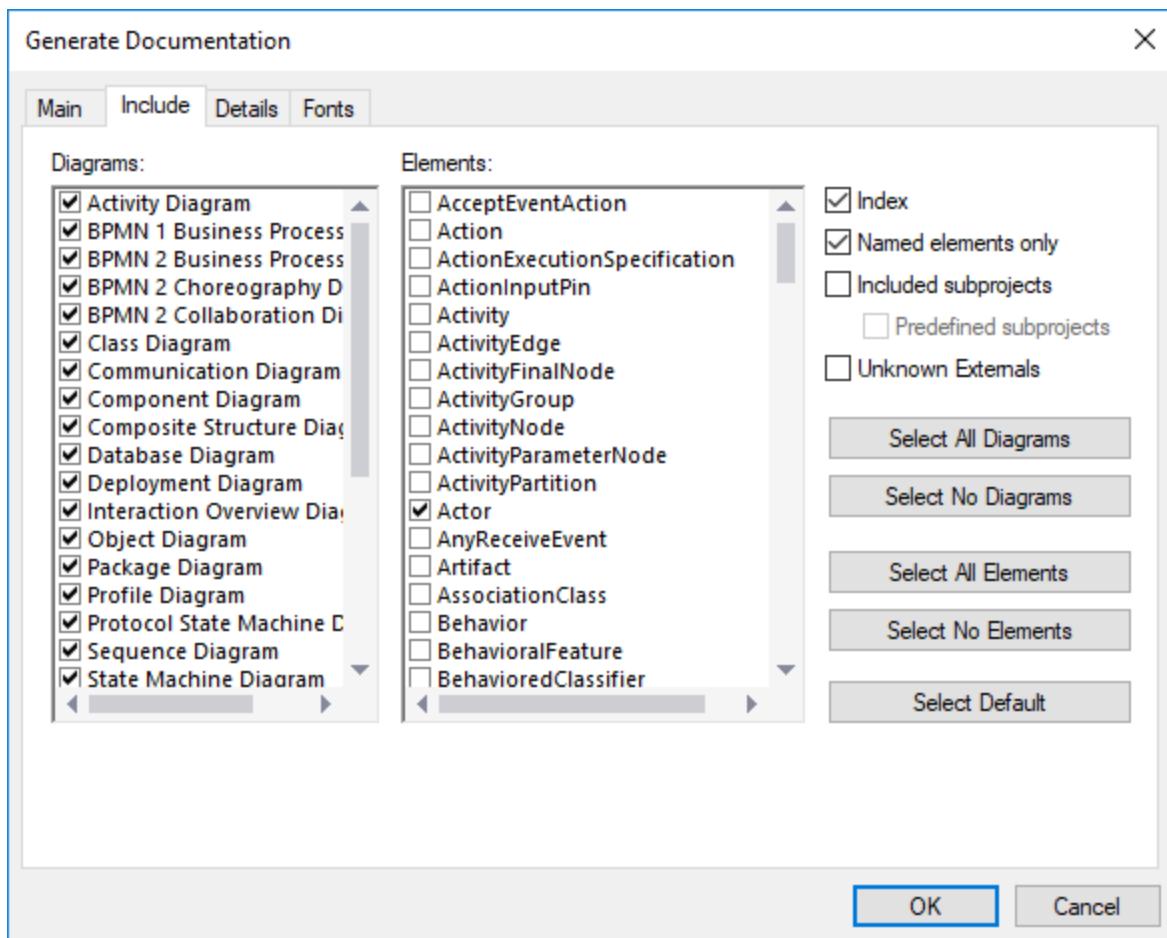
- Select **Use fixed design...** to use the UModel built-in documentation design.
- Select **Use user-defined...** to generate documentation formatted with the help of a custom StyleVision Power Stylesheet (.sps file) created in StyleVision. Note: This option requires Altova StyleVision to be installed, see also [Customizing Output with StyleVision](#)<sup>287</sup>.
- Click **Browse** to browse for a predefined stylesheet file.
- Click **Edit** to launch StyleVision and open the selected stylesheet file in a StyleVision window.

#### Output format:

- The output format can be one of the following: HTML, Microsoft Word, RTF, or PDF. Microsoft Word documents are created with the .doc file extension when generated using a fixed design, and with a .docx file extension when generated using a StyleVision Power Stylesheet. The PDF output format requires Altova StyleVision to be installed.
- **Split output to multiple files** generates an output file for each modeling element (class, interface, diagram, and so on). Clear this check box to generate one global file with all modeling elements.
- Select the **Embed CSS in HTML** check box to embed the generated CSS code in the HTML documentation. Clear this check box to keep the CSS file external.
- The **Embed diagrams** option is enabled for the Microsoft Word and RTF output options. When this check box is selected, diagrams are embedded in the generated file. Diagrams are created as .png files, which are displayed in the result file via object links.
- **Create folder for diagrams** generates a subfolder below the selected output folder, that will contain all diagrams.
- The **Show result file after generation** option is enabled for all output formats. When this check box is selected, the main generated file is displayed in the default browser (for HTML files), in Microsoft Word (for Word files), or in the default application (for .pdf or .rtf files).
- The **Generate links to local files** option allows you to specify if the generated links are to be absolute, or relative, to the output file.

### Include tab

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

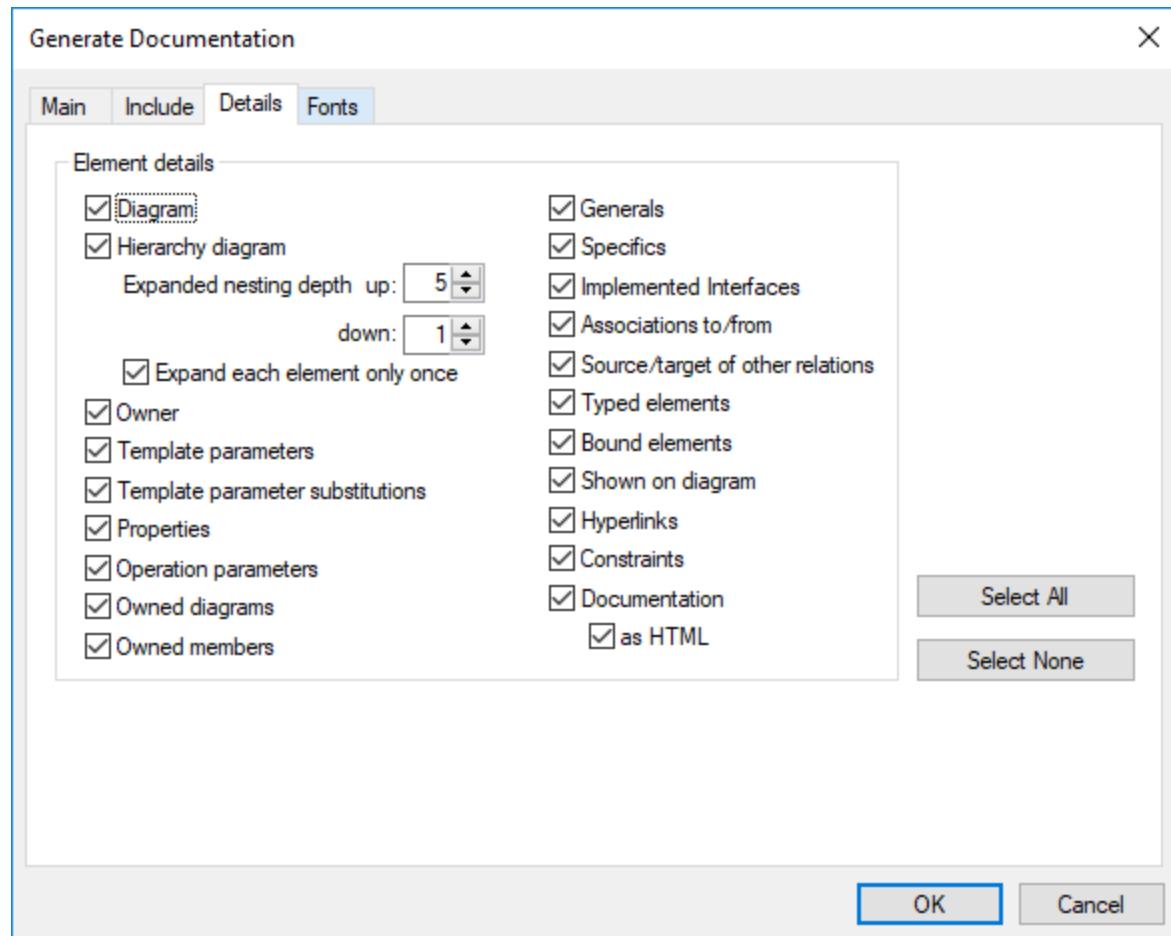


To prevent subprojects or profiles from being documented, clear the **Included subprojects** check box. Be aware that, if this check box is not selected, any elements or diagrams that are in subprojects will not be included in generated documentation. Select the **Predefined subprojects** check box to include UModel built-in profiles such as C# or Java profiles. Note, however, that generating documentation from predefined projects takes a very long time. **Unknown externals** refers to elements whose kind could not be identified—this usually happens after you import source code into UModel without first including the built-in subprojects for that language or language version, see [Including Subprojects](#)<sup>158</sup> for more information.

## Details tab

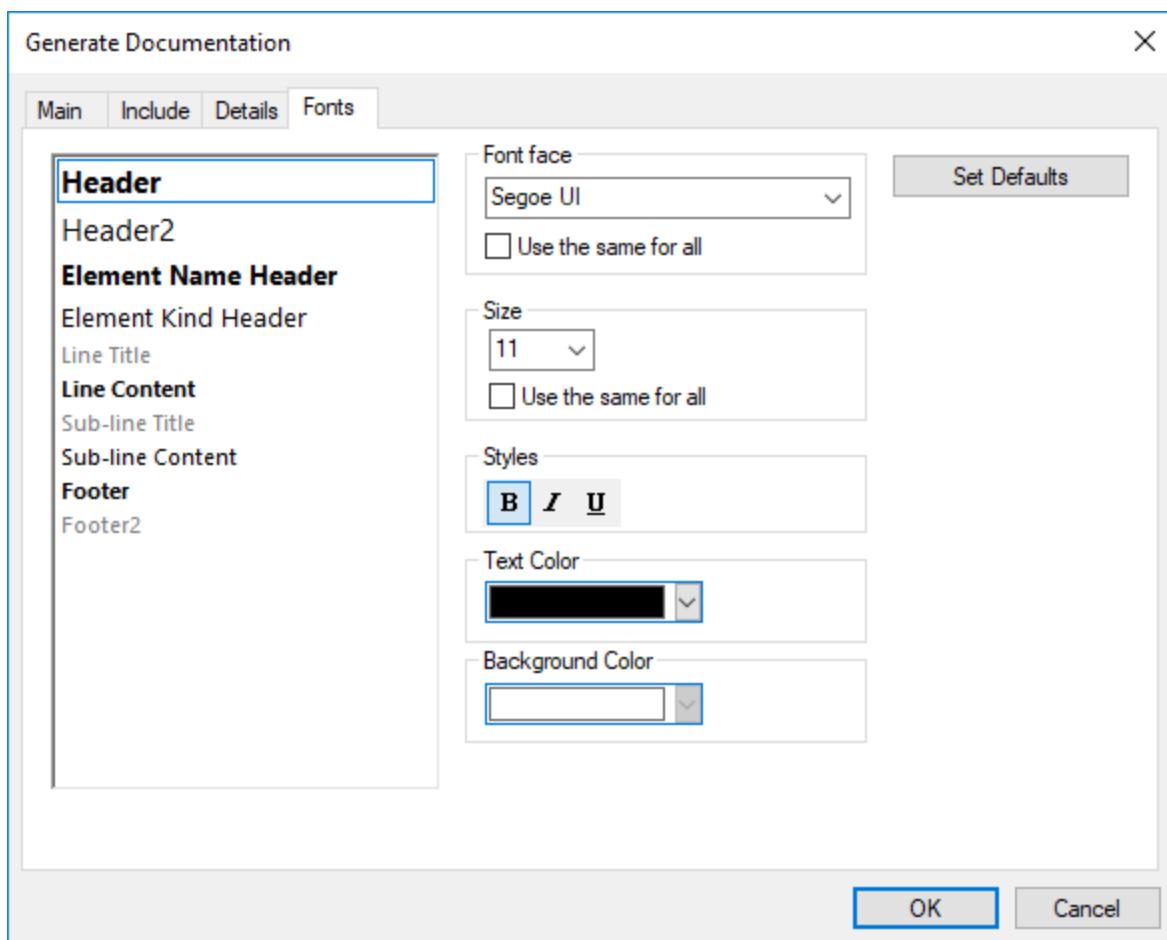
This tab allows you to select the element details that are to appear in the documentation.

- If you intend to import XML tags text in your documentation, clear the **as HTML** option under the **Documentation** option.
- The **up** and **down** fields allow you to define the nesting depth shown above or below the current class in the hierarchy diagram.
- The **expand each element only once** option allows only one of the same classifiers to be expanded in the same image or diagram.



## Fonts tab

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

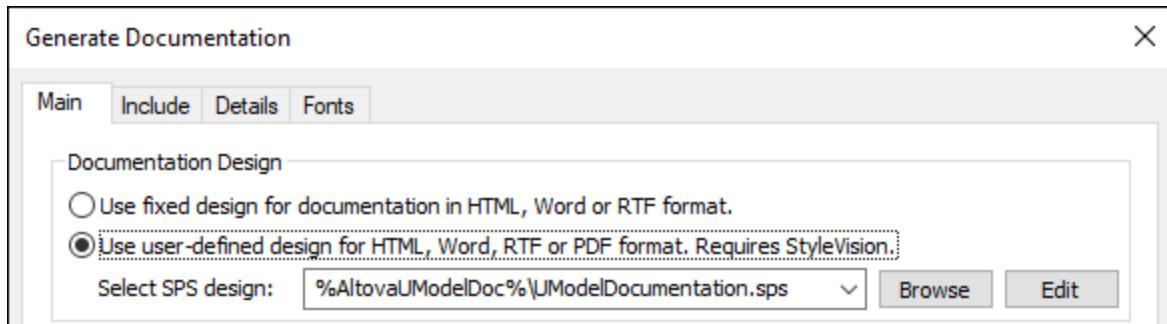


## 7.2 Customizing Output with StyleVision

You can customize the design of UModel-generated documentation with the help of StyleVision Power Stylesheet (.sps) files. Such files are created in Altova StyleVision (<https://www.altova.com/stylevision>). The advantage of using an .sps file is that you have complete control over the design of the documentation. In addition, PDF output is available if an .sps file is used.

To generate documentation with .sps files, Altova StyleVision must be installed and licensed.

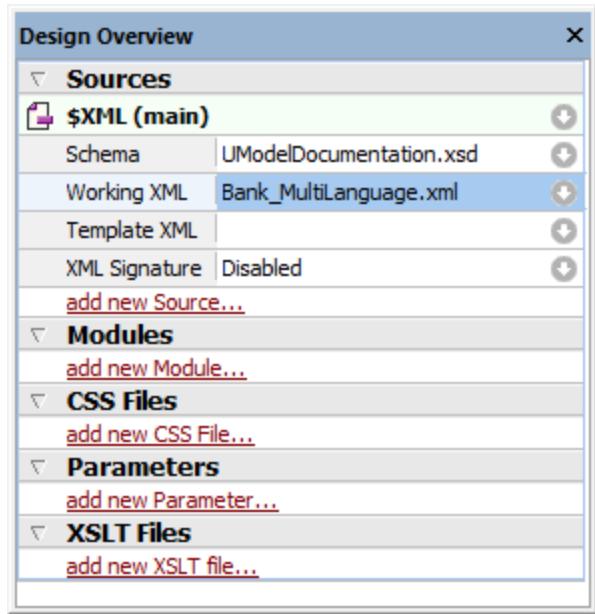
UModel includes a predefined .sps file, which is available at the following path: **C:\users\<username>\Documents\UModel2022\Documentation\UModel\UModelDocumentation.sps**. To format the generated documentation using a custom .sps file, select this option while generating documentation, for example:



You can begin the customization by creating a copy of the default **UModelDocumentation.sps** and editing it in StyleVision. For example, you can change the existing formatting or add links and images to the design.

Any StyleVision Power Stylesheet is based on an XML Schema. In case of stylesheets that control the design of UModel-generated documentation, this schema is available at the following path: **C:\users\<username>\Documents\UModel2022\Documentation\UModel\UModelDocumentation.xsd**. Note that the **UModelDocumentation.xsd** file references the **Documentation.xsd** file located in the folder above it.

When you author custom .sps files in StyleVision for UModel documentation, the **UModelDocumentation.xsd** file must be used as a schema. The image below illustrates the Design Overview window of StyleVison after you open the **UModelDocumentation.sps** file. Notice that it uses the **UModelDocumentation.xsd** schema file, and a working XML required to preview the design. The working XML file is available in the **SampleData** subfolder relative to the schema file.



For instructions about how to edit .sps files, refer to the StyleVision documentation (<https://www.altova.com/documentation>).

## 8 UML Diagrams

Altova website:  [UML\\_diagrams](#)

There are two major groups of UML diagrams, Structural diagrams, which show the static view of the model, and Behavioral diagrams, which show the dynamic view. UModel supports all fourteen diagrams of the UML 2.5 specification, as well as Additional diagrams.

- [Behavioral diagrams](#) 290 include Activity, State machine, Protocol State Machine and Use Case diagrams; as well as the Interaction, Communication, Interaction Overview, Sequence, and Timing diagrams.
- [Structural diagrams](#) 380 include: Class, Composite Structure, Component, Deployment, Object, and Package diagrams.
- [Additional diagrams](#) 417 XML schema diagrams.

**Note:** The **Ctrl+Enter** keys can be used to create multi-line labels for most of the modeling diagrams, e.g. Lifeline labels in sequence diagrams, timing diagrams; guard conditions, state names, activity names etc.

## 8.1 Behavioral Diagrams

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

-  [Activity Diagram](#)
-  [State Machine Diagram](#)
-  [Protocol State Machine Diagram](#)
-  [Use Case Diagram](#) (335)

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

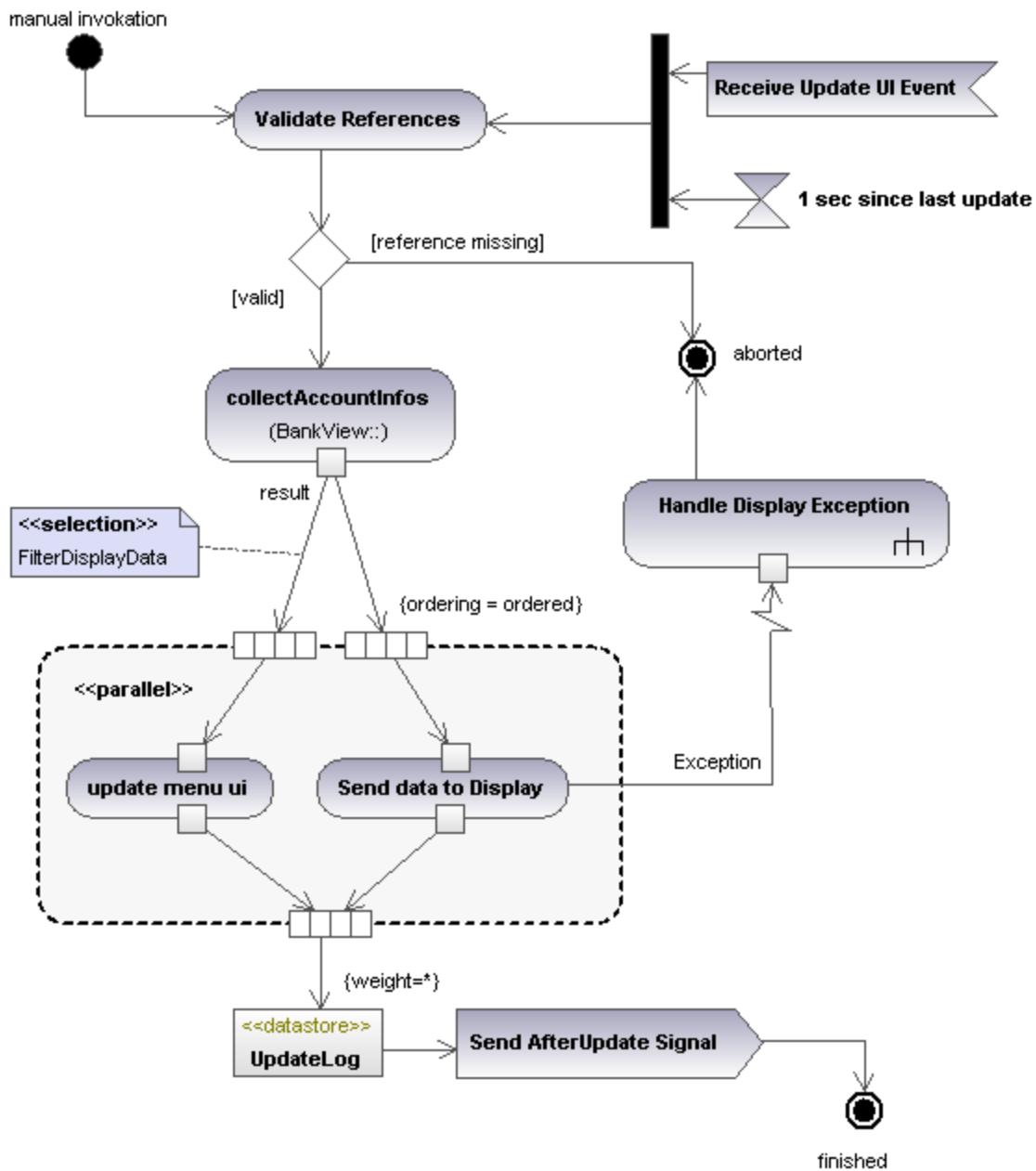
-  [Communication Diagram](#)
-  [Interaction Overview Diagram](#)
-  [Sequence Diagram](#)
-  [Timing Diagram](#) (371)

### 8.1.1 Activity Diagram

Altova website:  [UML Activity diagrams](#)

Activity diagrams are useful for modeling real-world workflows of business processes, and display which actions need to take place and what the behavioral dependencies are. The Activity diagram describes the specific sequencing of activities and supports both conditional and parallel processing. The Activity diagram is a variant of the State diagram, with the states being activities.

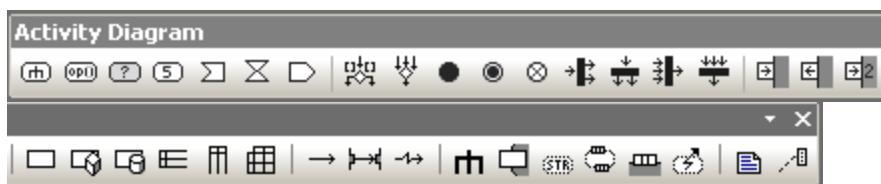
The Activity diagram shown below is available in the **Bank\_MultiLanguage.ump** sample, in the ... **\UModelExamples** folder supplied with UModel.



### 8.1.1.1 Inserting Activity Diagram elements

To add elements to the diagram:

1. Click the element's toolbar button in the Activity Diagram toolbar.



2. Click in the Activity Diagram to insert the element.

To insert multiple elements of the selected type, hold down the **Ctrl** key and click in the diagram window.

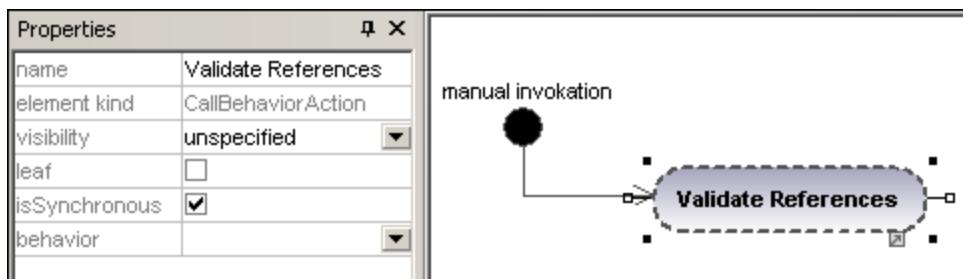
### Dragging existing elements into the activity diagram

Most elements occurring in other activity diagrams can be inserted into an existing activity diagram.

1. Locate the element you want to insert in the [Model Tree Window](#)<sup>79</sup> (you can use the search function text box, or press **Ctrl+F** to search for any element).
2. Drag the element(s) into the activity diagram.

### Inserting an action (CallBehavior)

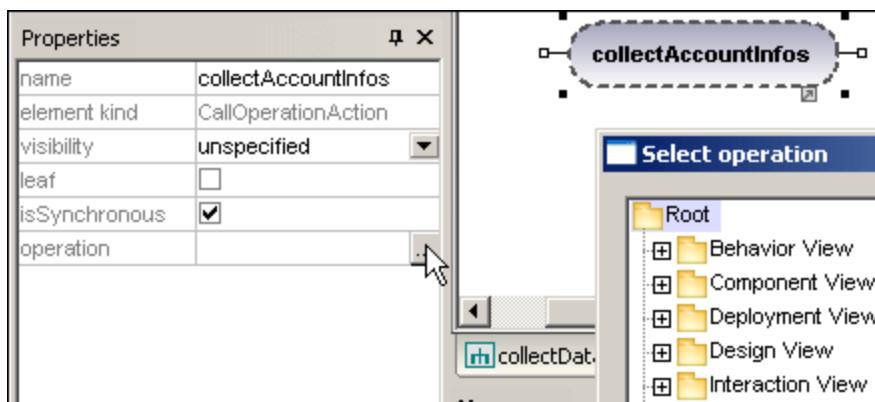
1. Click the **Action (CallBehavior)**  toolbar button, and click in the Activity diagram to insert it.
2. Enter the name of the Action, e.g. "Validate References", and press **Enter** to confirm.



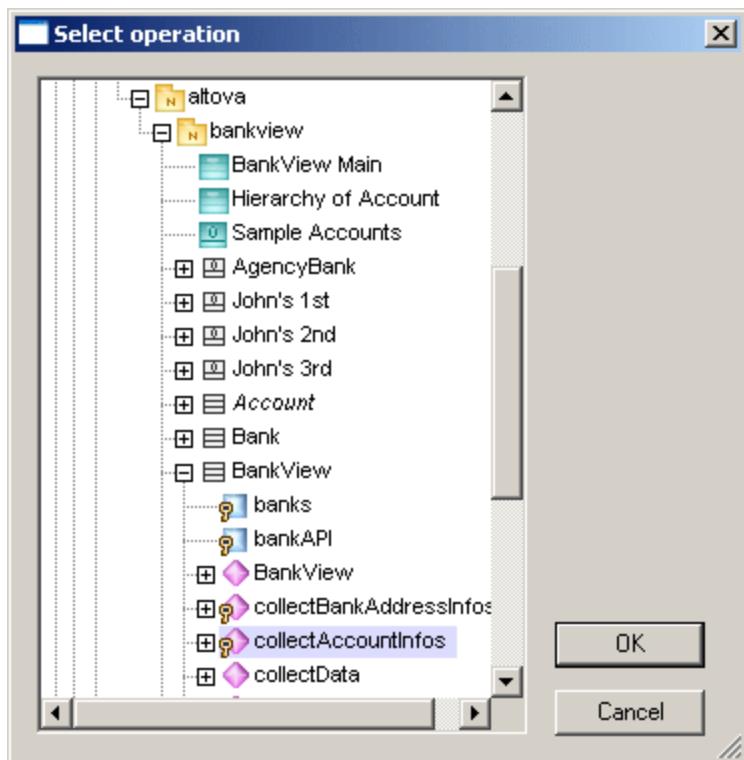
**Note:** Use **Ctrl+Enter** to create a multi-line name.

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

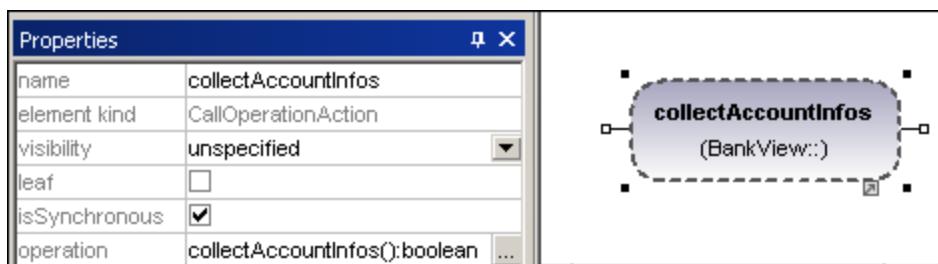
1. Click the **Action (CallOperation)** icon  in the icon bar, and click in the Activity diagram to insert it.
2. Enter the name of the Action, e.g. "collectAccountInfo", and press **Enter** to confirm.
3. Click the **Browse** button to the right of the operation field in the **Properties** tab. This opens the "Select Operation" dialog box in which you can select the specific operation.



4. Navigate to the specific operation that you want to insert, and click **OK** to confirm.



In this example, the operation "collectAccountInfos" is in the `BankView` class.



### 8.1.1.2 Creating branches and merges

A branch has a single incoming flow and multiple outgoing guarded flows. Only one of the outgoing flows can be traversed, so the guards should be mutually exclusive.

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

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

#### Creating a branch (alternate flow)

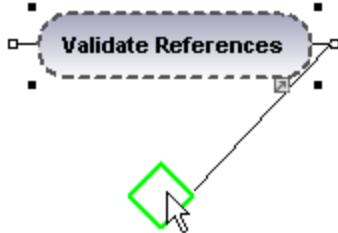
1. Click the **DecisionNode** icon  in the title bar, and insert it in the Activity diagram.

**Validate References**

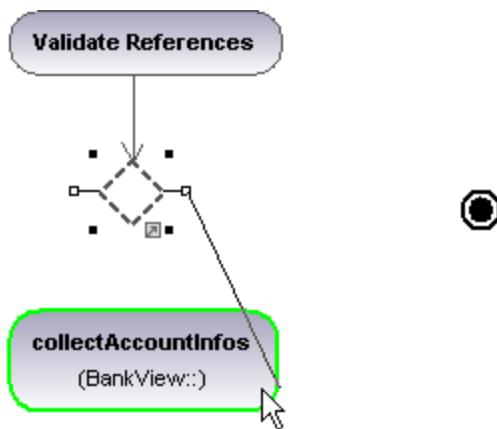


**collectAccountInfos**  
(BankView::)

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](#) 292" for more information.

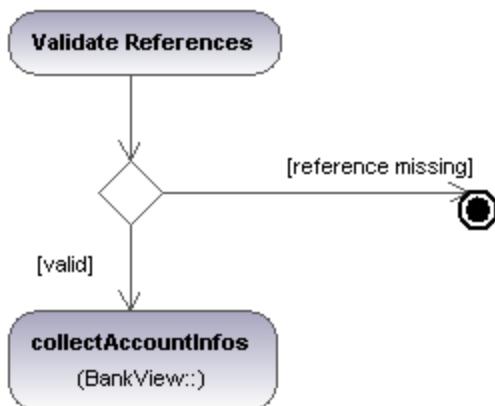


5. Enter the guard condition "valid", in the guard field of the **Properties** tab.

Properties	
name	
element kind	ControlFlow
visibility	unspecified
leaf	<input type="checkbox"/>
guard	valid
weight	
isMultiCast	<input type="checkbox"/>
isMultiReceive	<input type="checkbox"/>
selection	
transformation	

To the right of the properties table is the original Activity Diagram from step 4, showing the "collectAccountInfos" node with its guard condition set to "valid".

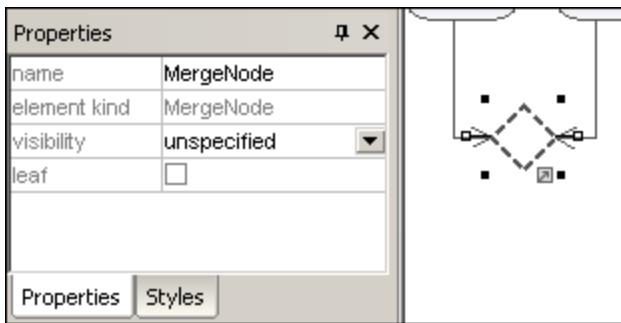
6. Click the DecisionNode element and drag from the right-hand handle, **ControlFlow**, and drop it on the "ActivityFinalNode" element. The guard condition on this transition is automatically defined as "else". Double click the guard condition in the diagram to change it e.g. "reference missing".



**Note:** UModel does not validate, or check, the number of Control/Object Flows in a diagram.

## Creating a merge

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



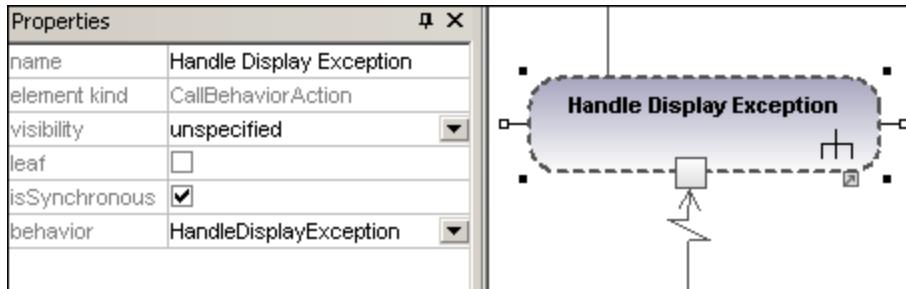
2. Click the ControlFlow (ObjectFlow) handles of the actions that are to be merged, and drop the arrow(s) on the "MergeNode" symbol.

### 8.1.1.3 Activity Diagram elements



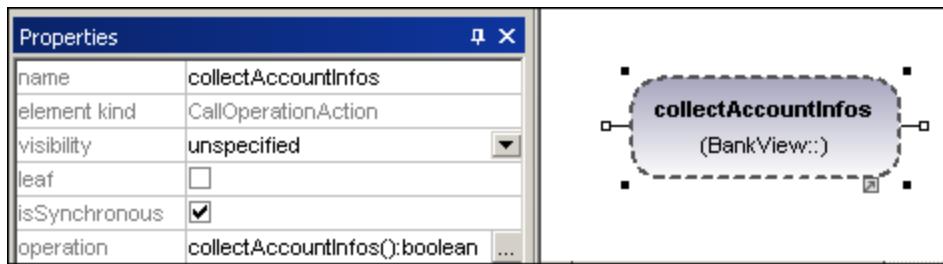
#### Action (CallBehavior)

Inserts a **CallBehaviorAction** element which directly invokes a specific behavior. Selecting an existing behavior using the **behavior** combo box, e.g. HandleDisplayException, displays a rake symbol within the element.



#### Action (CallOperation)

Inserts a **CallOperationAction** which indirectly invokes a specific behavior as a method. Please see "[Inserting an action \(CallOperation\)](#)" for more information.



### Action (OpaqueAction)

A type of action used to specify implementation information. Can be used as a placeholder until you decide which specific action type you want to use.



### Action (ValueSpecificationAction)

A type of action that evaluates(/generates) a specific value at the output pin. (Defined by the specific properties, e.g. upperBound.)



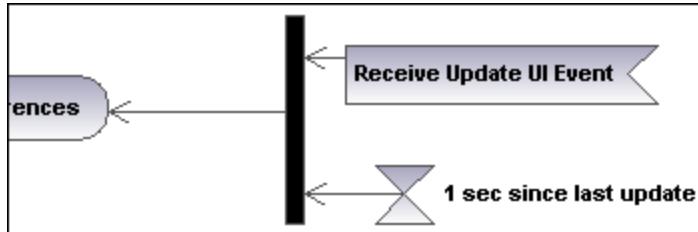
### AcceptEventAction

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



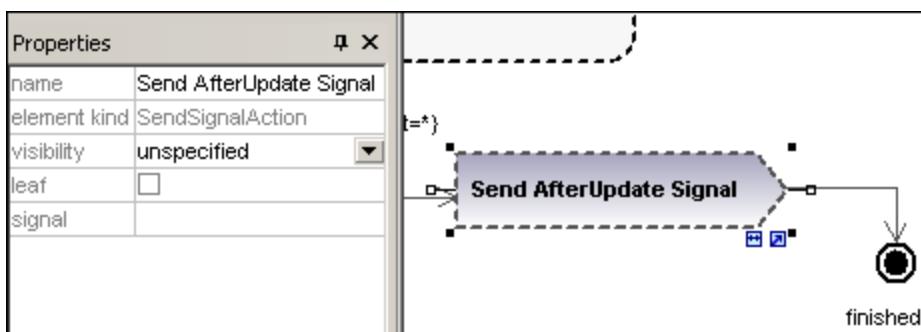
### AcceptEventAction (TimeEvent)

Inserts an **AcceptEventAction**, triggered by a time event, which specifies an instant of time by an expression e.g. 1 sec. since last update.



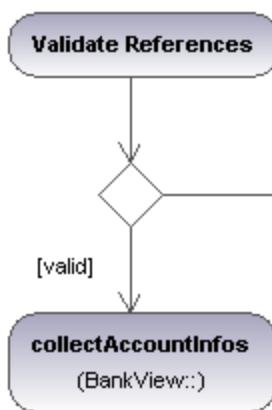
### SendSignalAction

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



### 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 than 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.



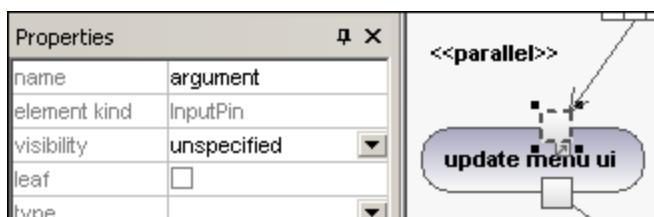
### Join Node (horizontal)

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.

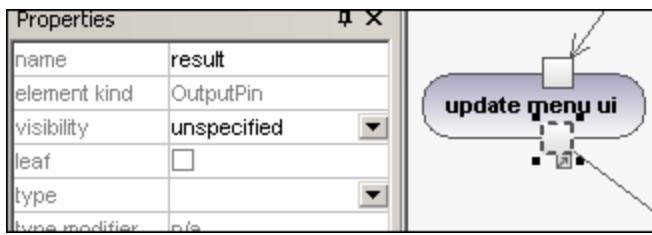


The input pin symbol can only be placed onto those activity elements where the mouse pointer changes to the hand symbol . Dragging the symbol repositions it on the element border.



### 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.

## Exception Pin

An OutputPin can be changed to an Exception pin by clicking the pin and selecting "isExceptionPin" from the Properties pane.



### ValuePin

Inserts a Value Pin which is an input pin that provides a value to an action, that does not come from an incoming object flow. It is displayed as an input pin symbol, and has the same properties as an input pin.



### ObjectNode

Inserts an object node which is an abstract activity node that defines object flow in an activity. Object nodes can only contain values at runtime that conform to the type of the object node.



### CentralBufferNode

Inserts a Central Buffer Node which acts as a buffer for multiple in- and out flows from other object nodes.



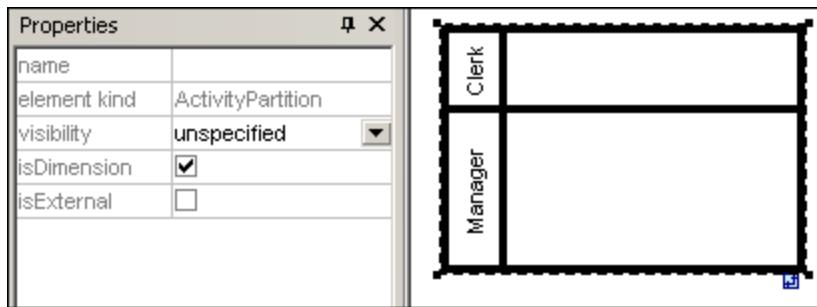
### DataStoreNode

Inserts a Data Store Node which is a special "Central Buffer Node" used to store persistent (i.e. non transient) data.



### ActivityPartition (horizontal)

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



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

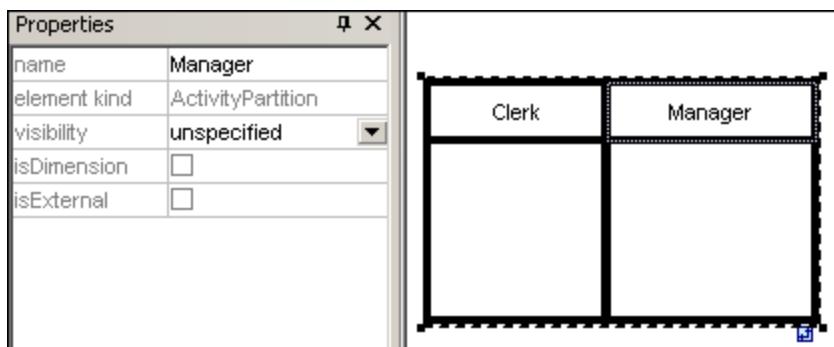
Please note that Activity Partitions are the UML 2.0 update to the "swimlane" functionality of previous UML versions.

- Elements placed within a ActivityPartition become part of it when the boundary is highlighted.
- Objects within an ActivityPartition can be individually selected using **Ctrl+Click**, or by dragging the marquee inside the boundary.
- Click the ActivityPartition boundary, or title, and drag to reposition it.



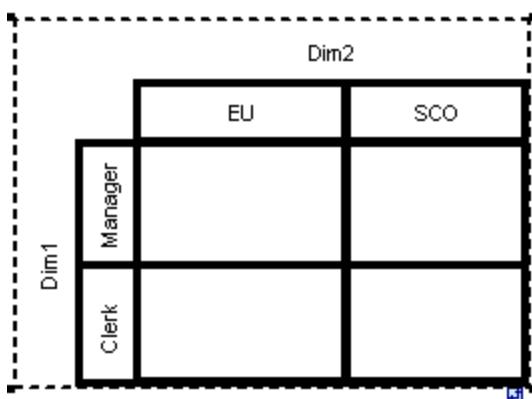
### ActivityPartition (vertical)

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



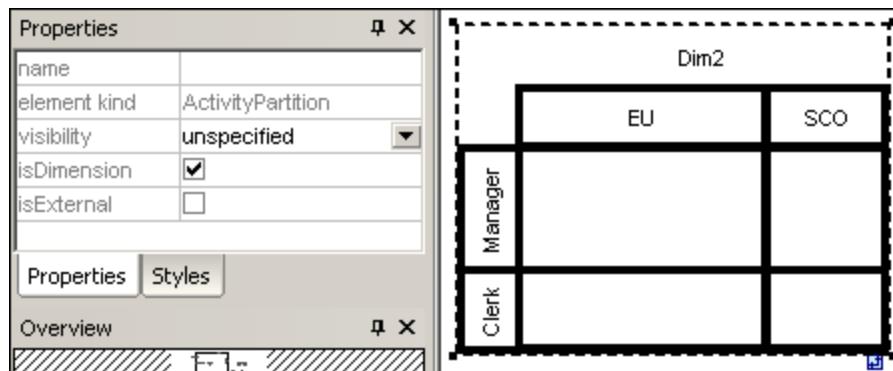
### ActivityPartition (2 Dimensional)

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



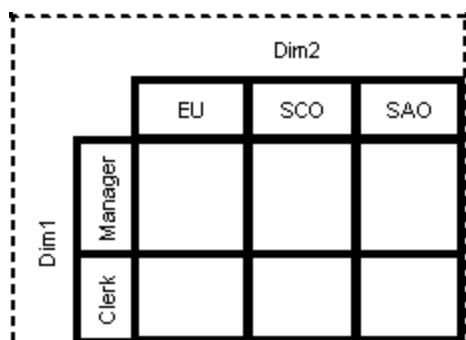
To remove the Dim1, Dim2 dimension labels:

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:

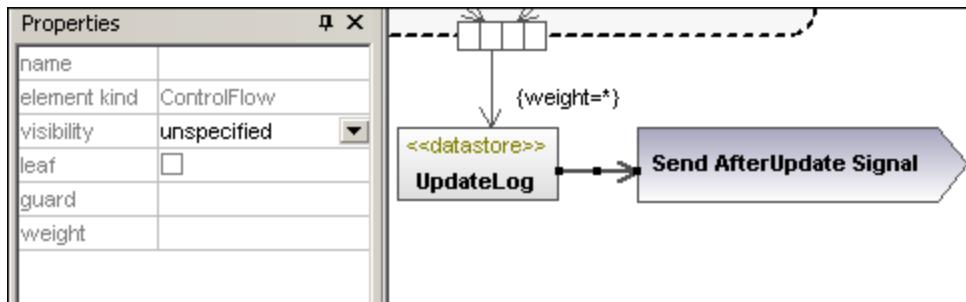
1. Right click the label where you want to insert a new partition.
2. Select **New | ActivityPartition**.





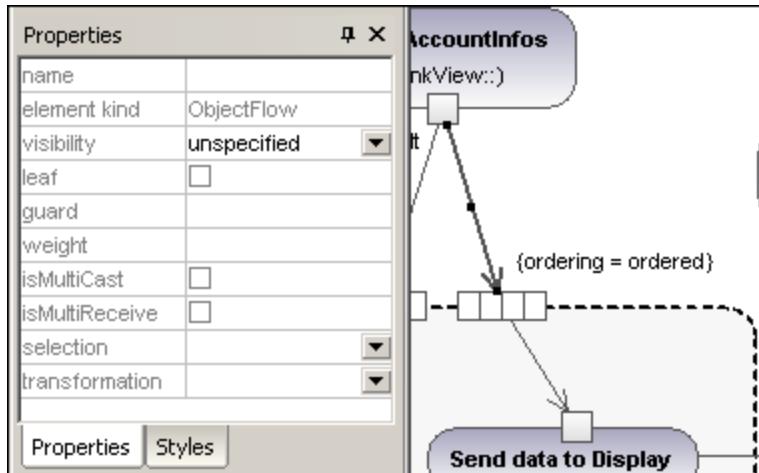
### ControlFlow

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.



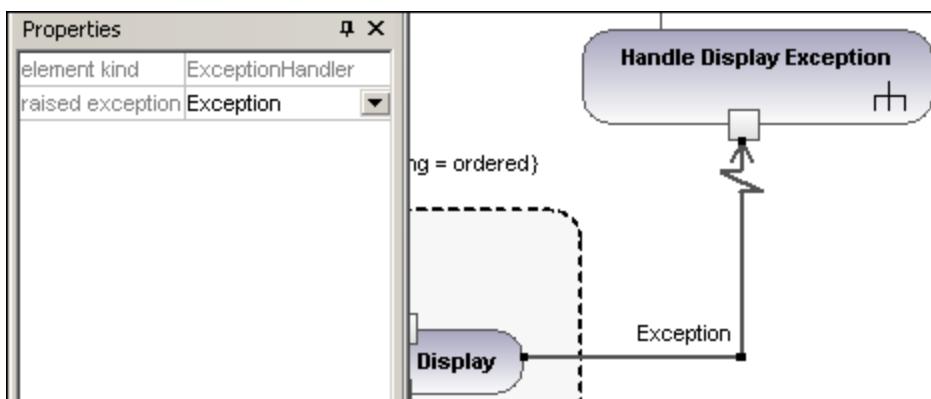
### ObjectFlow

An 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.

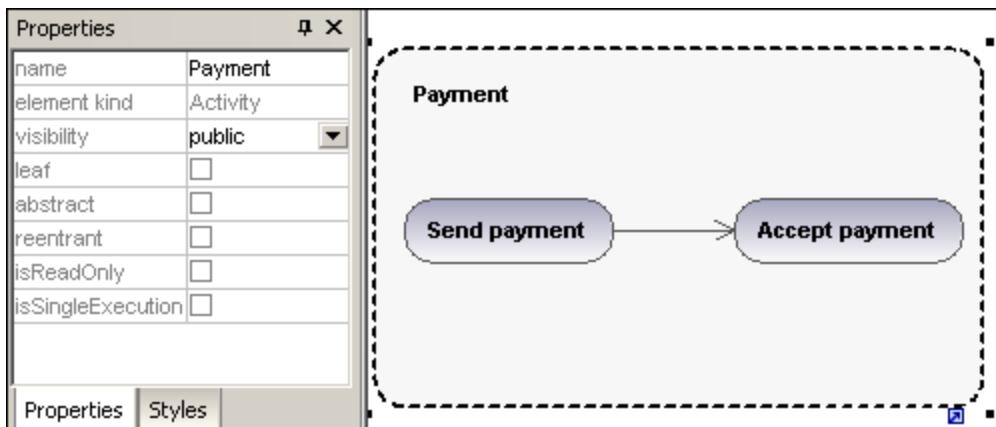


An Exception Handler can only be dropped on an Input Pin of an Action.



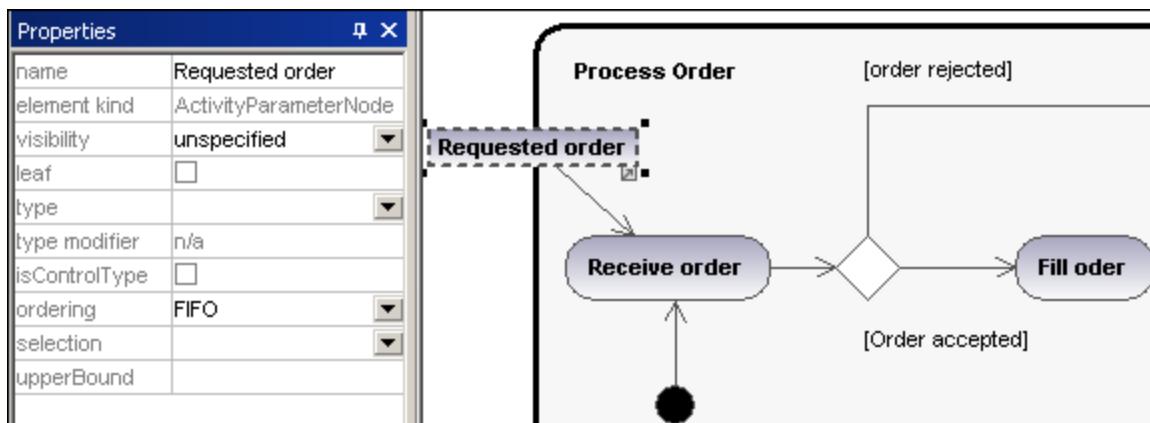
### Activity

Inserts an Activity into the activity diagram.



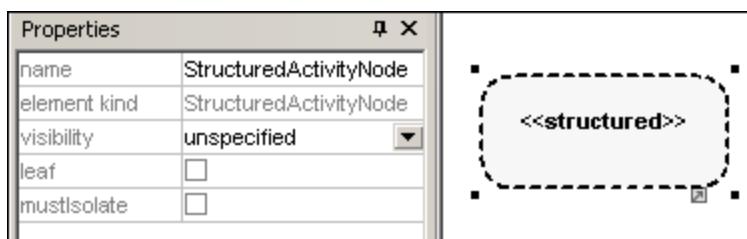
### ActivityParameterNode

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



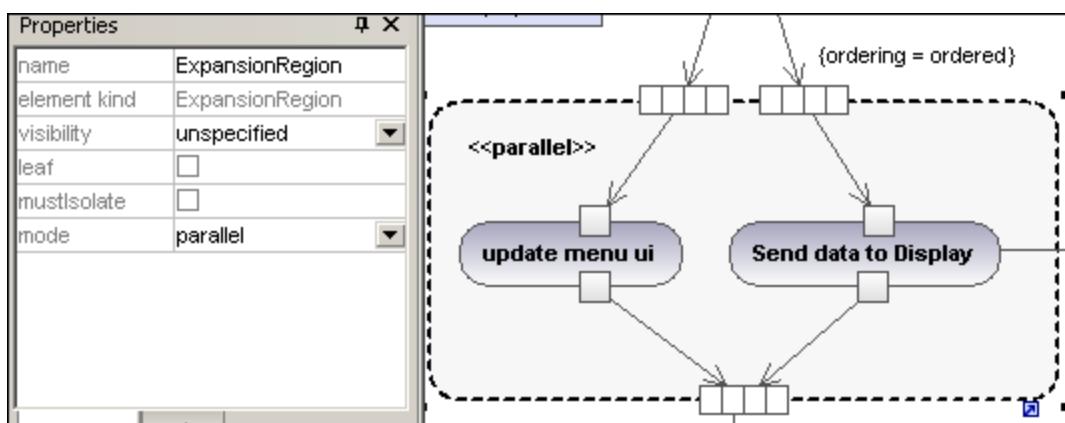
### StructuredActivityNode

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.

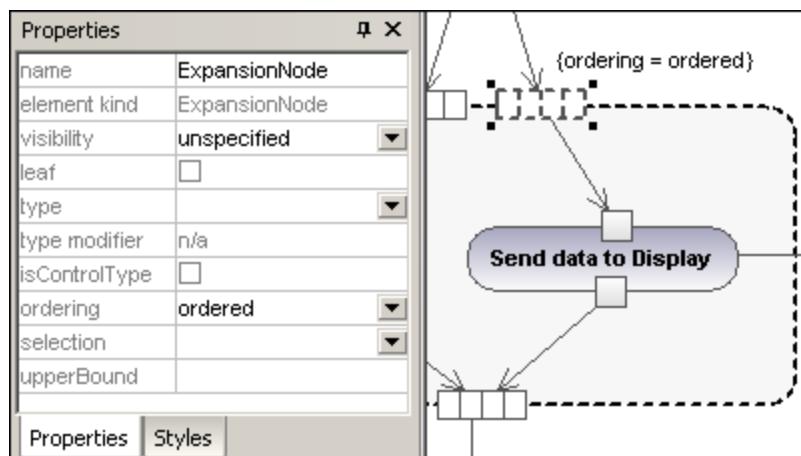


The expansion region mode is displayed as a keyword, and can be changed by clicking the "mode" combo box in the Properties tab. Available settings are:parallel, iterative, or stream.



### 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.

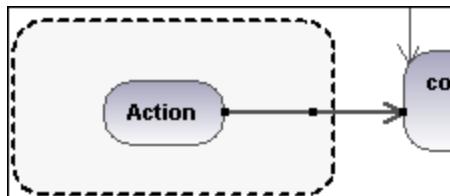


### InterruptableActivityRegion

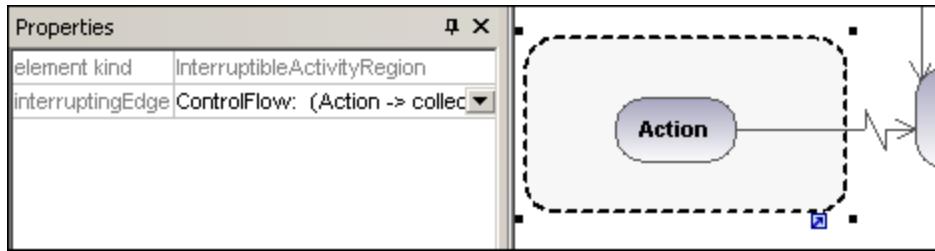
An interruptible region contains activity nodes. When a control flow leaves an interruptible region all flows and behaviors in the region are terminated.

#### To add an interrupting edge:

1. Make sure that an Action element is present in the InterruptableActivityRegion, as well as an outgoing Control Flow to another action:



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



**Note:** You can also add an InterruptingEdge by clicking the InterruptableActivityRegion, right clicking in the Properties window, and selecting Add InterruptingEdge from the pop-up menu.

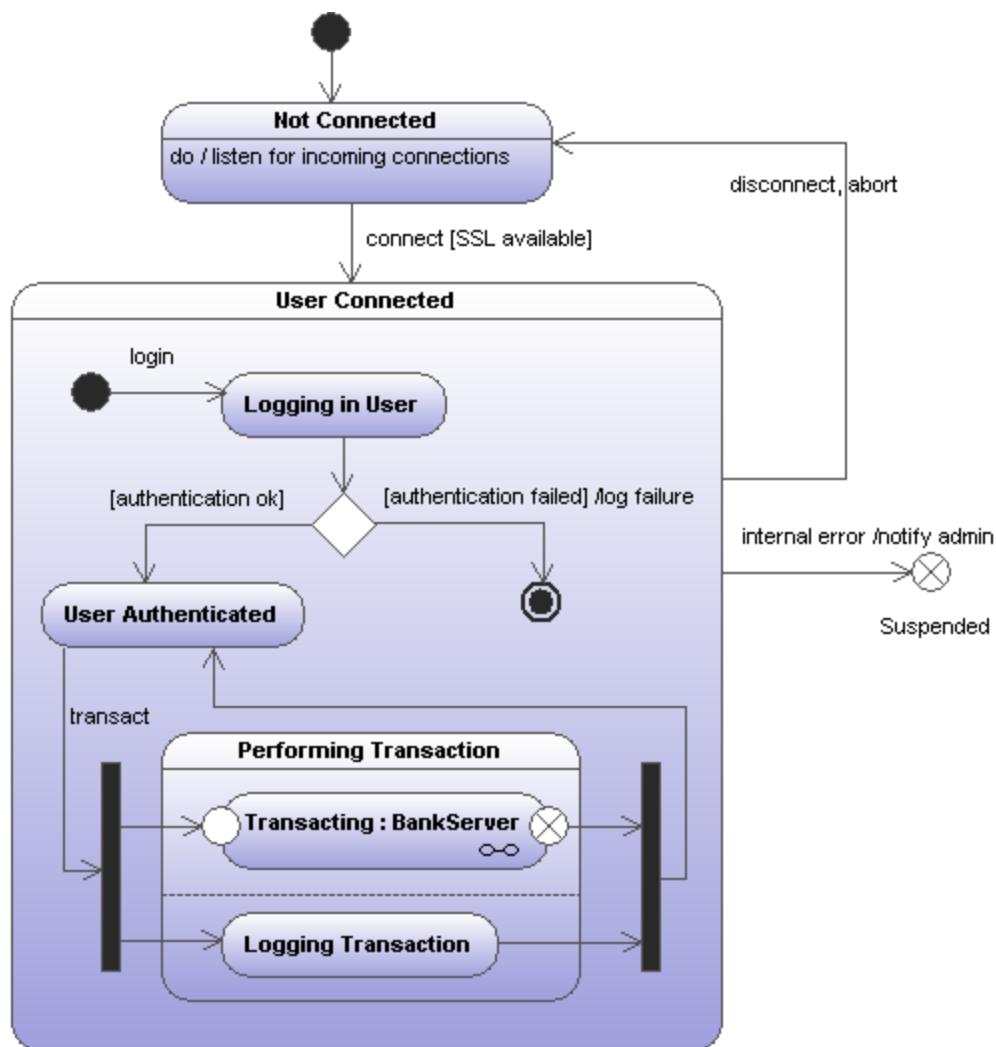
### 8.1.2 State Machine Diagram

The State Machine Diagram models the behavior of a system by describing the various states an object may be in, and the transitions between those states. They are generally used to describe the behavior of an object spanning several use cases.

Two types of processes can achieve this:

1. **Actions**, which are associated to **transitions**, are short-term processes that cannot be interrupted (for example, **internal error /notify admin** in the diagram below)
2. State **Activities** (behaviors), which are associated to **states**, are longer-term processes that may be interrupted by other events (for example, **listen for incoming connections**, in the diagram below).

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



*Sample State Machine diagram*

The State machine diagram illustrated above is available in the following sample UModel project: C:\Users\<username>\Documents\Altova\UModel2022\UModel\Examples\Bank\_MultiLanguage.ump.

### 8.1.2.1 Inserting state machine diagram elements

#### To insert state machine diagram elements:

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



2. Click in the State Diagram to insert the element. To insert multiple elements of the selected type, hold down the **Ctrl** key and click in the diagram window.

## Dragging existing elements into the state machine diagram

Most elements occurring in other state machine diagrams can be inserted into an existing state machine.

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

### 8.1.2.2 Creating states, activities and transitions

#### To add a simple state:



1. Click the **State** toolbar icon ( ), and then click inside the diagram.
2. Enter the name of the state and press **Enter** to confirm.

#### To add an activity to a state:

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



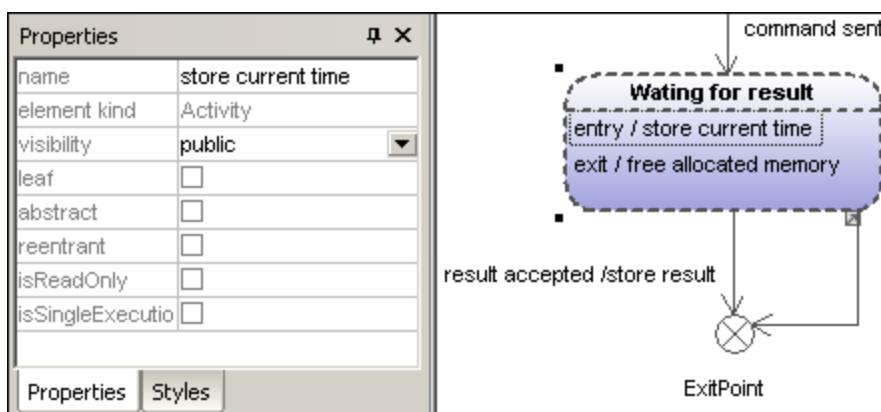
The **Entry**, **Exit**, and **Do** activities are associated with one of the following possible behaviors: "Activity", "Interaction", and "StateMachine". Therefore, the options available in the context menu are:

- Do: Activity
- Do: Interaction
- Do: StateMachine
- Entry: Activity
- Entry: Interaction

- Entry: StateMachine
- Exit: Activity
- Exit: Interaction
- Exit: StateMachine

These options originate in the UML specification. Namely, each of these internal actions are behaviors, and, in the UML specification, three classes derive from the "Behavior" class: Activity, StateMachine, and Interaction. In the generated code, it does not make a difference which particular behavior (Activity, StateMachine, or Interaction) has been selected.

You can select one action from the **Do**, **Entry** and **Exit** action categories. Activities are placed in their own compartment in the state element, though not in a separate region. The type of activity that you select is used as a prefix for the activity e.g. **entry / store current time**.

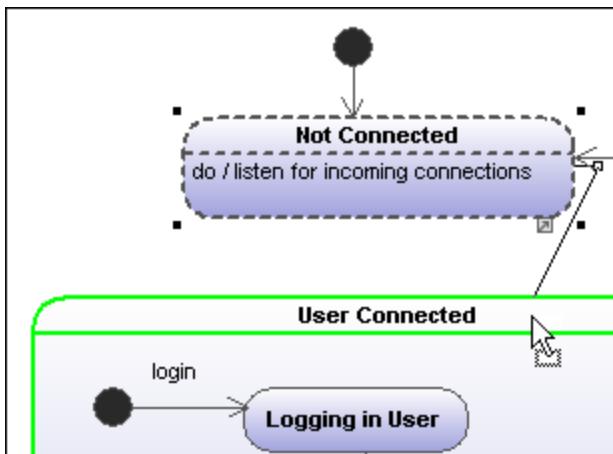


#### To delete an activity:

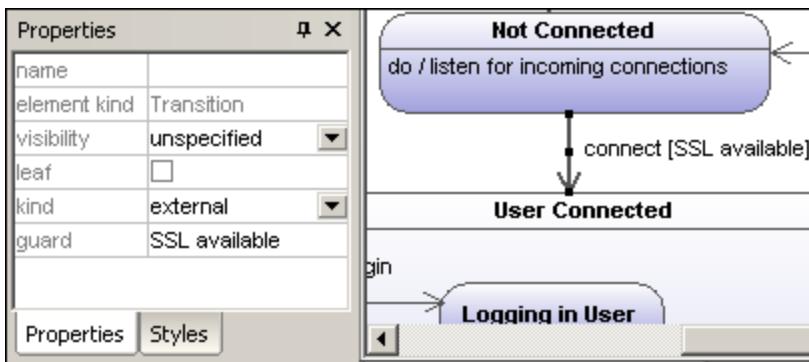
- Click the respective activity in the state element and press the **Del** key.

#### To create a transition between two states:

1. Click the Transition handle of the source state (on the right of the element).
2. Drag-and-drop the transition arrow onto the target state.



The Transition properties are now visible in the **Properties** tab. Clicking the "kind" combo box, allows you to define the transition type: external, internal or local.



Transitions can have an event trigger, a guard condition and an action in the form **eventTrigger [guard condition] /activity**.

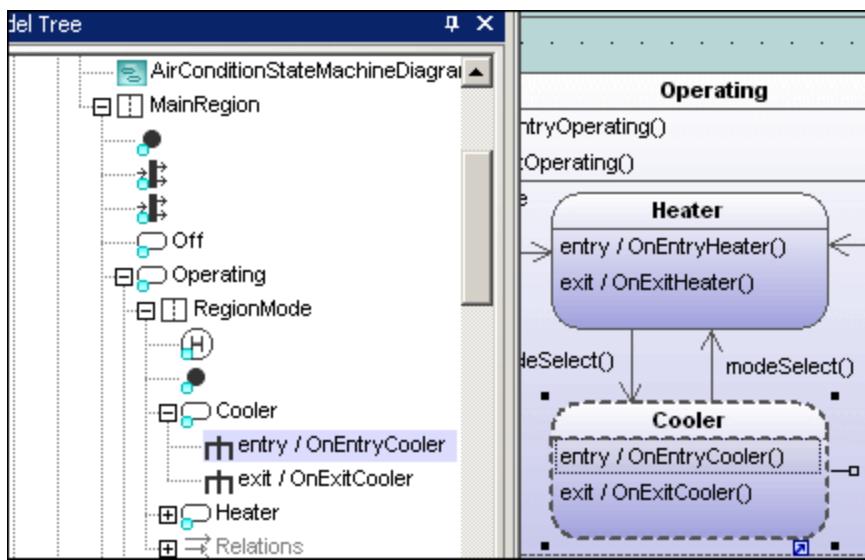
#### To automatically create operations from transitions:

Activating the "Toggle automatic creation of operations in target by typing operation names" icon , automatically creates the corresponding operation in the referenced class, when creating a transition and entering a name e.g. myOperation().

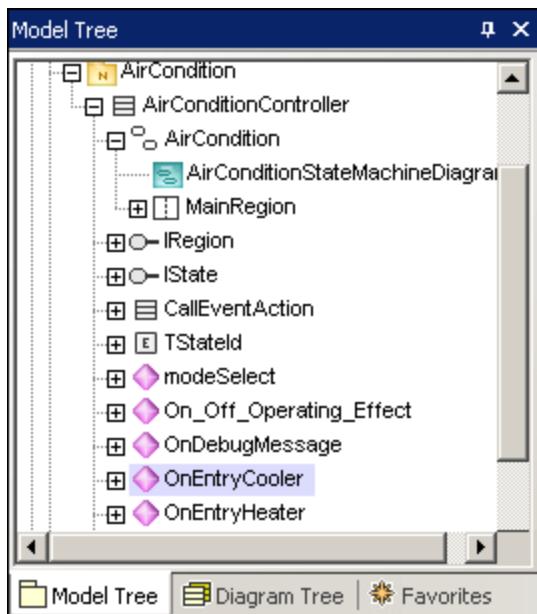
**Note:** Operations can only be created automatically when the state machine is inside a class or interface.

#### To automatically create operations from activities:

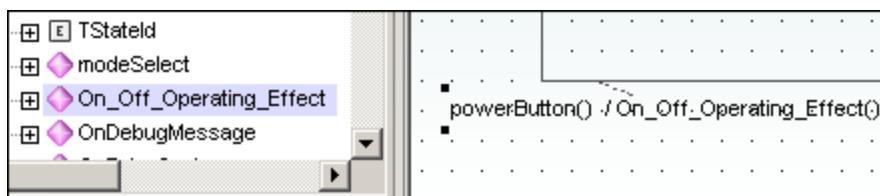
1. Right click the State and select the specific action/activity, e.g. New | Entry:Activity.
2. Enter the name of the activity making sure to finish with the open/close brackets "()", e.g. **entry / OnEntryCooler()**.



The new element is also visible in the Model Tree. Scrolling down the Model Tree, you will notice that the OnEntryCooler operation has been added to the parent class AirConditionController.



**Note:** Operations are automatically added for: Do:Activity, Entry:Activity, Exit:Activity, as well as guard condition activities and effects (on transitions).



### To create a transition trigger:

1. Right-click a previously created transition (arrow).
2. Select **New | Trigger**.



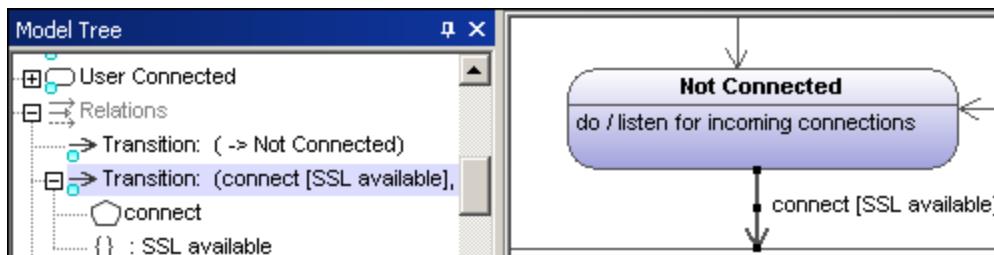
An "a" character appears in the transition label above the transition arrow, if it is the first trigger in the state diagram. Triggers are assigned default values of the form alphabetic letter, source state -> target state.

3. Double-click the new character and enter the transition properties in the form **eventTrigger [guard condition] / activity**.

#### Transition property syntax

The text entered before the square brackets is the trigger; the text between brackets is the guard condition, and the text after the slash—the activity. Manipulating this string automatically creates or deletes the respective elements in the Model Tree.

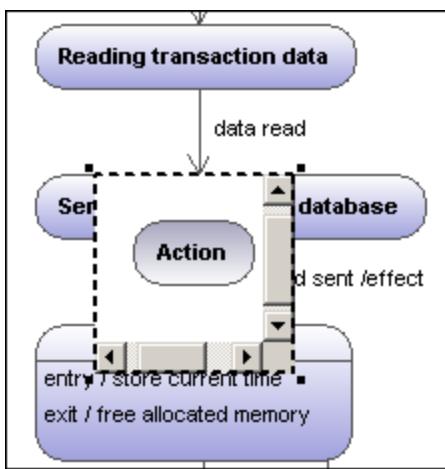
**Note:** To see the individual transition properties, right-click the transition (arrow) and select "Select in Model Tree". The event, activity and constraint elements are all shown below the selected transition.



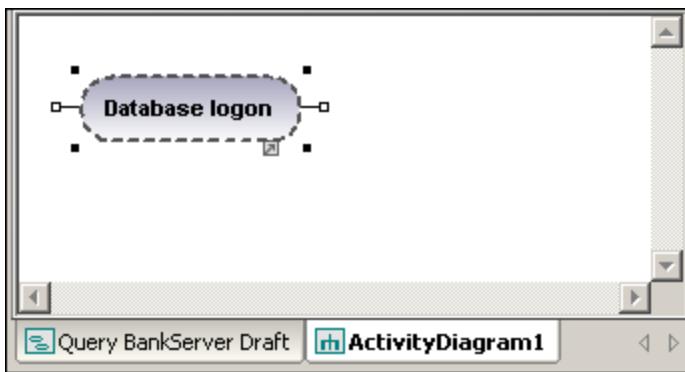
### Adding an Activity diagram to a transition

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

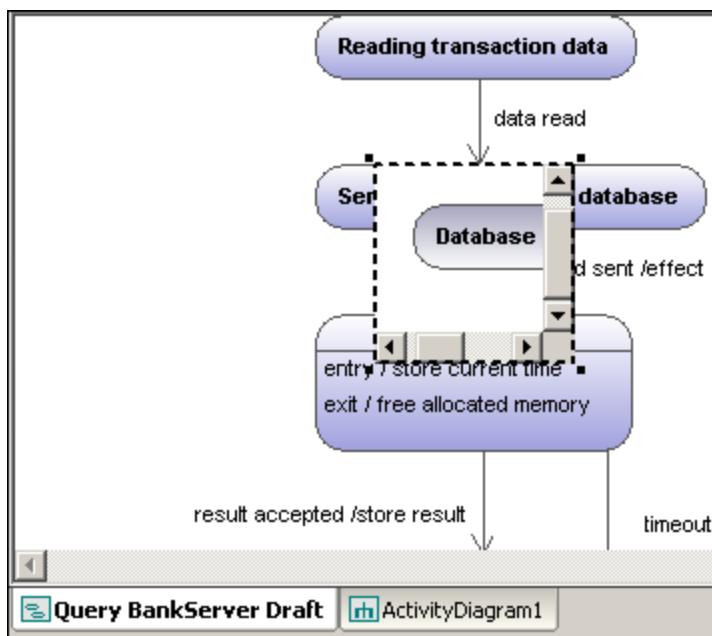
1. Right-click a transition arrow in the diagram, and select **New | Activity Diagram**. This inserts an Activity diagram window into the diagram at the position of the transition arrow.
2. Click the inserted window to make it active. You can now use the scroll bars to scroll within the window.



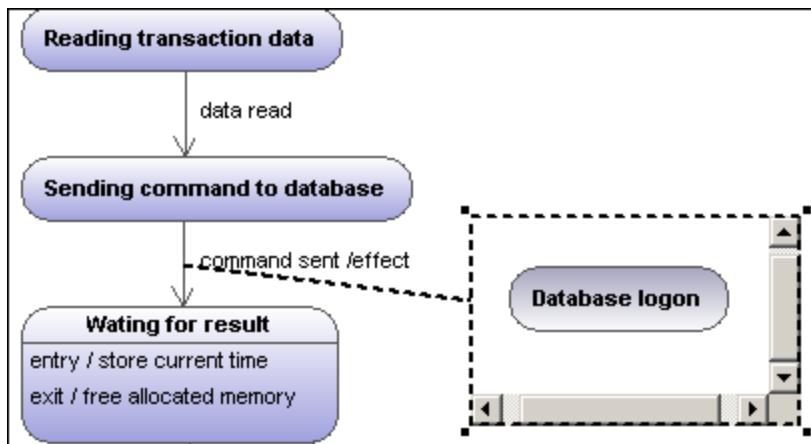
3. Double-click the Action window to switch into the Activity diagram and further define the transition, e.g. change the Action name to "Database logon". Note that a new **Activity Diagram** tab has now been added to the project. You can add any activity modeling elements to the diagram, please see "[Activity Diagram](#)"<sup>290</sup>" for more information.



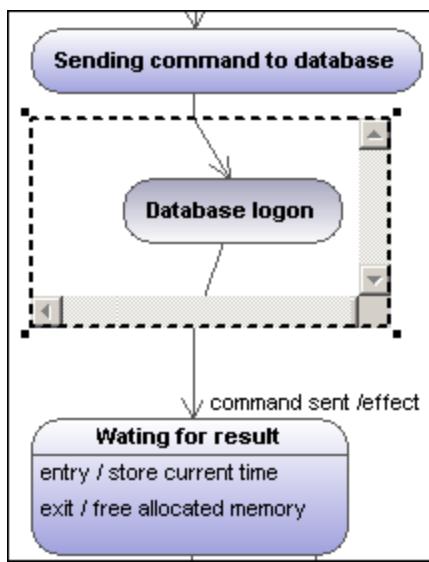
4. Click the **State Machine Diagram** tab to switch back to see the updated transition.



5. Drag the Activity window to reposition it in the diagram, and click the resize handle if necessary.



Dragging the Activity window between the two states displays the transition in and out of the activity.



### 8.1.2.3 Composite states



#### Composite state

This type of state contains a second compartment comprised of a single region. Any number of states may be placed within this region.

##### To add a region to a composite state:

- Right-click the composite state and select **New | Region** from the context menu. A new region is added to the state. Regions are divided by dashed lines.

##### To delete a region:

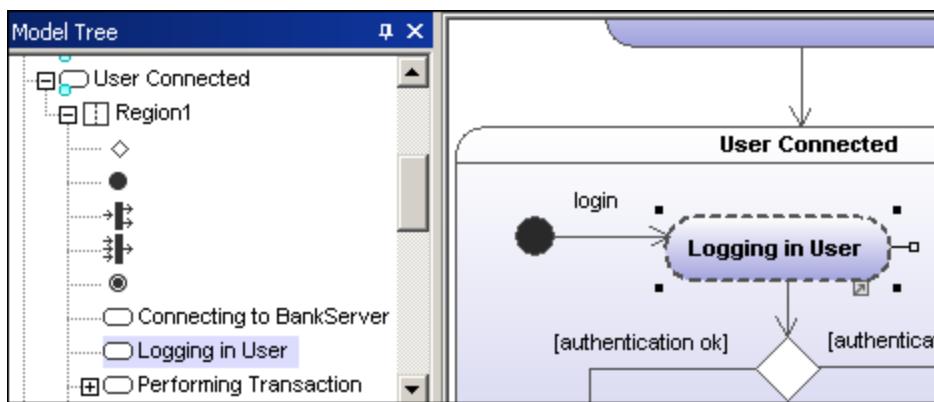
- Click the region you want to delete in the composite state and press the **Del** key.

Deleting a region of an orthogonal state reverts it back to a composite state; deleting the last region of a composite state changes it back to a simple state.

##### To place a state within a composite state:

- Click the state element you want to insert (e.g. Logging in User), and drop it into the region compartment of the composite state.

The region compartment is highlighted when you can drop the element. The inserted element is now part of the region, and appears as a child element of the region in the Model Tree pane.



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



### Orthogonal state

This type of state contains a second compartment comprised of two or more regions, where the separate regions indicate concurrency.

Right clicking a state and selecting **New | Region** allows you add new regions.



### To show/hide region names:

- Click the **Styles** tab, scroll to the "Show region names on states" entry, and select true/false.

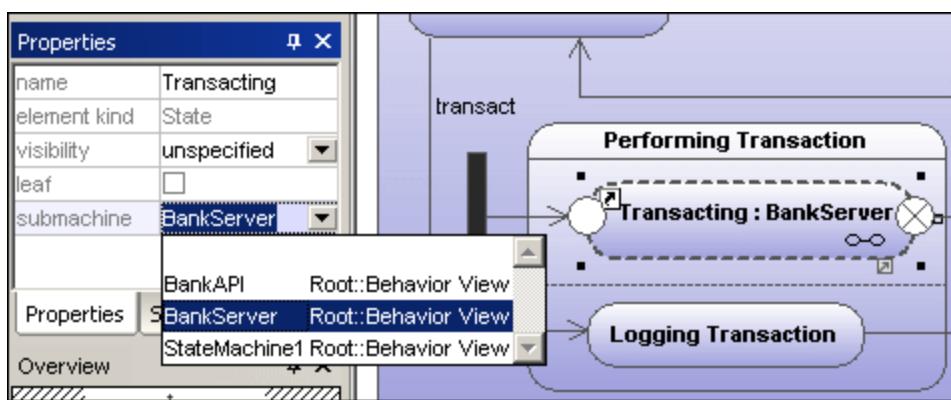


### Submachine state

This state is used to hide details of a state machine. This state does not have any regions but is associated to a separate state machine.

### To define a submachine state:

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



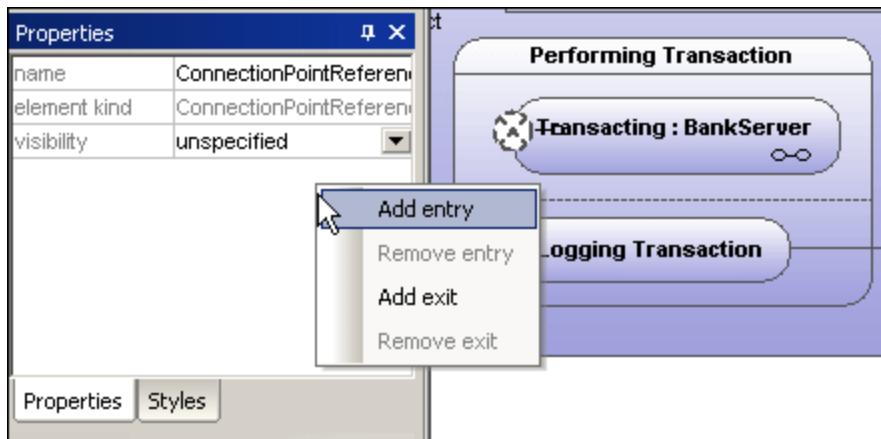
A hyperlink icon automatically appears in the submachine. Clicking it opens the referenced state machine, "BankServer" in this case.

#### To add entry / exit points to a submachine state:

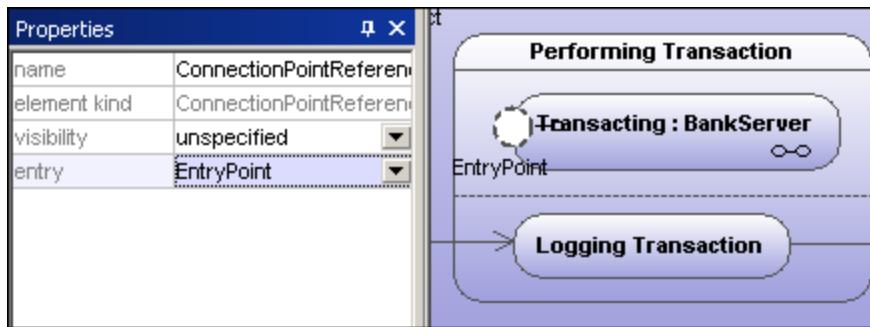
- The state which the point is connected to, must itself reference a submachine State Machine (visible in the Properties tab).
  - This submachine must contain one or more Entry and Exit points
- Click the **ConnectionPointReference** icon  in the title bar, then click the submachine state that you want to add the entry/exit point to.



- Right-click in the **Properties** tab and select **Add entry**. Please note that another Entry, or Exit Point has to exist elsewhere in the diagram to enable this pop-up menu.



This adds an **EntryPoint** row to the **Properties** tab, and changes the appearance of the **ConnectionPointReference** element.



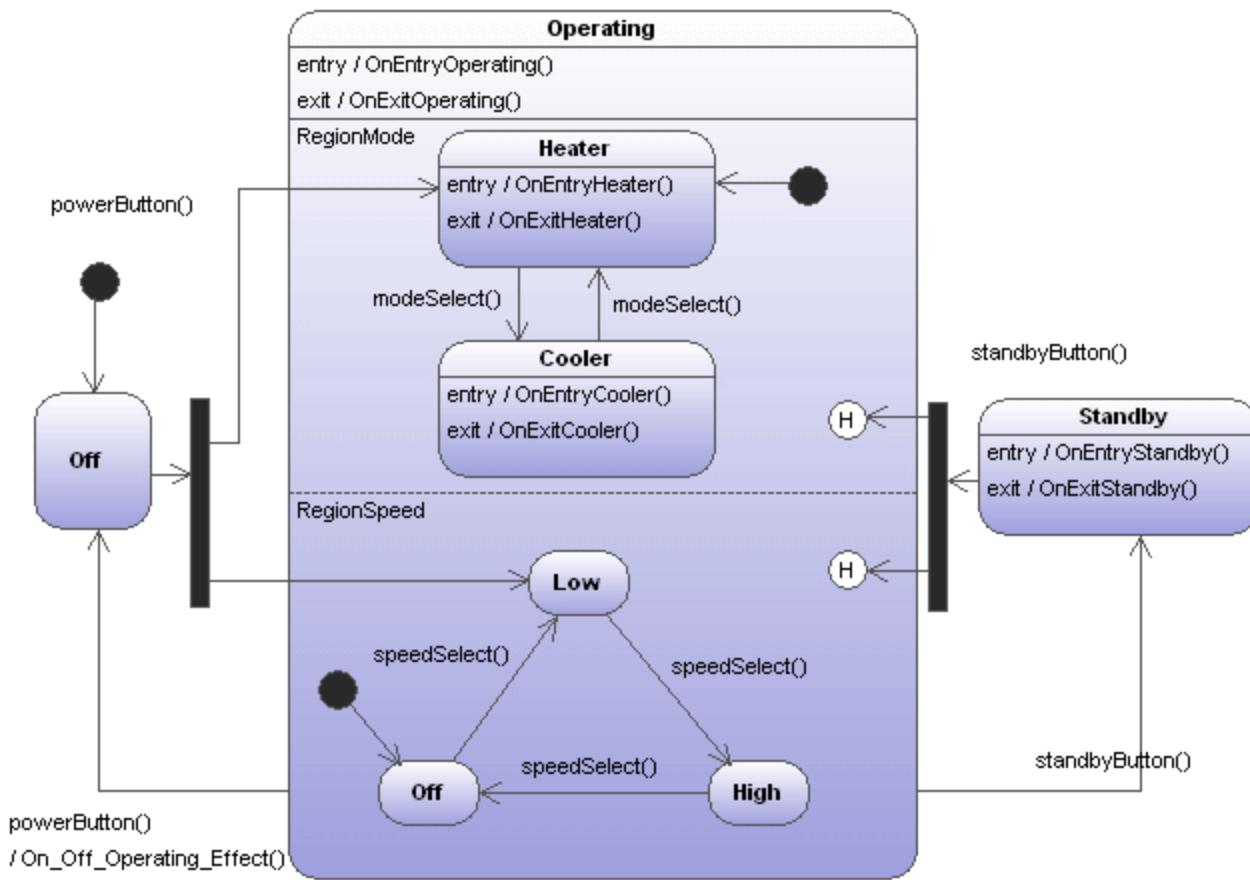
3. Use the same method to insert an ExitPoint, by selecting "Add exit" from the context menu.

#### 8.1.2.4 Generating code from State Machine diagrams

UModel can generate executable code from State Machine diagrams (C#, Java, VB.NET). Almost all of the State Machine diagram elements and features are supported:

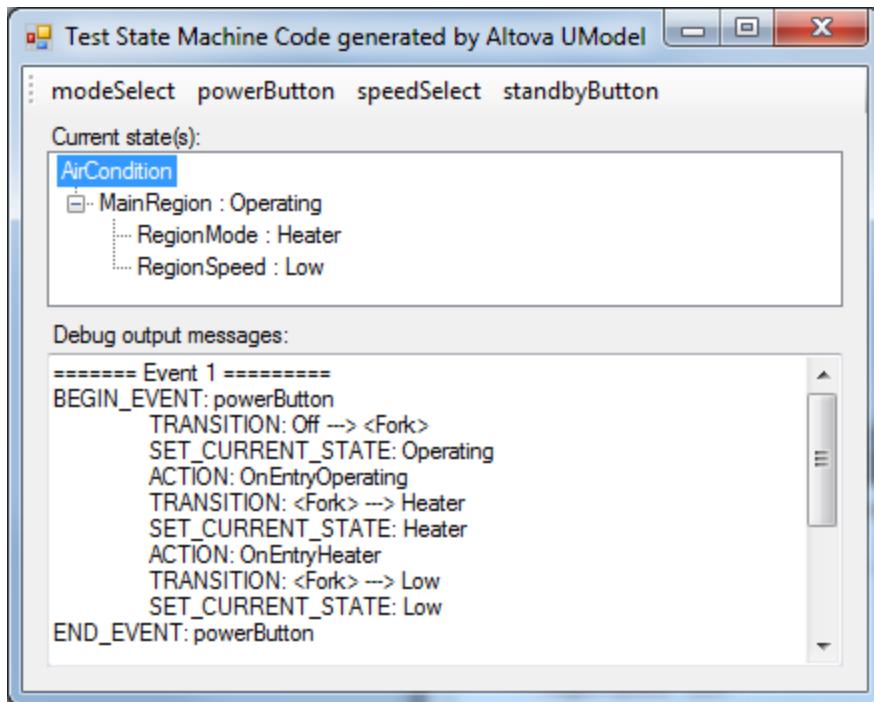
- State
- CompositeState, with any hierarchical level
- OrthogonalState, with any number of regions
- Region
- InitialState
- FinalState
- Transition
- Guard
- Trigger
- Call-Event
- Fork
- Join
- Choice
- Junction
- DeepHistory
- ShallowHistory
- Entry/exit/do actions
- Effects

State Machine code generation is integrated into the "normal" round-trip engineering process. This means that State Machine code can be automatically updated on every forward-engineering process.



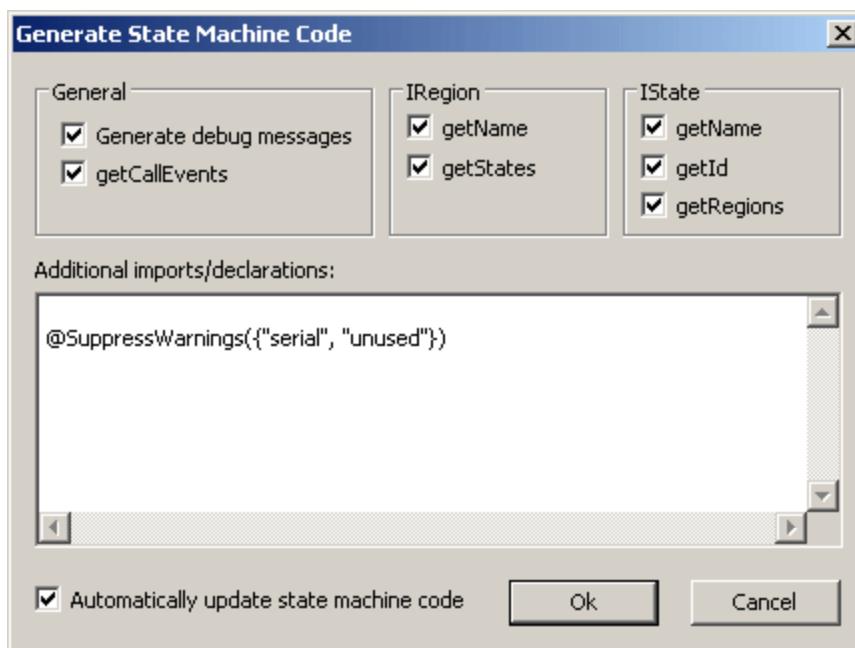
Each directory contains an AirCondition and Complex folder, which contains the respective UModel project, programming language project files, as well as the generated source files. The Complex.ump project file contains almost all of the modeling elements and functionality that UModel supports when generating code from State Machine diagrams.

Each directory also contains a test application, e.g. TestSTMAirCondition.sln for C#, allowing you to work with the generated source files immediately.



#### To generate code from a State Machine diagram:

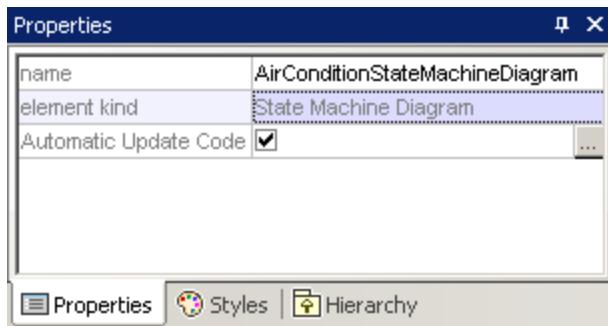
- Right-click in the State Machine diagram and select "Generate State Machine code", or
- Select the menu option **Project | Generate State Machine Code**



The default settings are shown above. Click OK to generate the code.

State Machine code is automatically updated when you start the forward engineering process. You can however change this setting by clicking on the State Machine diagram background and clicking the "Automatic Update Code" check box.

Changes should not be made manually in the generated code, as these changes are not reflected in the State Machine diagram during the reverse-engineering process.



Clicking the icon of the Automatic Update field, opens the Generate State Machine Code dialog box, allowing you to change the code generation settings.

#### To perform a syntax check on a State Machine diagram:

- Right-click the diagram and selecting **Check State Machine Syntax** from the context menu.

### 8.1.2.5 Working with state machine code

The parent class of the state machine (i.e. the "controller class", or "context class") is the one, and only, "interface" between the state machine user and the state machine implementation.

The controller class provides methods which can be used from "outside" to change the states (e.g. after external events occur).

The state machine implementation however, calls controller class methods ("callbacks") to inform the state machine user about state changes (OnEntry, OnExit, ...), transition effects, and the possibility to override and implement methods for conditions (guards).

UModel can automatically create simple operations (without any parameter) for entry/exit/do behaviors, transition effects, ... when the corresponding option is turned on (also see [Creating states, activities and transitions](#)<sup>309</sup>). These methods can be changed to whatever you want in UModel (add parameters, set them as abstract, etc.).

A state machine (i.e. its controller class) can be instantiated several times. All instances work independently of each other.

- The UML State machine execution is designed for the "Run-to-completion execution model".
- UML state machines assume that processing of each event is completed before the next event is processed.

- This also means no entry/exit/do action or transition effect may directly trigger a new transition/state change.

## Initialization

- Every region of a state machine has to have an initial state.
- The code generated by UModel automatically initializes all regions of the state machine (or when the `Initialize()` method of the controller class is called).
- If OnEntry events are not wanted during initialization, you can call the `Initialize()` method manually and ignore OnEntry events during the startup.

## Getting the current state(s)

UModel supports composite states as well as orthogonal states, so there is not just one current state—every region (in any hierarchy level) can have one current state.

The `AirCondition.ump` example shows how to walk through the regions to the current state(s):

```
TreeNode rootNode = m_CurrentStateTree.Nodes.Add(m_STM.getRootState().getName());
UpdateCurrentStateTree(m_STM.getRootState(), rootNode);

private void UpdateCurrentStateTree(AirCondition.AirConditionController.IState state,
TreeNode node)
{
    foreach (AirCondition.AirConditionController.IRegion r in state.getRegions())
    {
        TreeNode childNode = node.Nodes.Add(r.getName() + " : " +
r.getCurrentState().getName());
        UpdateCurrentStateTree(r.getCurrentState(), childNode);
    }
}
```

## Example 1 - a simple transition



The corresponding operation is automatically generated in UModel



Generated method in code:

```
private class CTestStateMachine : IState
{
    ...
}
```

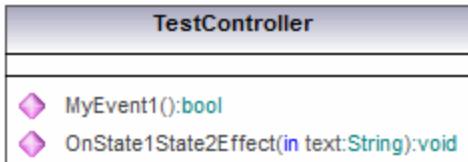
```
public bool MyEvent1()
{
    ...
}
```

**Notes:**

- The state machine user should call the generated method "MyEvent1" when the corresponding event occurs (outside the state machine).
- The return parameter of these event-methods provides information about whether the event caused a state change (i.e. if it had any effect on the state machine) or not. For example, if "State1" is active and event "MyEvent1()" occurs, the current state changes to "State2" and "MyEvent1()" returns true. If "State2" is active and "MyEvent1()" occurs, nothing changes in the state machine and MyEvent1() returns false.

**Example 2 - a simple transition with an effect**

The corresponding operation is automatically generated in UModel



Generated method in code:

```
private class CTestStateMachine : IState
{
    ...
    // Override to handle entry/exit/do actions, transition effects,....
    public virtual void OnState1State2Effect() { }
```

**Notes:**

- "OnState1State2Effect()" will be called by the state machine implementation, whenever the transition between "State1" and "State2" is fired.
- To react to this effect, "OnState1State2Effect()" should be overridden in a derived class of "CTestStateMachine".
- "CTestStateMachine:: OnState1State2Effect()" can also be set to abstract, and you will get compiler errors until the method is overridden.

- When "OnState1State2Effect()" is not abstract, and the "Generate debug messages" option is active, UModel will generate following debug output:

```
// Override to handle entry/exit/do actions, transition effects,....
public virtual void OnState1State2Effect() {OnDebugMessage("ACTION:
OnState1State2Effect");}
```

### Example 3 - a simple transition with an effect and parameter

MyEvent1() / OnState1State2Effect("1 => 2")



The corresponding operation is automatically generated in UModel



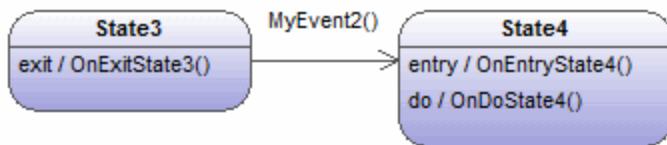
Generated method in code:

```
private class CTestStateMachine : IState
{
    ...
    // Additional defined operations of the controller class:
    public virtual void OnState1State2Effect(String text)
    {
    }
}
```

Notes:

- To effect operations (automatically created by UModel) parameters can be added manually (UModel cannot know the required type).
- In this sample, the parameter "text:String" has been added to the Effect method in TestController. A proper argument has to be specified when calling this method (here: "1 => 2").
- Another possibility would be: e.g. to call static methods ("MyStatic.OnState1State2Effect("1 => 2"))), or methods of singletons ("getSingleton().OnState1State2Effect("1 => 2")).

### Example 4 - entry/exit/do actions



The corresponding operations are automatically generated in UModel



Generated method in code:

```

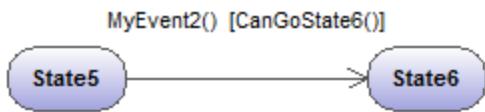
private class CTestStateMachine : IState
{
    ...
    // Override to handle entry/exit/do actions, transition effects,....
    public virtual void OnExitState3() {}
    public virtual void OnEntryState4() {}
    public virtual void OnDoState4() {}
}
  
```

Notes:

- States can have entry/exit/do behaviors. UModel automatically creates the corresponding operations to handle them.
- When "MyEvent2()" occurs in the sample above, the state machine implementation calls "OnExitState3()". If "MyEvent2" would have an Effect, it would be subsequently called, then "OnEntryState4" and "OnDoState4" would be called.
- Normally, these methods should be overridden. When they are not abstract and the "Generate debug messages" option is active, UModel provides default debug output as described in Example 2.
- These methods can also have parameters as shown in Example 3.

### Example 5 - guards

Transitions can have guards, which determine if the transition really can fire.



The corresponding operation is automatically generated in UModel

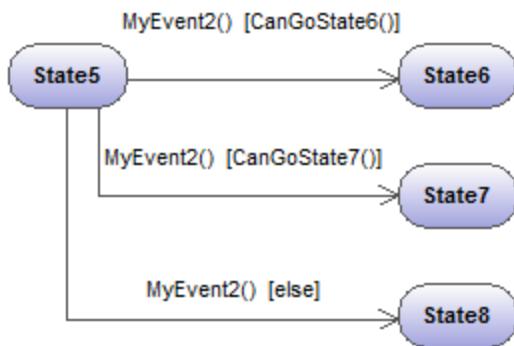


Generated method in code:

```
private class CTestStateMachine : IState
{
    ...
    // Additional defined operations of the controller class:
    public virtual bool CanGoState6()
    {
        return true; // Override!
    }
}
```

Notes:

- If "State5" is the active state and "MyEvent2" occurs, the state machine implementation will call "CanGoState6" and, depending on its result, the transition will fire or not.
- Normally, these methods should be overridden. When they are not abstract and the "Generate debug messages" option is active, UModel provides default debug output as described in Example 2.
- These methods also can have parameters as shown in Example 3.
- Multiple transitions with the same event, but having different guards, are possible. The order in which the different guards are polled is undefined. If a transition does not have a guard, or the guard is "else", it will be considered as the last (i.e., only when all other transition guards return false, will this one will fire). For example, in the diagram below, it is undefined whether `CanGoState6()` or `CanGoState7()` is called first. The third transition will only fire if `CanGoState6()` and `CanGoState7()` return false.



Additional constructs and functionality can be found in the **AirCondition.ump** and **Complex.ump** samples.

### 8.1.2.6 State Machine Diagram elements



#### InitialState (pseudostate)

The beginning of the process.



#### FinalState

The end of the sequence of processes.



#### EntryPoint (pseudostate)

The entry point of a state machine or composite state.



#### ExitPoint (pseudostate)

The exit point of a state machine or composite state.



#### Choice

This represents a dynamic conditional branch, where mutually exclusive guard triggers are evaluated (OR operation).



#### Junction (pseudostate)

This represents an end to the OR operation defined by the Choice element.



#### Terminate (pseudostate)

The halting of the execution of the state machine.



### Fork (pseudostate)

Inserts a vertical Fork bar. Used to divide sequences into concurrent subsequences.



### Fork horizontal (pseudostate)

Inserts a horizontal Fork bar. Used to divide sequences into concurrent subsequences.



### Join (pseudostate)

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



### Join horizontal (pseudostate)

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



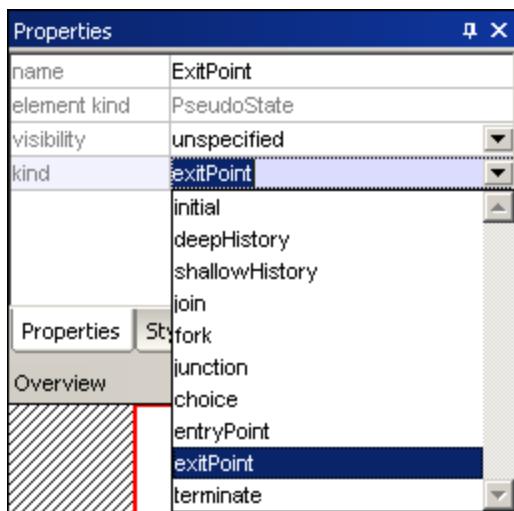
### DeepHistory

A pseudostate that restores the previously active state within a composite state.



### ShallowHistory

A pseudostate that restores the initial state of a composite state. All pseudostate elements can be changed to a different "type", by changing the **kind** combo box entry in the Properties tab.





### ConnectionPointReference

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

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

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



### Transition

A direct relationship between two states. An object in the first state performs one or more actions and then enters the second state depending on an event and the fulfillment of any guard conditions. Transitions have an event trigger, guard condition(s), an action (behavior), and a target state. The supported event subelements are:

- ReceiveSignalEvent
- SignalEvent
- SendSignalEvent
- ReceiveOperationEvent
- SendOperationEvent
- ChangeEvent.



### Toggle automatic creation of operations in target by typing operation names

Activating the "Toggle automatic creation of operations in target by typing operation names" icon, automatically creates the corresponding operation in the referenced class, when creating a transition and entering a name `myOperation()`.

**Note:** Operations can only be created automatically when the state machine is inside a class or interface.

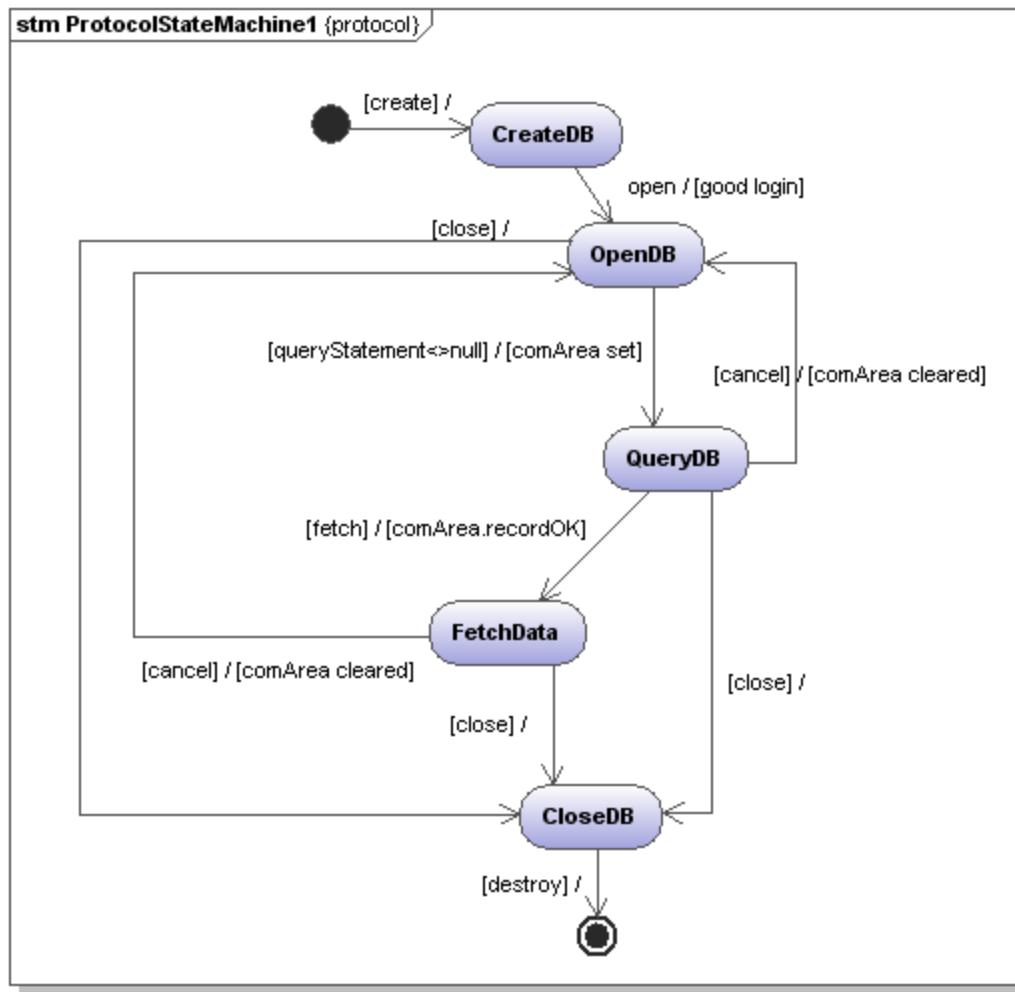
## 8.1.3 Protocol State Machine

Altova website:  [UML Protocol State Machine diagrams](#)

Protocol State Machines are used to show a **sequence** of events that an object responds to, without having to show the specific behavior. The required sequence of events, and the resulting changes in the state of the object, are modeled in this diagram.

Protocol State Machines are most often used to describe complex protocols, e.g. database access through a specific interface, or communication protocols such as TCP/IP.

Protocol State Machines are created in the same way as State Machine diagrams, but have fewer modeling elements. Protocol-Transitions between states can have pre- or post conditions which define what must be true for a transition to another state to occur, or what the resulting state must be, once the transition has taken place.



### 8.1.3.1 Inserting Protocol State Machine elements



Using the toolbar icons:

1. Click the Protocol State Machine icon in the toolbar.
2. Click in the Protocol State Machine Diagram to insert the element. To insert multiple elements of the selected type, hold down the **Ctrl** key and click in the diagram window.

## Dragging existing elements into the Protocol State Machine diagram

Most elements occurring in other Protocol State Machine diagrams, can be inserted into an existing diagram.

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

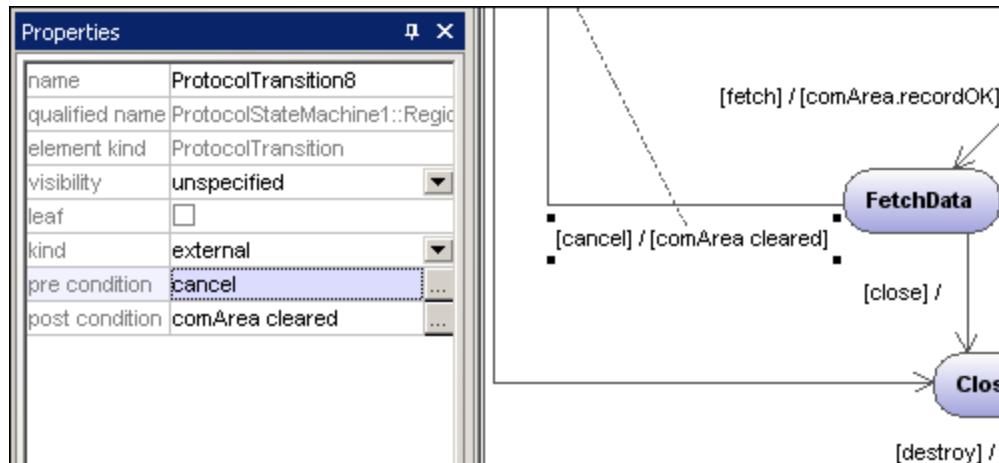
### To insert a simple state:

1. Click the **State** icon  in the icon bar and click in the Protocol State Machine diagram to insert it.
2. Enter the name of the state and press **Enter** to confirm. Simple states do not have any regions or any other type of substructure.

### To create a Protocol Transition between two states:

1. Click the Transition handle of the source state (on the right of the element), or use the Protocol Transition icon in the icon bar.
2. Drag-and-drop the transition arrow onto the target state. The text cursor is automatically set for you to enter the pre and/or post condition. Please make sure to use the square brackets [] and slash character when entering the conditions directly.

Entering the pre/post conditions in the Properties window automatically inserts the square brackets and slash character into the diagram.



For information about how to create and insert composite state elements and submachine states, see [Composite states](#) (316)

### 8.1.3.2 Protocol State Machine Diagram elements



#### State

A simple state element with one compartment.



#### Composite state

This type of state contains a second compartment comprised of a single region. Any number of states may be placed within this region.



#### Orthogonal state

This type of state contains a second compartment comprised of two or more regions, where the separate regions indicate concurrency. Right clicking a state and selecting **New | Region** allows you add new regions.



#### Submachine state

This state is used to hide details of a state machine. This state does not have any regions but is associated to a separate state machine.



#### InitialState (pseudostate)

The beginning of the process.



#### FinalState

The end of the sequence of processes.



#### EntryPoint (pseudostate)

The entry point of a state machine or composite state.



#### ExitPoint (pseudostate)

The exit point of a state machine or composite state.



#### Choice

This represents a dynamic conditional branch, where mutually exclusive guard triggers are evaluated (OR operation).



### Junction (pseudostate)

This represents an end to the OR operation defined by the Choice element.



### Terminate (pseudostate)

The halting of the execution of the state machine.



### Fork (pseudostate)

Inserts a vertical Fork bar. Used to divide sequences into concurrent subsequences.



### Fork horizontal (pseudostate)

Inserts a horizontal Fork bar. Used to divide sequences into concurrent subsequences.



### Join (pseudostate)

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



### Join horizontal (pseudostate)

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



### ConnectionPointReference

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

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

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



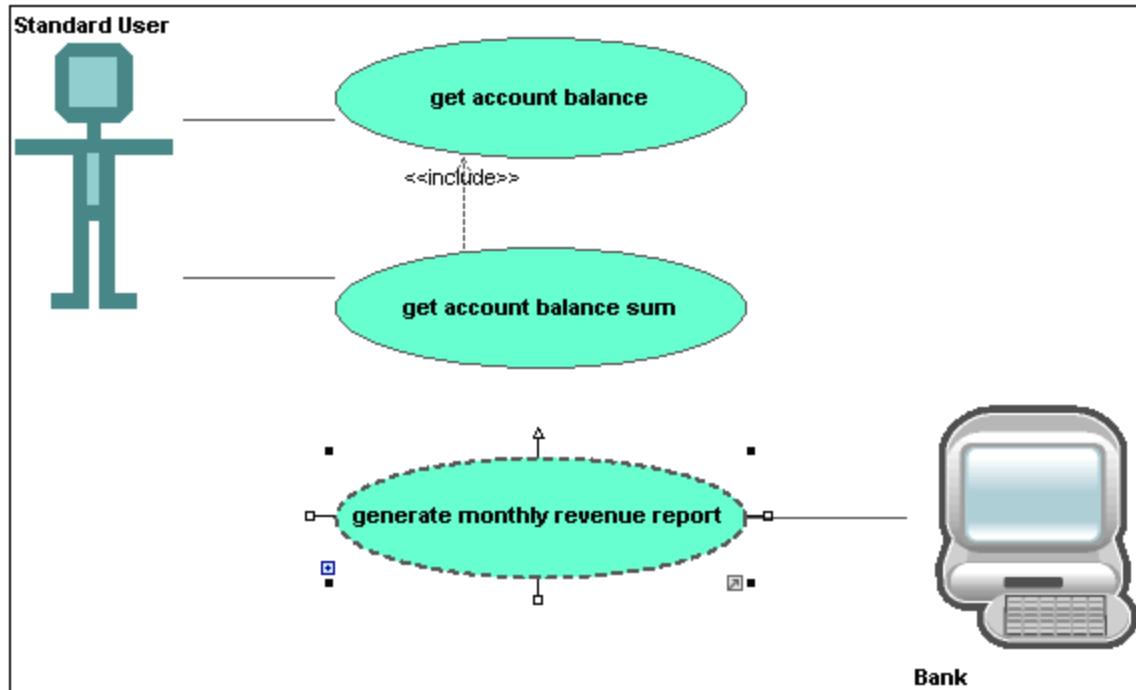
### Protocol Transition

A direct relationship between two states. An object in the first state performs one or more operations and then enters the second state depending on an event and the fulfillment of any pre- or post conditions.

Please see [Inserting Protocol State Machine elements](#)<sup>332</sup> for more information.

### 8.1.4 Use Case Diagram

Please see the [Use Cases](#) 18 section in the tutorial for more information on how to add use case elements to the diagram.



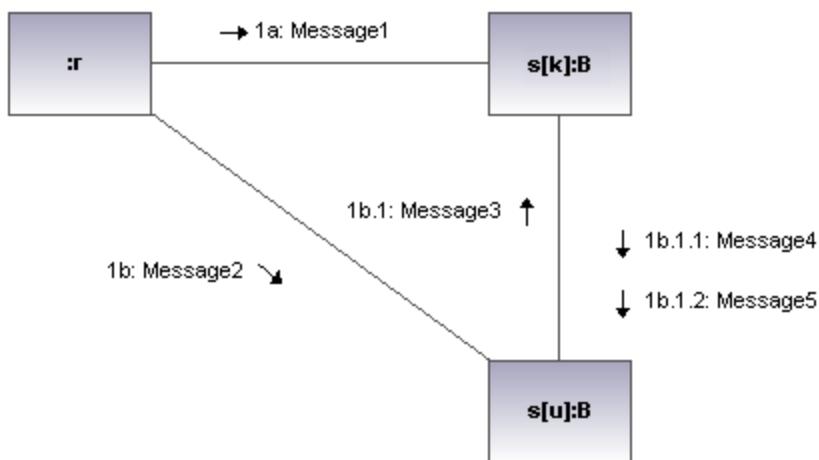
### 8.1.5 Communication Diagram

Altova website:  [UML Communication diagrams](#)

Communication diagrams display the interactions i.e. message flows, between objects at run-time, and show the relationships between the interacting objects. Basically, they model the dynamic behavior of use cases.

Communication diagrams are designed in the same way as sequence diagrams, except that the notation is laid out in a different format. Message numbering is used to indicate message sequence and nesting.

UModel allows you to generate Communication diagrams from Sequence diagrams and vice versa, in one simple action see "[Generating Sequence diagrams](#)" 338 for more information.



### 8.1.5.1 Inserting Communication Diagram elements

**Using the toolbar icons:**

1. Click the specific communication icon in the Communication Diagram toolbar.



2. Click in the Communication diagram to insert the element. To insert multiple elements of the selected type, hold down the **Ctrl** key and click in the diagram window.

### Dragging existing elements into the Communication Diagram

Elements occurring in other diagrams, e.g. classes, can be inserted into a Communication diagram.

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



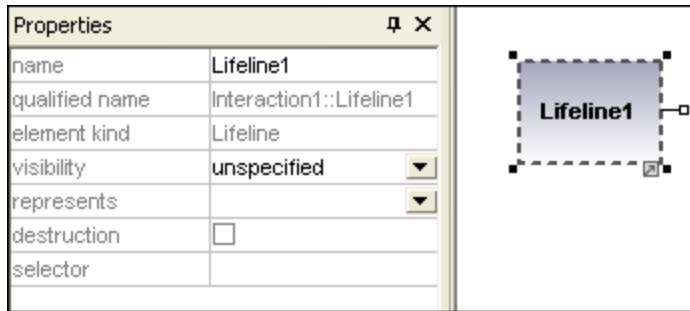
#### Lifeline

The lifeline element is an individual participant in an interaction. UModel allows you to insert other elements into the sequence diagram, e.g. classes. Each of these elements then appear as a new lifeline. You can redefine the lifeline colors/gradient using the "Header Gradient" combo boxes in the Styles tab.

To create a **multiline** lifeline, press **Ctrl+Enter** to create a new line.

### To insert a Communication lifeline:

1. Click the Lifeline icon in the title bar, then click in the Communication diagram to insert it.



2. Enter the lifeline name to change it from the default name, Lifeline1, if necessary.

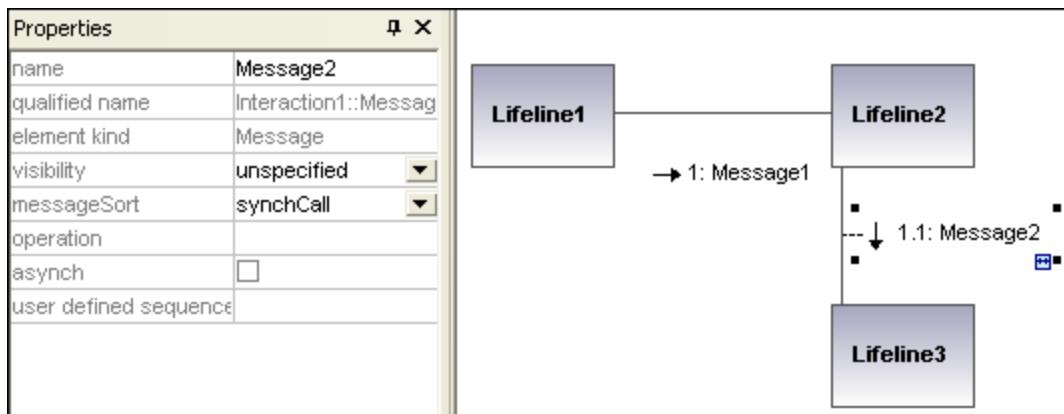
## Messages

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



### To insert a message:

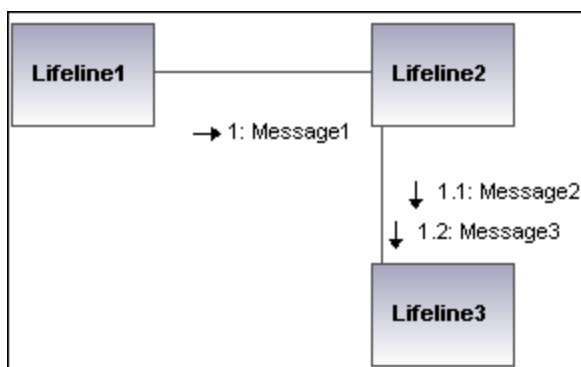
1. Click the specific message icon in the toolbar.
  2. Drag and drop the message line onto the receiver objects.
- Lifelines are highlighted when the message can be dropped.



**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.

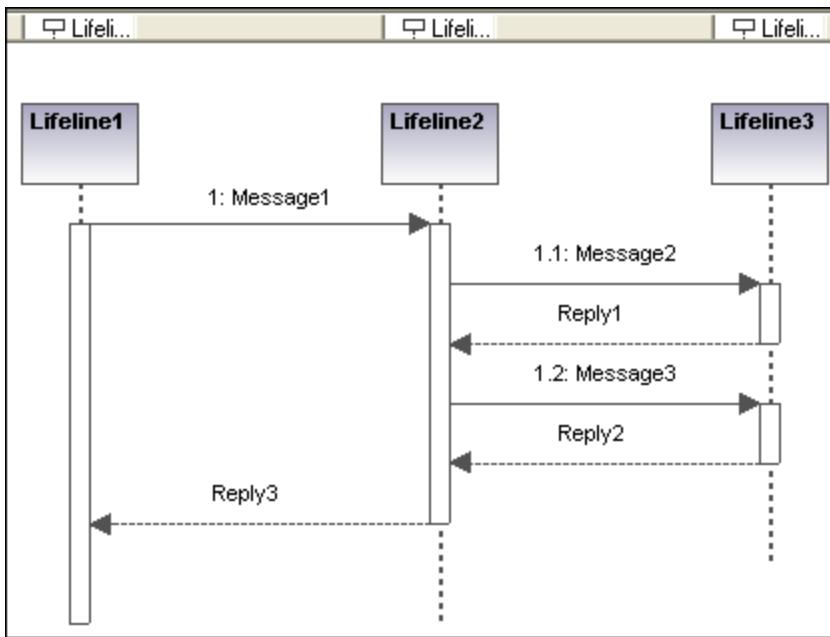
### Message numbering

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:

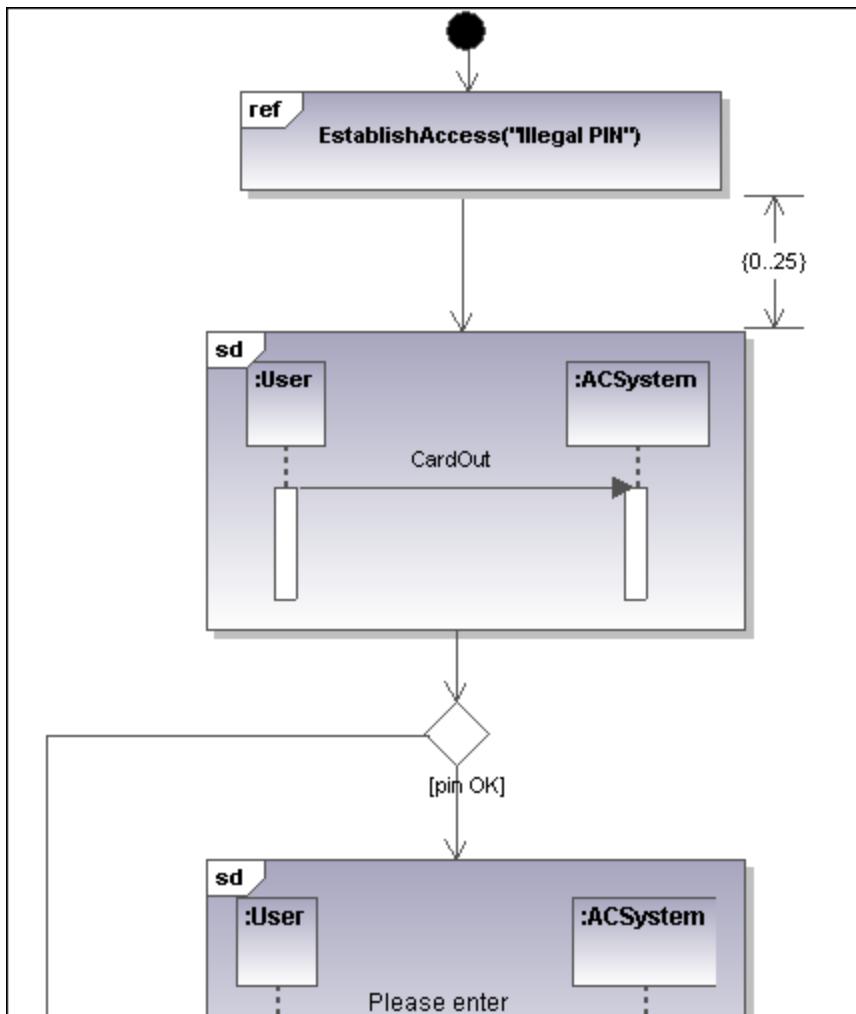
- Right-click anywhere in a Communication diagram and select **Generate Sequence Diagram** from the context menu.



### 8.1.6 Interaction Overview Diagram

Altova website: [UML Interaction Overview diagrams](#)

Interaction Overview Diagrams are a variant of Activity diagrams and give an overview of the interaction between other interaction diagrams such as Sequence, Activity, Communication, or Timing diagrams. The method of constructing a diagram is similar to that of Activity diagram and uses the same modeling elements: start/end points, forks, joins etc.



Two types of interaction elements are used instead of activity elements: Interaction elements and Interaction use elements.

Interaction elements are displayed as iconized versions of a Sequence, Communication, Timing, or Interaction Overview diagram, enclosed in a frame with the "SD" keyword displayed in the top-left frame title space.

Interaction occurrence elements are references to existing Interaction diagrams with "Ref" enclosed in the frame's title space, and the occurrence's name in the frame.

### 8.1.6.1 Inserting Interaction Overview elements

#### Using the toolbar icons

1. Click the specific icon in the Interaction Overview Diagram toolbar.



- Click in the diagram to insert the element. To insert multiple elements of the selected type, hold down the **Ctrl** key and click in the diagram window.

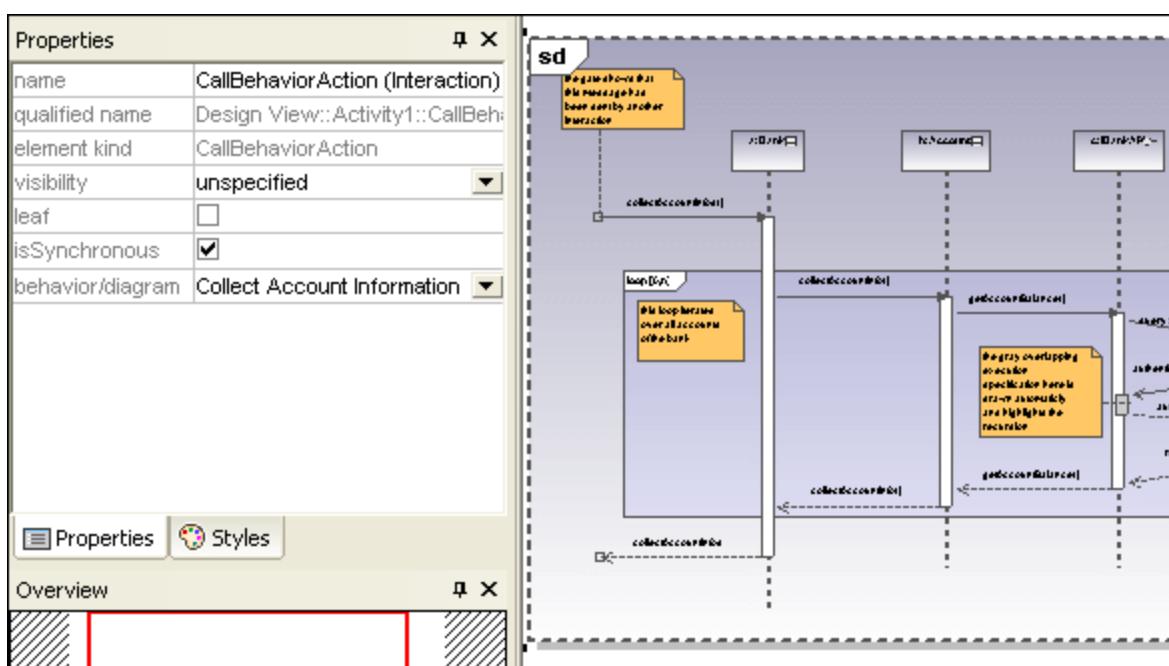
## Dragging existing elements into the Interaction Overview Diagram

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

- 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).
- Drag the element(s) into the diagram.

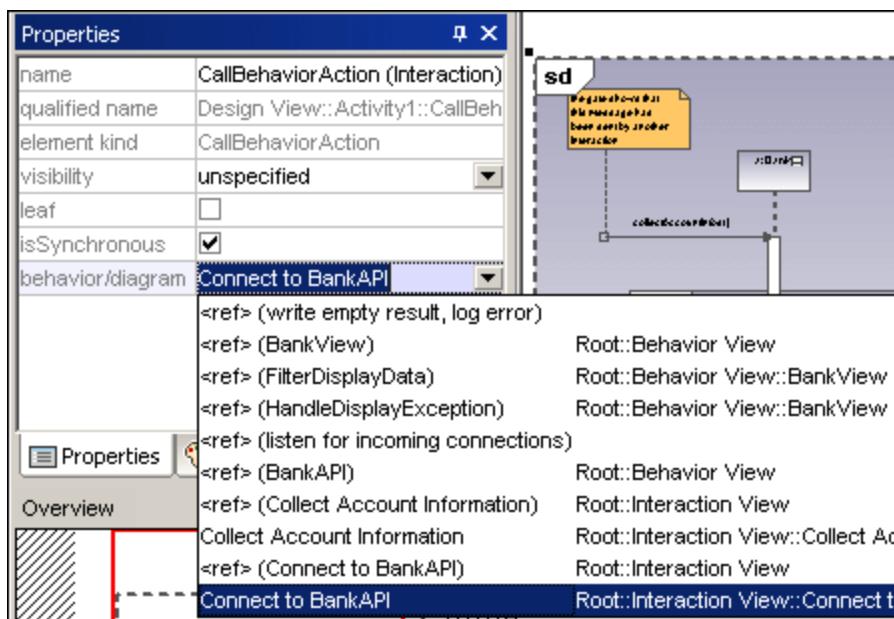
## Inserting an Interaction element

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

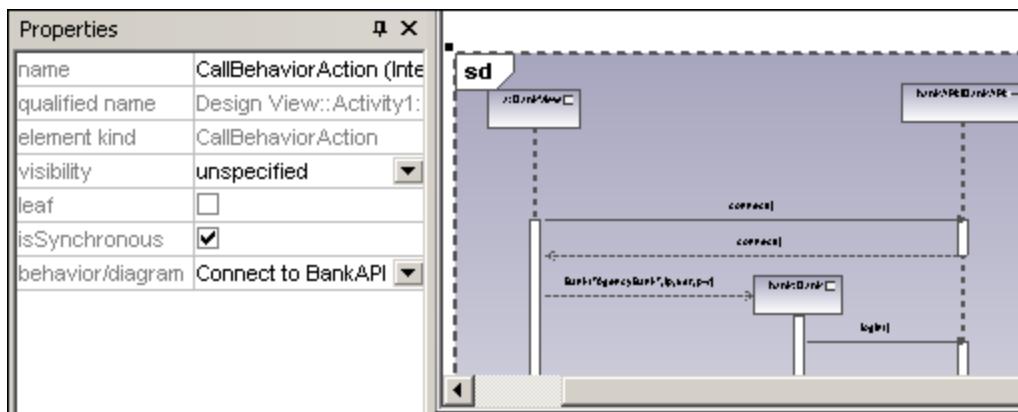


The Collect Account Information sequence diagram is automatically inserted if you are using the **Bank\_Multilanguage.ump** example file from the ...\\UModelExamples folder. The first sequence diagram, found in the model tree, is selected by default.

- 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.

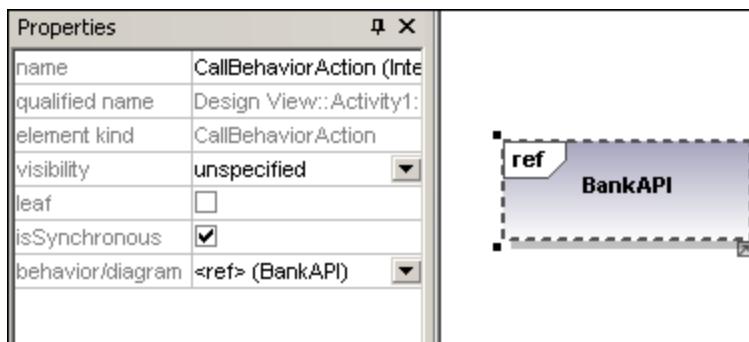


- Click the element you want to insert to e.g. Connect to BankAPI.



As this is also a sequence diagram, the Interaction element appears as an iconized version of the sequence diagram.

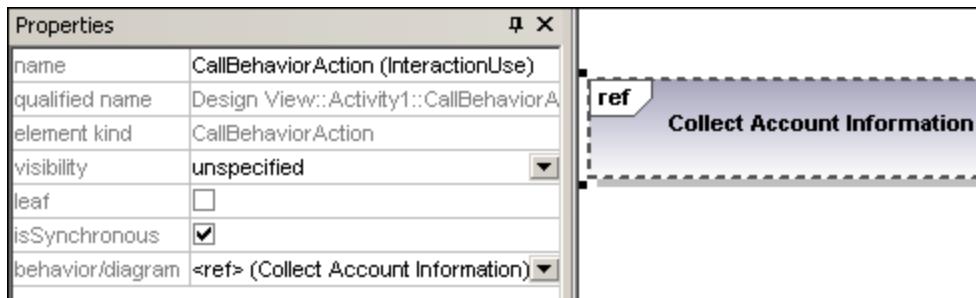
If you select <ref> BankAPI, then the Interaction element occurrence is displayed.



## Inserting an Interaction element occurrence

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

Collect Account Information is automatically inserted as a Interaction occurrence element, if you are using the **Bank\_Multilanguage.ump** example file from the ...\\UModelExamples folder. The first existing sequence diagram is selected per default.



2. To change the Interaction element, double-click the **behavior** combo box in the **Properties** tab. A list of all the possible elements that can be inserted is presented.
3. Select the occurrence you want to insert.

**Note:** All elements inserted using this method appear in the form shown in the screenshot above i.e. with "ref" in the frame's title space.



### DecisionNode

Inserts a Decision Node which has a single incoming transition and multiple outgoing guarded transitions. Please see "[Creating a branch](#)"<sup>294</sup> 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 than 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

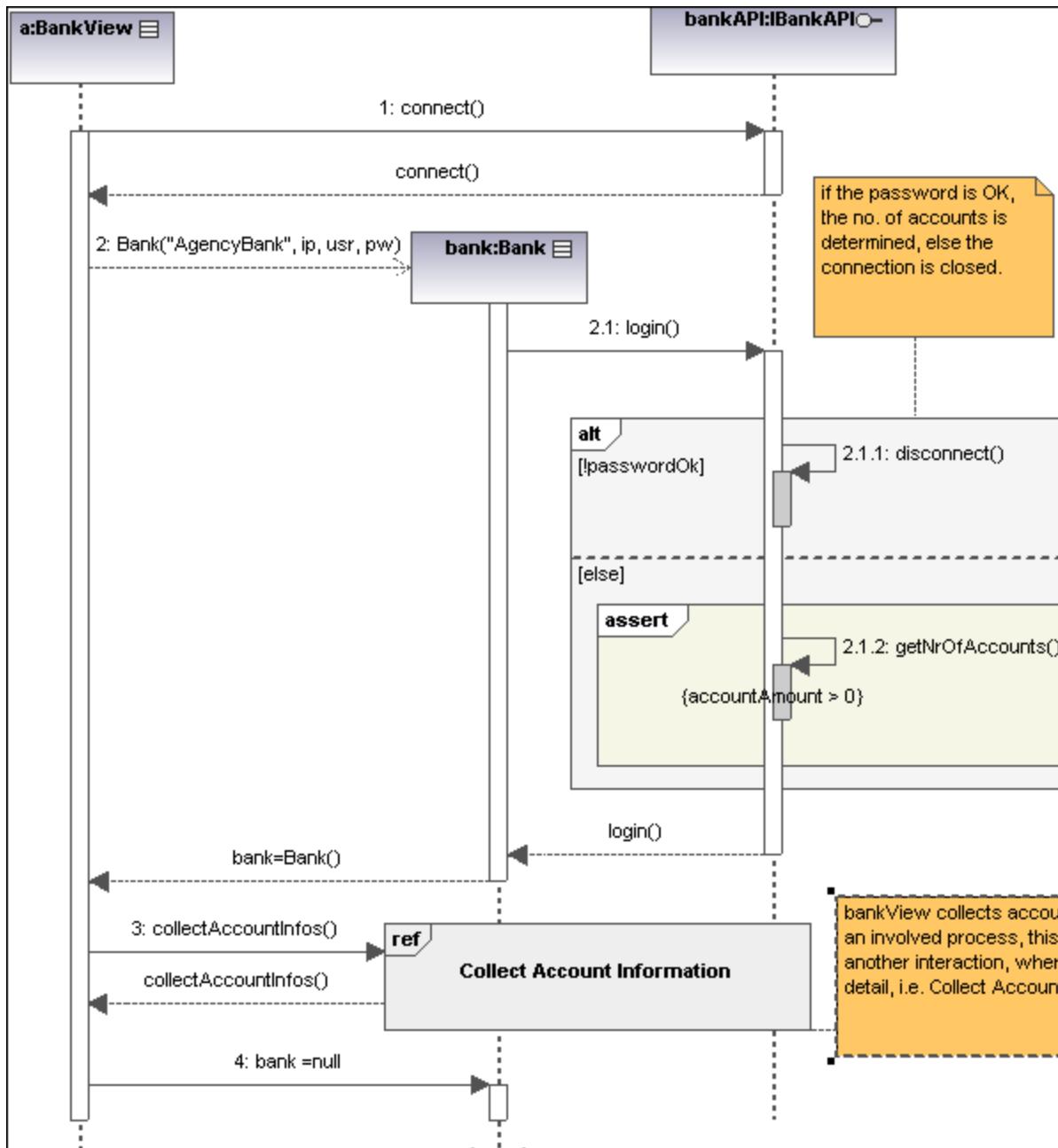
A Control Flow is an edge, i.e. an arrowed line, that connects two behaviours, and starts an interaction after the previous one has been completed.

## 8.1.7 Sequence Diagram

Altova website:  [UML Sequence diagrams](#)

UModel supports the standard Sequence diagram defined by UML, and allows easy manipulation of objects and messages to model use case scenarios. The sequence diagrams shown in the following sections are available in the **Bank\_Java.ump**, **Bank\_CSharp.ump** and **Bank\_MultiLanguage.ump** samples, in the ... \UModelExamples folder supplied with UModel.

You can model sequence diagrams manually, or, alternatively, generate them from reverse-engineered source code, as described in [Generating Sequence Diagrams from Source Code](#) 359.



### 8.1.7.1 Inserting Sequence Diagram Elements

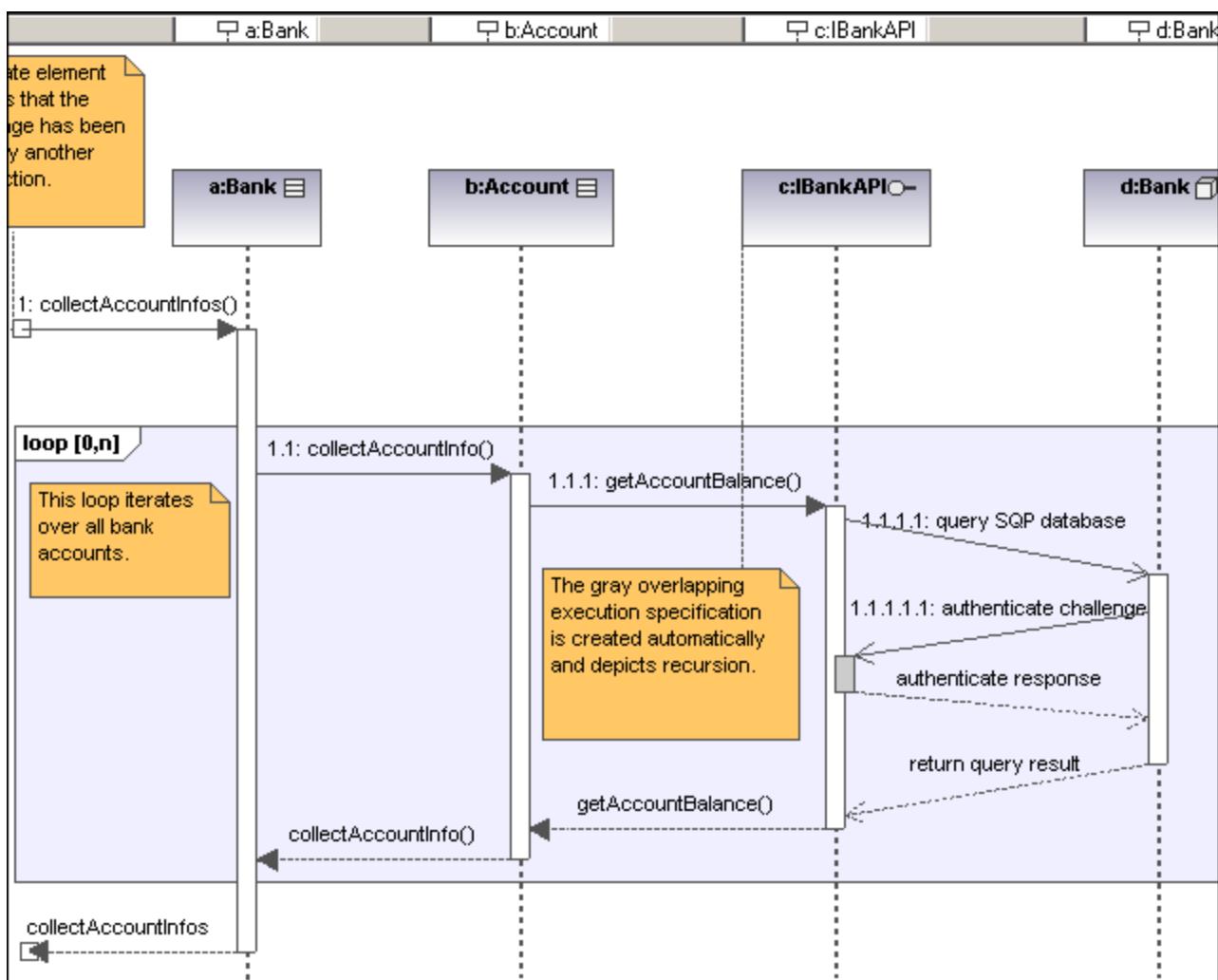
A sequence diagram models runtime dynamic object interactions, using messages. Sequence diagrams are generally used to explain individual use case scenarios.

- **Lifelines** are the horizontally aligned boxes at the top of the diagram, together with a dashed vertical line representing the object's life during the interaction. Messages are shown as arrows between the lifelines of two or more objects.
- **Messages** are sent between sender and receiver objects, and are shown as labeled arrows. Messages can have a sequence number and various other optional attributes: argument list etc. Conditional, optional, and alternative messages are all supported.

See also:

- [Lifeline](#)<sup>347</sup>
- [Combined Fragment](#)<sup>349</sup>
- [Interaction Use](#)<sup>352</sup>
- [Gate](#)<sup>352</sup>
- [State Invariant](#)<sup>353</sup>
- [Messages](#)<sup>353</sup>

Sequence diagram and other UModel elements, can be inserted into a sequence diagram using several methods.



## Using the toolbar icons

1. Click the specific sequence diagram icon in the Sequence Diagram toolbar.
2. Click in the Sequence diagram to insert the element. To insert multiple elements of the selected type, hold down the **Ctrl** key and click in the diagram window.

## Dragging existing elements into the sequence diagram

Most classifier types, as well as elements occurring in other sequence diagrams, can be inserted into an existing sequence diagram.

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

### 8.1.7.1.1 Lifeline

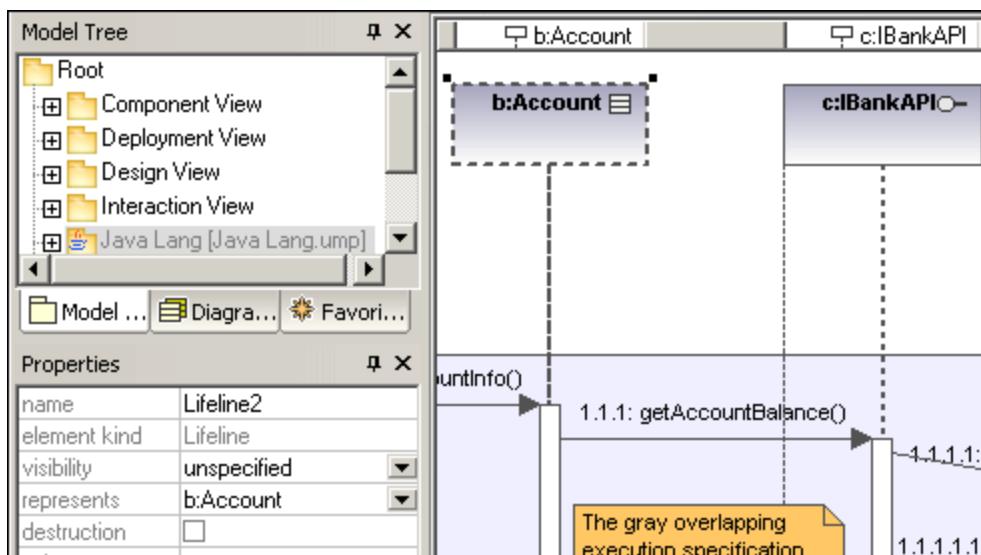


The **lifeline** element is an individual participant in an interaction. UModel also allows you to insert other elements into the sequence diagram, e.g. classes and actors. Each of these elements appear as a new lifeline once they have been dragged into the diagram pane from the **Model Tree** tab.

The "lifeline" label appears in a bar at the top of the sequence diagram. Labels can be repositioned and resized in the bar, with changes taking immediate effect in the diagram tab. You can also redefine the label colors/gradient using the "Header Gradient" combo boxes in the **Styles** tab.

To create a **multiline** lifeline, press **Ctrl+Enter** to create a new line.

Most classifier types can be inserted into the sequence diagram. The "represents" field in the Properties tab displays the element type that is acting as the lifeline. Dragging **typed** properties onto a sequence diagram also creates a lifeline.



## Execution Specification (Object activation)

An execution specification (activation) is displayed as a box (rectangle) on the object lifeline. An activation is the execution of a procedure and the time needed for any nested procedures to execute. Activation boxes are automatically created when a message is created between two lifelines.

A recursive, or self message (one that calls a different method in the same class) creates stacked activation boxes.

### To display/hide activation boxes:

- Click the **Styles** tab and scroll to the bottom of the list.

The "**Show Execution Specifications**" combo box allows you to show/hide the activation boxes in the sequence diagram.

## Lifeline attributes

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

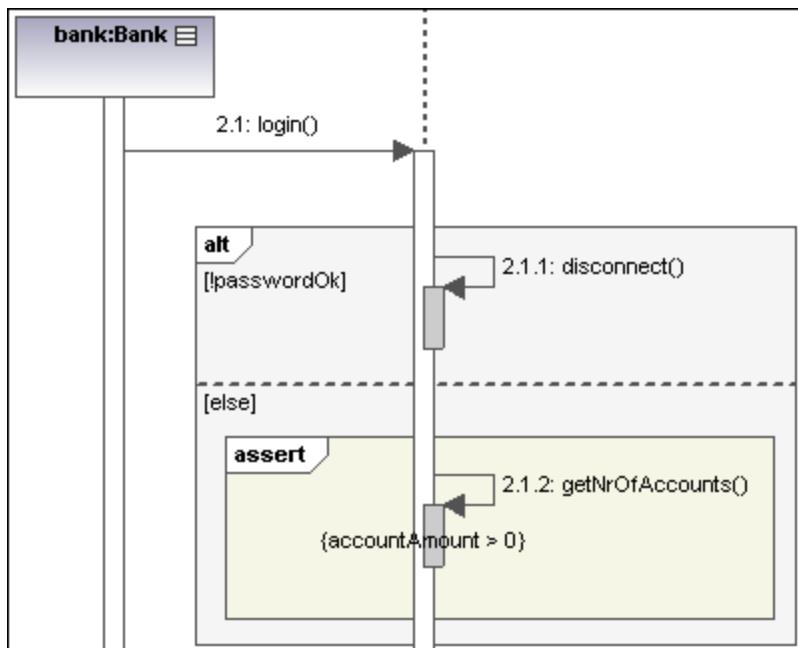
The **selector** field allows you to enter an expression that specifies the particular part represented by the lifeline, if the ConnectableElement is multivalued, i.e. has a multiplicity greater than one.

## Goto lifeline element

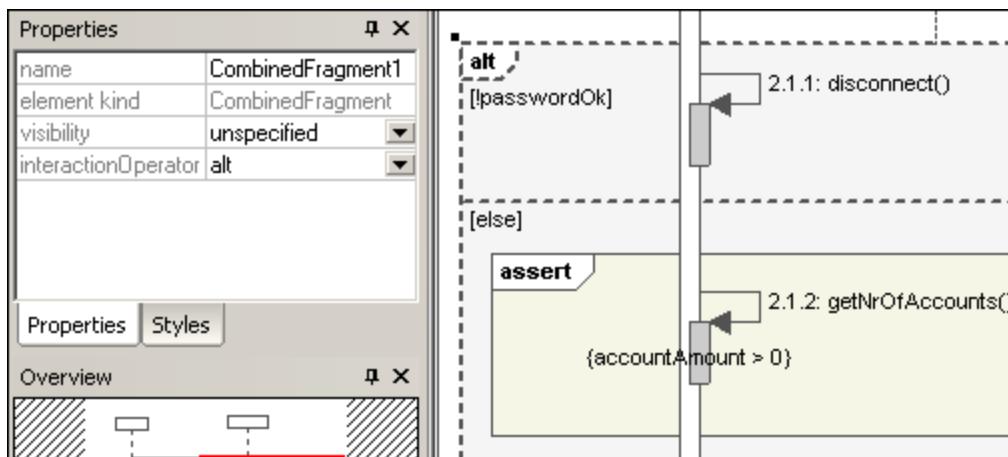
Right clicking a lifeline allows you to select Goto XXX, where XXX is the specific lifeline type that you clicked. The element will then be visible in the Model Tree window.

### 8.1.7.1.2 Combined Fragment

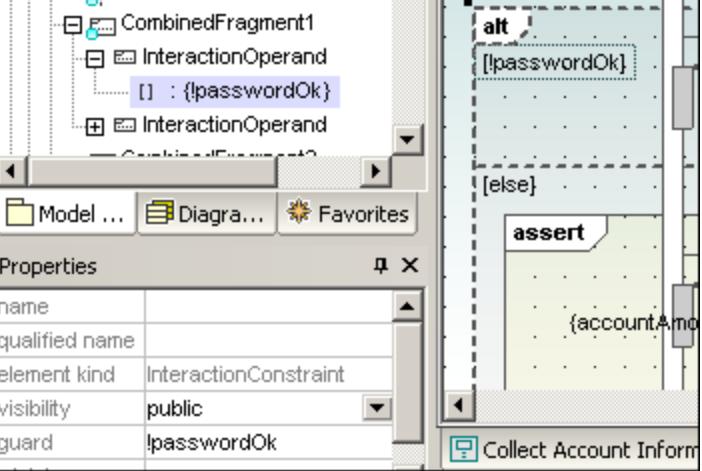
Combined fragments  are subunits, or sections of an interaction. The **interaction operator** visible in the pentagon at top left, defines the specific kind of combined fragment. The constraint thus defines the specific fragment, e.g. loop fragment, alternative fragment etc. used in the interaction.

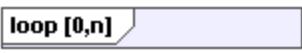


The combined fragment icons in the icon bar allow you to insert a specific combined fragment: seq, alt or loop. Clicking the **interactionOperator** combo box also allows you to define the specific interaction fragment.



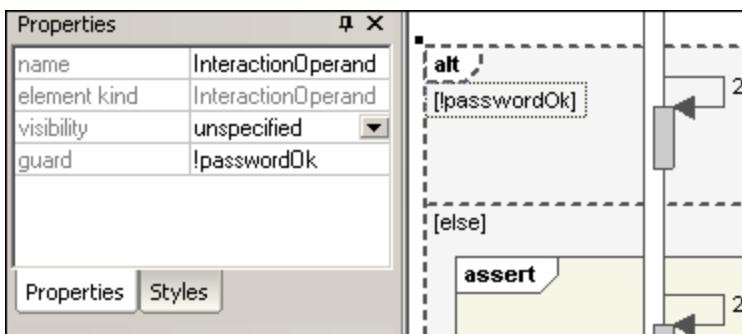
## InteractionOperators

<i>Weak sequencing</i>	<b>seq</b> 	 <p>The combined fragment represents weak sequencing between the behaviours of the operands.</p>
<i>Alternatives</i>	<b>alt</b> 	<p>Only one of the defined operands will be chosen, the operand must have a guard expression that evaluates to true.</p>  <p>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.</p>  <p>The <b>InteractionConstraint</b> is actually the guard expression between the square brackets.</p>
<i>Option</i>	<b>opt</b>	Option represents a choice where either the sole operand is executed, or nothing happens.
<i>Break</i>	<b>break</b>	The break operator is chosen when the guard is true, the rest of the enclosing fragment is ignored.
<i>Parallel</i>	<b>par</b>	Indicates that the combined fragment represents a parallel merge of operands.
<i>Strict sequencing</i>	<b>strict</b>	The combined fragment represents a strict sequencing between the behaviours of the operands.

<i>Loop</i>	<b>loop</b> 	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.
<i>Critical Region</i>	<b>critical</b>	The combined fragment represents a critical region. The sequence(s) may not be interrupted/interleaved by any other processes.
<i>Negative</i>	<b>neg</b>	Defines that the fragment is invalid, and all others are considered to be valid.
<i>Assert</i>	<b>assert</b>	Designates the valid combined fragment, and its sequences. Often used in combination with consider, or ignore operands.
<i>Ignore</i>	<b>ignore</b>	Defines which messages should be ignored in the interaction. Often used in combination with assert, or consider operands.
<i>Consider</i>	<b>consider</b>	Defines which messages should be considered in the interaction.

### Adding InteractionOperands to a combined fragment

1. Right-click the combined fragment and select **New | InteractionOperand**. The text cursor is automatically set for you to enter the guard condition.
2. Enter the guard condition for the InteractionOperand e.g. **!passwordOK** and press Enter to confirm. Use **Ctrl+Enter** to create a **multi-line** InteractionOperand.



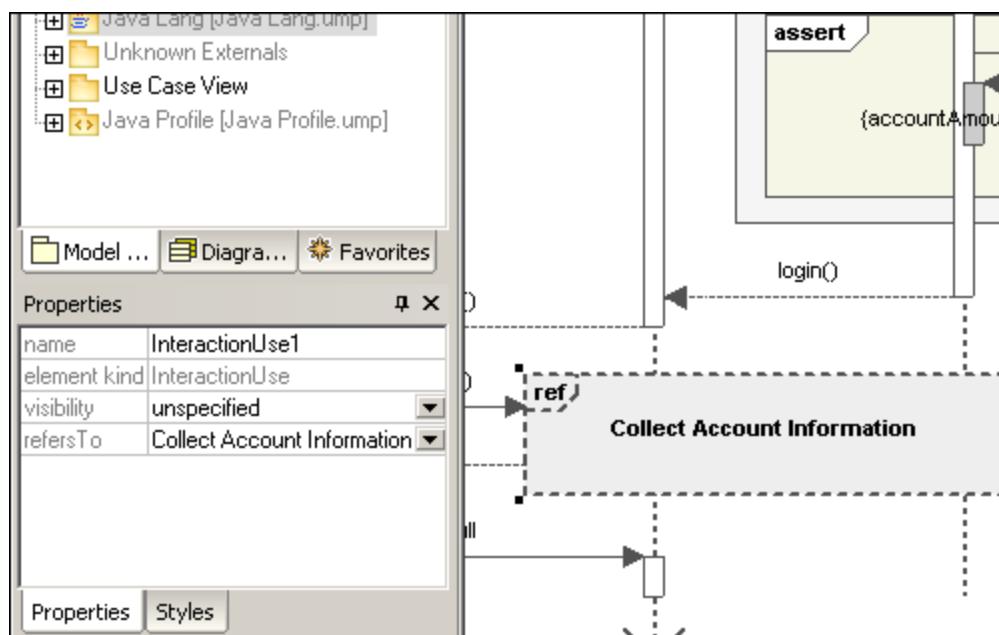
3. Use the same method to add the second interaction operand with the guard condition "else". Dashed lines separate the individual operands in the fragment.

### Deleting InteractionOperands

1. Double-click the guard expression in the combined fragment element, of the diagram (not in the **Properties** tab).
2. Delete the guard expression completely, and press Enter to confirm. The guard expression/interaction operand is removed and the combined fragment is automatically resized.

### 8.1.7.1.3 Interaction Use

The **InteractionUse**  element is a reference to an interaction element. This element allows you to share portions of an interaction between several other interactions.



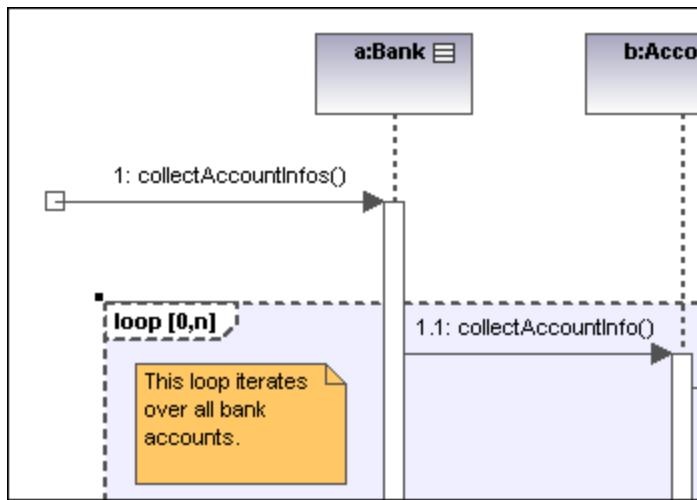
Clicking the "refersTo" combo box, allows you to select the interaction that you want to refer to. The name of the interaction use you select appears in the element.

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

### 8.1.7.1.4 Gate

A **gate**  is a connection point which allows messages to be transmitted into, and out of, interaction fragments. Gates are connected using messages.

1. Insert the gate element into the diagram.
2. Create a new message and drag from the gate to a lifeline, or drag from a lifeline and drop onto a gate. This connects the two elements. The square representing the gate is now smaller.

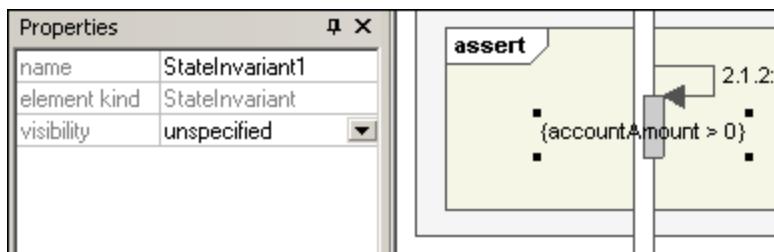


#### 8.1.7.1.5 State Invariant

A **StateInvariant**  is a condition, or constraint applied to a lifeline. The condition must be fulfilled for the lifeline to exist.

##### To define a StateInvariant:

1. Click the **State invariant** icon, then click a lifeline, or an object activation to insert it.
2. Enter the condition/constraint you want to apply, e.g. `accountAmount > 0`, and press **Enter** to confirm.



#### 8.1.7.1.6 Messages

Messages are sent between sender and receiver lifelines, and are shown as labeled arrows. Messages can have a sequence number and various other optional attributes: argument list etc. Messages are displayed from top to bottom, i.e. the vertical axis is the time component of the sequence diagram.

- A **call** is a synchronous, or asynchronous communication which invokes an operation that allows control to return to the sender object. A call arrow points to the **top** of the activation that the call initiates.

- Recursion, or calls to another operation of the same object, are shown by the stacking of activation boxes (Execution Specifications).

#### To insert a message:

1. Click the specific message icon in the Sequence Diagram toolbar.
  2. Click the lifeline, or activation box of the sender object.
  3. Drag and drop the message line onto the receiver objects lifeline or activation box. Object lifelines are highlighted when the message can be dropped.
- The direction in which you drag the arrow defines the message direction. Reply messages can point in either direction.
  - Activation box(es) are automatically created, or adjusted in size, on the sender/receiver objects. You can also manually size them by dragging the sizing handles.
  - Depending on the message numbering settings you have enabled, the numbering sequence is updated.
  - Having clicked a message icon and holding down **Ctrl** key, allows you to insert multiple messages by repeatedly clicking and dragging in the diagram tab.

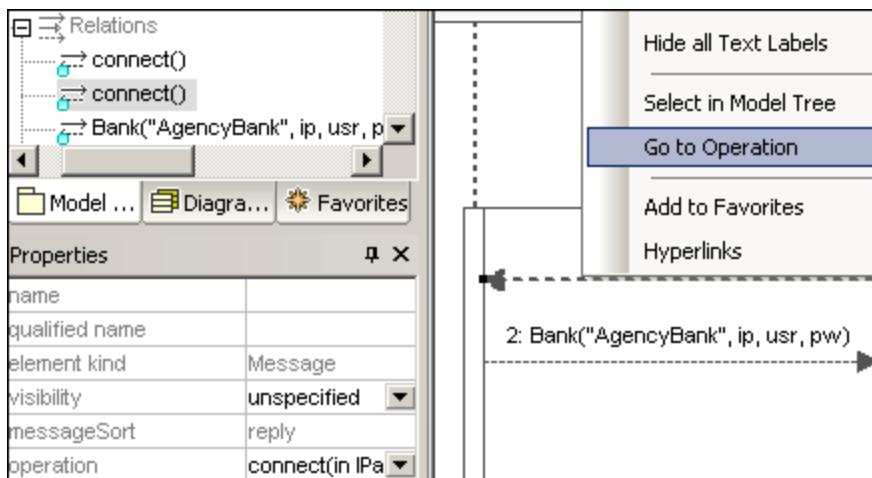
#### To delete a message:

1. Click the specific message to select it.
2. Press the **Del.** key to delete it from the model, or right click it and select "Delete from diagram". The message numbering and activation boxes of the remaining objects are updated.

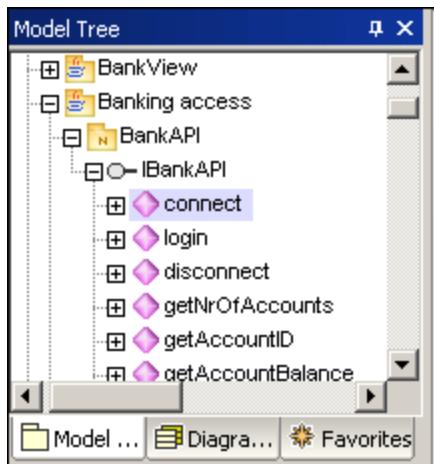
#### "Go to operation" for call messages:

The operations referenced by call messages can be found in sequence and communication diagrams.

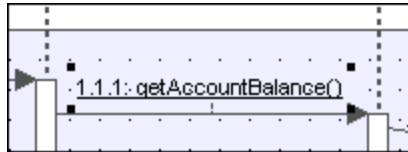
1. Right-click a call message and select "Go to Operation".



The display changes and the connect operation is displayed in the Model Tree tab.



**Note:** Static operation names are shown as underlined in sequence diagrams.



#### To position dependent messages:

- Click the respective message and drag vertically to reposition it.

The default action when repositioning messages is to move all dependent messages related to the active one. Using **Ctrl+Click** allows you to select multiple messages.

#### To position messages individually:

1. Click the **Toggle dependent message movement** icon  to deselect it.
2. Click the message you want to move and drag to move it.

Only the selected message moves during dragging. You can position the message anywhere in the vertical axis between the object lifelines.

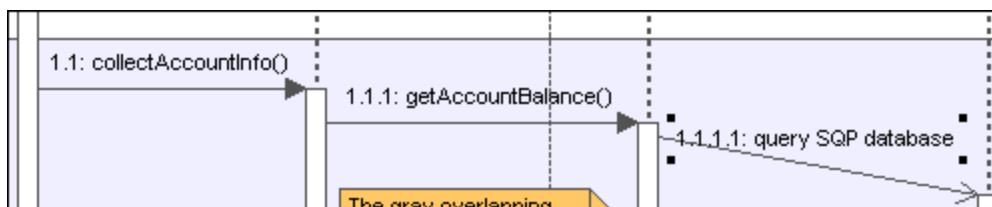
#### To automatically create reply messages:

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

#### Message numbering

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

- **None** removes all message numbering.
- **Simple** assigns a numerical sequence to all messages from top to bottom i.e. in the order that they occur on the time axis.
- **Nested** uses the decimal notation, which makes it easy to see the hierarchical structure of the messages in the diagram. The sequence is a dot-separated list of sequence numbers followed by a colon and the message name.



There are two methods of selecting the numbering scheme:

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

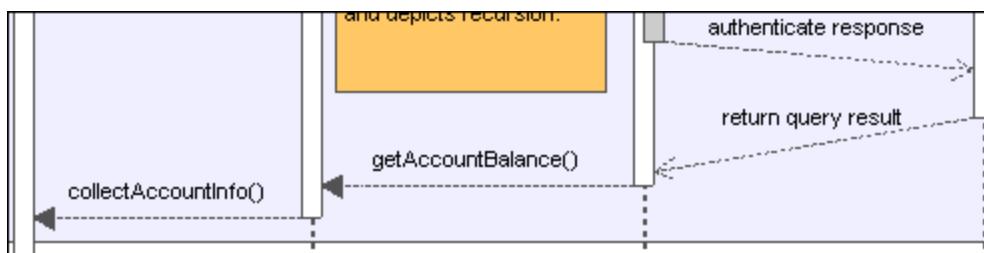
#### To select the numbering scheme using the Styles tab:

1. Click the **Styles** tab and scroll down to the **Show Message Numbering** field.
2. Click the combo box and select the numbering option you want to use. The numbering option you select is immediately displayed in the sequence diagram.

**Note:** The numbering scheme might not always correctly number all messages, if ambiguous traces exist. If this happens, adding return messages will probably clear up any inconsistencies.

#### Message replies

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



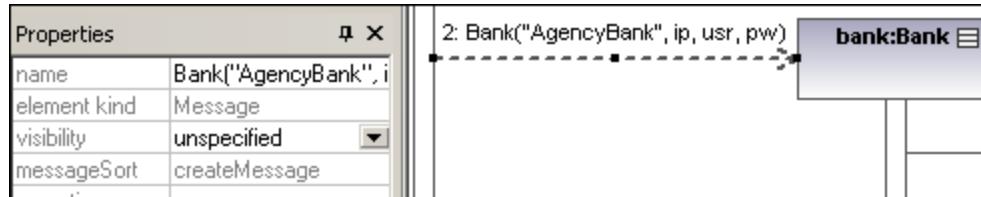
Reply messages are also generally implied by the bottom of the activation box when activation boxes are present. If activation boxes have been disabled (**Styles tab | Show Execution Specifics=false**), then reply arrows should be used for clarity.



Activating the "toggle reply messages" icon, automatically creates syntactically correct reply messages when creating a call message between lifelines/activations boxes.

## Creating objects with messages

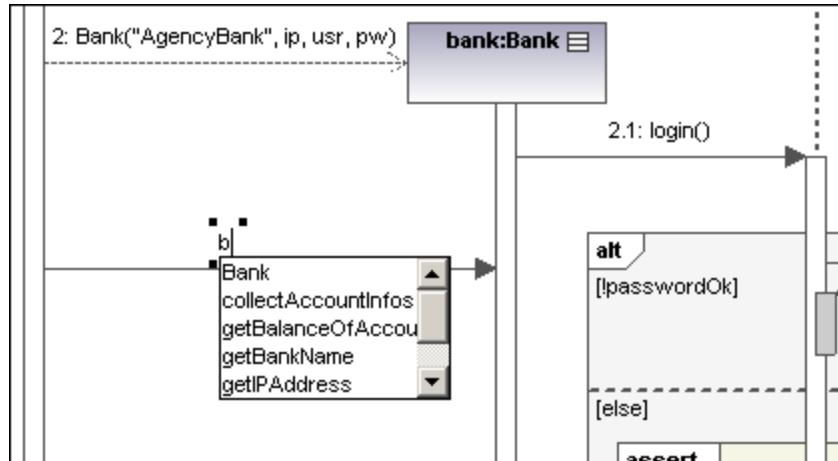
1. Messages can create new objects. This is achieved using the **Message Creation** icon .
2. Drag the message arrow to the lifeline of an existing object to create that object. This type of message ends in the middle of an object rectangle, and often repositions the object box vertically.



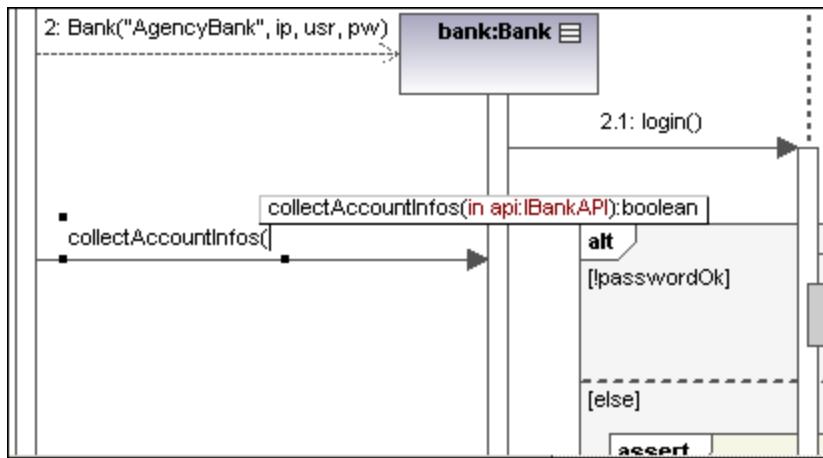
## Sending messages to specific class methods/operations in sequence diagrams

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

1. Create a message between two lifelines, the receiving object being a class lifeline (Bank). As soon as you drop the message arrow, the message name is automatically highlighted.
2. Enter a character using the keyboard e.g. "b". A pop-up window containing a list of the existing class methods is opened.



3. Select an operation from the list, and press **Enter** to confirm e.g. `collectAccountInfos`.
4. Press the space bar and press **Enter** to select the parenthesis character that is automatically supplied. A syntax helper now appears, allowing you to enter the parameter correctly.



## Creating operations in referenced classes

Activating the **Toggle automatic creation of operations in target by typing operation names** icon, automatically creates the corresponding operation in the referenced class, when creating a message and entering a name e.g. myOperation().

**Note:** Operations can only be created automatically when the lifeline references a class or interface.

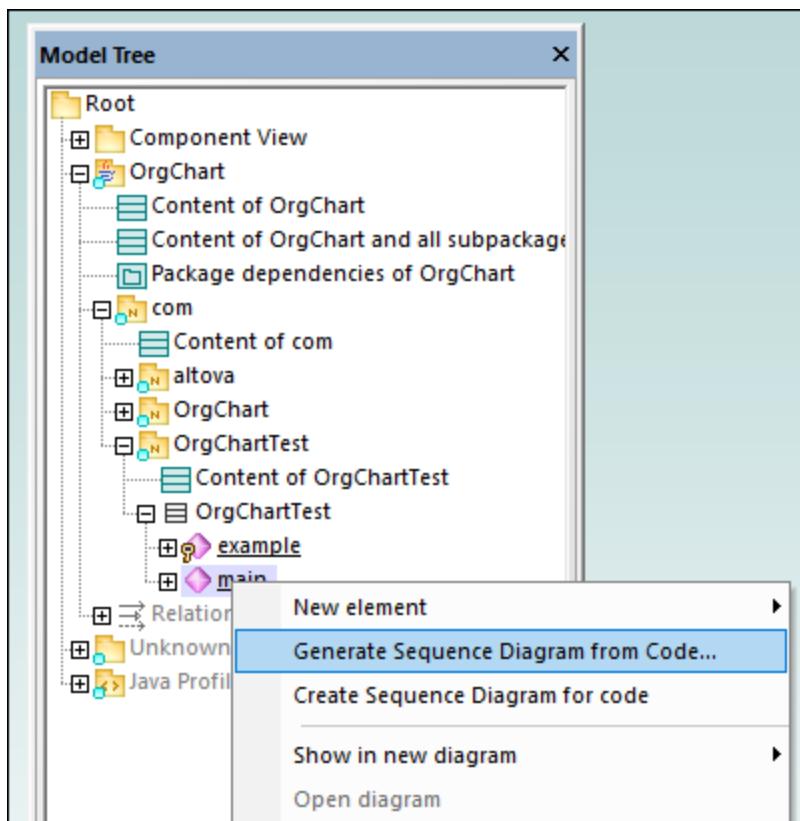
## Message icons

	Message (Call)
	Message (Reply)
	Message (Creation)
	Message (Destruction)
	Asynchronous Message (Call)
	Asynchronous Message (Reply)
	Asynchronous Message (Destruction)
	Toggle dependent message movement
	Toggle automatic creation of replies for messages
	Toggle automatic creation of operations in target by typing operation names

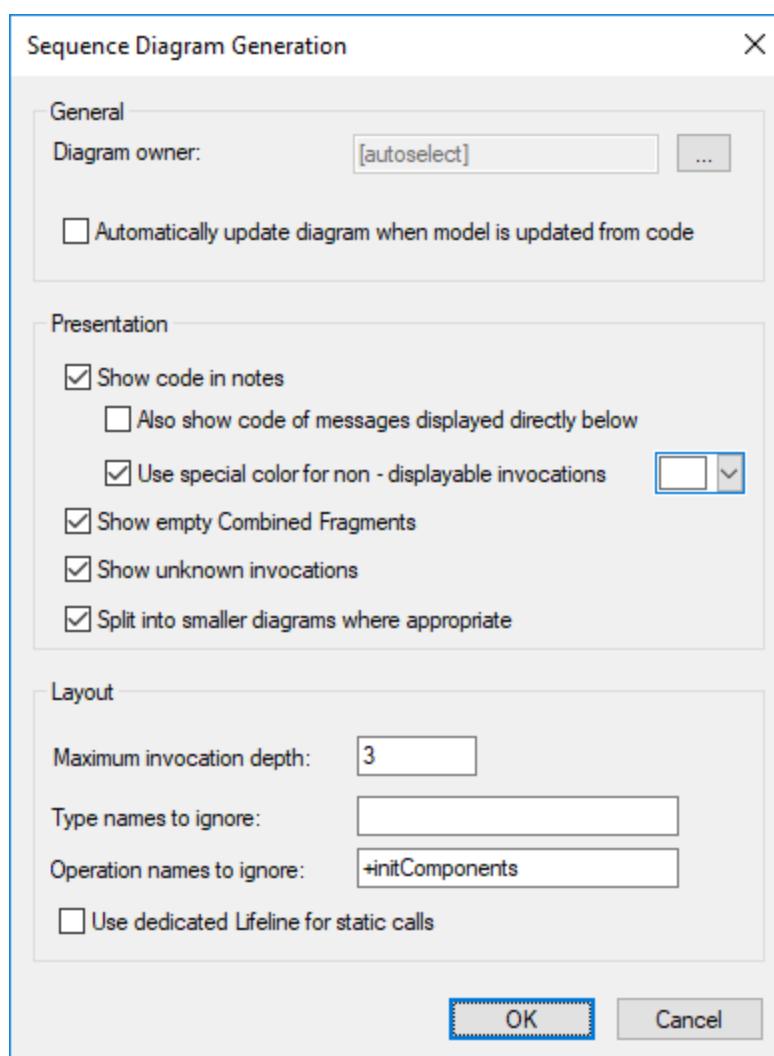
### 8.1.7.2 Generate Sequence Diagrams from Source Code

This example shows you how to generate a Sequence diagram from a method. The project containing this method will be reverse-engineered from Java source code. You can find the Java source code at the following path: **C:\Users\<user>\Documents\Altova\UModel2022\UModelExamples\OrgChart.zip**. First, unzip the **OrgChart.zip** archive to the same location (for example, right-click the archive in Windows Explorer and select **Extract All**).

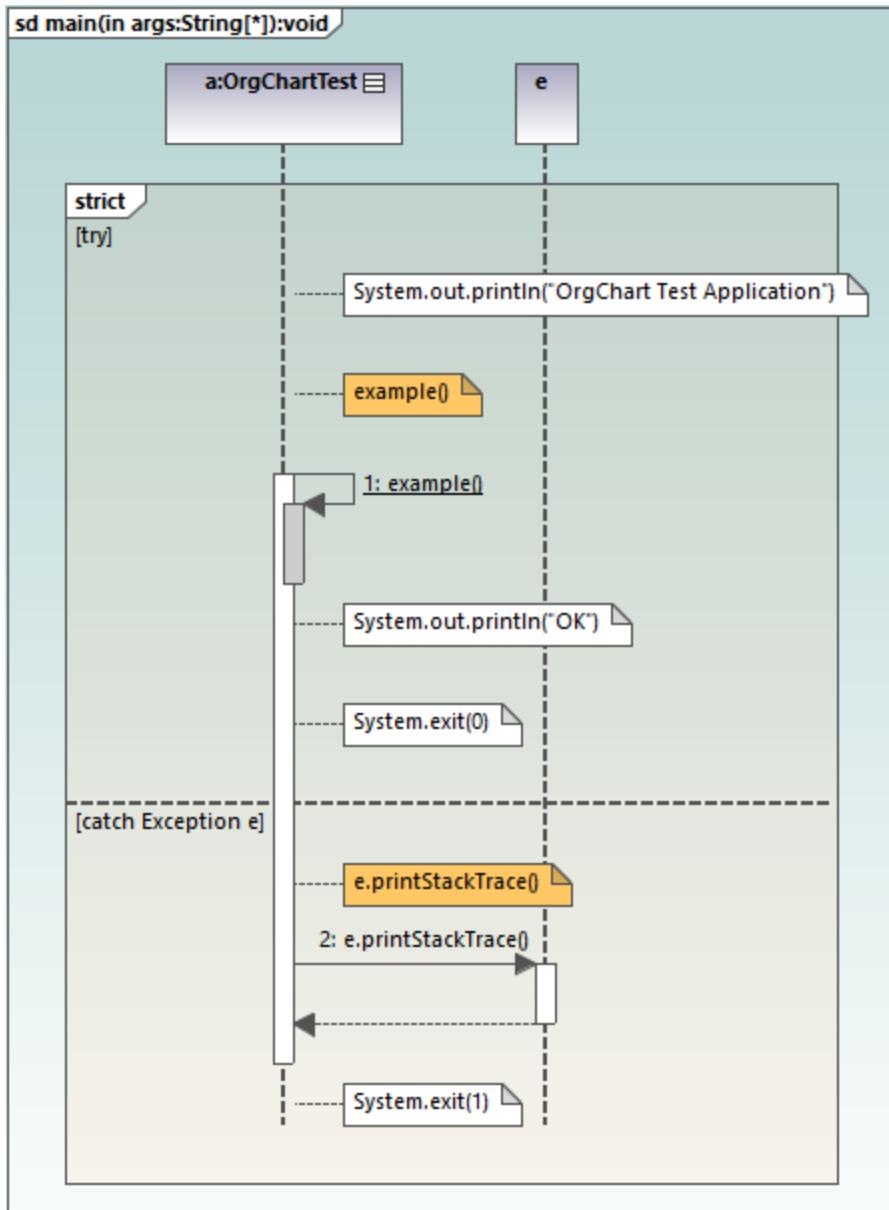
1. On the **Project** menu, click **Import Source Directory**, and select the directory unzipped previously.
2. Go through the wizard steps to import the source code as a Java project. For more information about this step, see [Reverse Engineering \(from Code to Model\)](#).
3. Having imported the code, right-click the `main` method of the `OrgChartTest` class in the Model Tree and select **Generate Sequence Diagram from Code...** from the context menu.



This opens the Sequence Diagram Generation dialog box in which you define the generation settings.



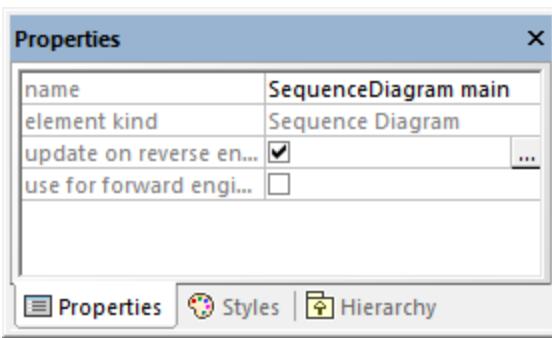
4. Select the presentation and layout options, and then click **OK** to generate the diagram. The settings shown above produce the sequence diagram below.



## Sequence diagram generation options

The table below lists the generation options pertaining to Sequence diagrams.

Option	Purpose
<i>Diagram owner</i>	You can set this option when generating a diagram for the first time. For existing diagrams, this information is read-only.

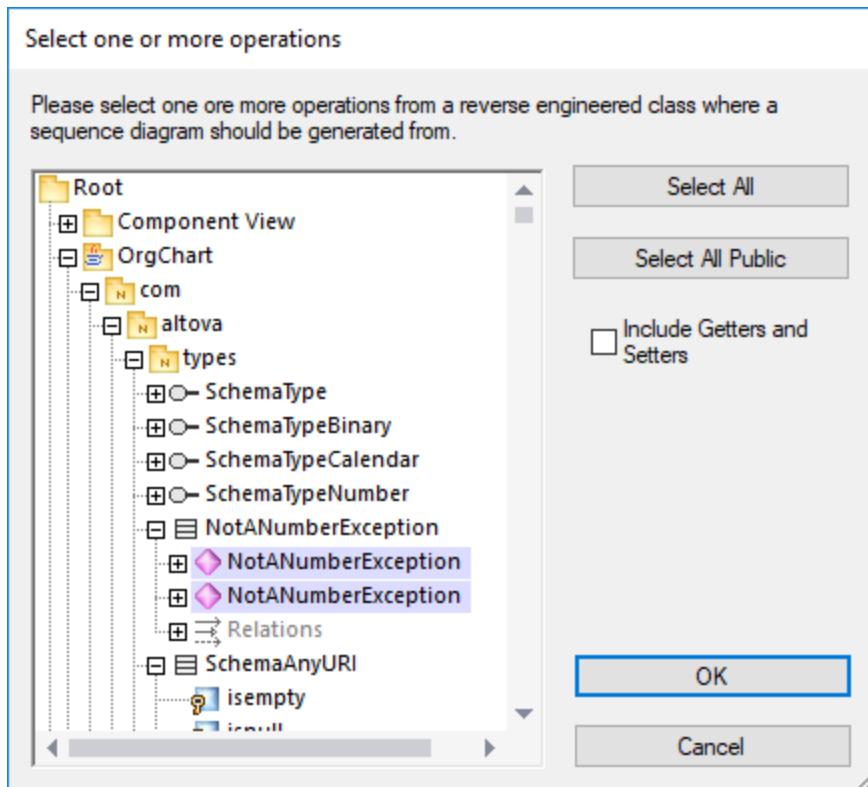
Option	Purpose								
	Click the <b>Ellipsis</b> button to select the owner package of the diagram. Otherwise, the option <b>[autoselect]</b> places the diagram in the default package.								
<i>Automatically update diagram when model is updated from code</i>	<p>When you perform reverse engineering (from code to model), sequence diagrams are re-generated automatically in the model, provided that you have selected the option <b>Automatically update diagram when model is updated from code</b> when generating the diagram for the first time.</p> <p>For existing diagrams, you can change this option as follows:</p>								
	<ol style="list-style-type: none"> <li>1. Select the Sequence diagram in the Model Tree or in the Diagram Tree.</li> <li>2. In the Properties window, select the <b>update on reverse engineering</b> check box.</li> </ol>								
	 <table border="1" data-bbox="817 973 1351 1142"> <tr> <td>name</td> <td>SequenceDiagram main</td> </tr> <tr> <td>element kind</td> <td>Sequence Diagram</td> </tr> <tr> <td>update on reverse en...</td> <td><input checked="" type="checkbox"/> ...</td> </tr> <tr> <td>use for forward engi...</td> <td><input type="checkbox"/></td> </tr> </table> <p>The Properties window has tabs for Properties, Styles, and Hierarchy.</p>	name	SequenceDiagram main	element kind	Sequence Diagram	update on reverse en...	<input checked="" type="checkbox"/> ...	use for forward engi...	<input type="checkbox"/>
name	SequenceDiagram main								
element kind	Sequence Diagram								
update on reverse en...	<input checked="" type="checkbox"/> ...								
use for forward engi...	<input type="checkbox"/>								
	<p>If you select the <b>use for forward engineering</b> check box, the synchronization from model to code will generate code based on the sequence diagram, when you perform forward engineering (from model to code), see also <a href="#">Generate Code from Sequence Diagram</a>.</p> <p>If the two "engineering" check boxes are missing, it is likely that this diagram is just a fragment of a bigger diagram, or perhaps you have created the diagram from a non reverse-engineered operation.</p>								
<i>Show code in notes</i>	Select this check box to generate the diagram with notes (callouts) that contain program code.								
<i>Also show code of messages displayed directly below</i>	Even when it is possible to show a piece of code as UML Message on the diagram, this option still displays the code of that message as a note.								

<b>Option</b>	<b>Purpose</b>
<i>Use special color for non-displayable invocations</i>	Assigns a color of your choice to non-displayable invocations.
<i>Show empty Combined Fragments</i>	Keeps the <a href="#">Combined Fragment</a> <sup>349</sup> blocks on the diagram, even if they don't contain anything.
<i>Show unknown invocations</i>	When selected, this option also displays messages for operations or constructors which could not be resolved (that is, not found in the model).
<i>Split into smaller diagrams where appropriate</i>	Automatically splits sequence diagrams into smaller sub-diagrams, and automatically generates hyperlinks between them for easy navigation.
<i>Maximum invocation depth</i>	Defines the call depth to be used in the diagram. For example, if <code>method1()</code> calls <code>method2()</code> which calls <code>method3()</code> , and the invocation depth is set to <b>2</b> , then only <code>method2</code> is shown, and <code>method3</code> is no longer shown.
<i>Type names to ignore</i>	Lets you define a comma delimited list of types that should not appear in the sequence diagram when it is generated.
<i>Operation names to ignore</i>	Lets you define a comma delimited list of operations that should not appear in the generated sequence diagram. Adding the operation names to the list causes the complete operation to be ignored. Prepending a "+" character to the operation in the list (for example, <code>+InitComponent</code> ) causes the operation calls to be shown in the diagram, but without their content.
<i>Use dedicated Lifeline for static calls</i>	If there are static methods calls, and if there is already an instance of that object on the diagram, messages are normally drawn to that existing lifeline. With this option enabled, the diagram generator uses a dedicated new lifeline just for static method calls for that classifier.

### 8.1.7.2.1 Generate Multiple Sequence Diagrams

You can also create multiple sequence diagram models from multiple operations, as follows:

1. Select the menu option **Project | Generate Sequence diagrams from Code**.



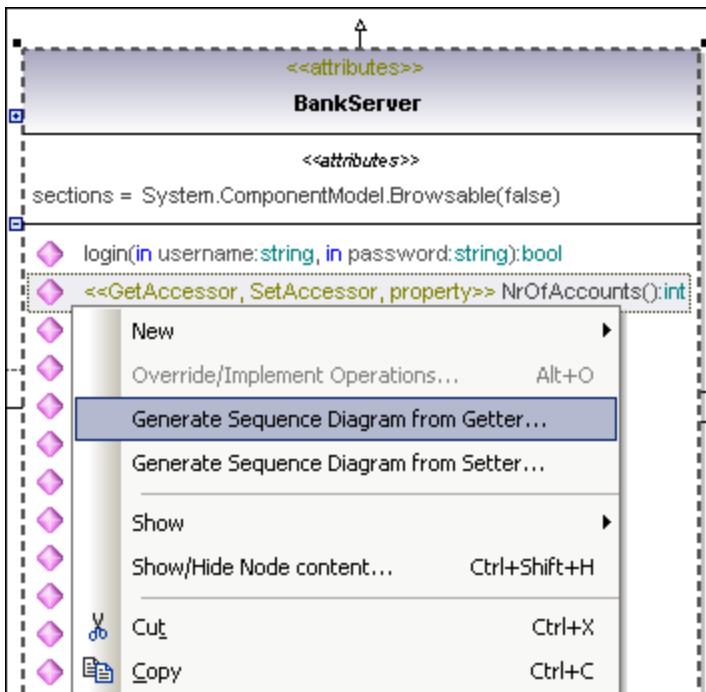
2. Select the operations that you want to generate a sequence diagram for and click **OK**. (Use the **Select All Public** and **Select All** buttons where necessary.)
3. Optionally, select the **Include Getters and Setters** check box to generate sequence diagrams for C#/VB.NET getters and setters.
4. Click **OK**. This opens a dialog box where you can specify the [sequence diagram generation options](#) (361)
5. Click **OK**. A sequence diagram is generated for each selected operation, and UModel automatically opens it.

Creating multiple Sequence diagrams will likely take longer if your project is large. Note that only the first 10 diagrams will be opened automatically by UModel; all the rest will be generated without being opened.

### 8.1.7.2.2 Generate Sequence Diagrams from Getters/Setters

You can also generate a sequence diagram from getter/setter properties (in C#, VB .NET), as follows:

1. Right-click an Operation with a `GetAccessor/SetAccessor` stereotype.



2. Select **Generate Sequence Diagram from Code (Getter/Setter)** from the context menu. This opens a dialog box where you can specify the [sequence diagram generation options](#) <sup>361</sup>.
3. Click **OK** to generate the Sequence Diagram.

### 8.1.7.3 Generate Code from Sequence Diagram

UModel can create code from a sequence diagram which is linked to at least one operation. Code generation from sequence diagrams is available for:

- VB.NET, C# and Java
- UModel standalone, Eclipse, and Visual Studio editions
- All three UModel editions

Creating code from Sequence diagrams is possible by either:

- Starting from a reverse engineered operation, see [Generating Sequence Diagrams from source code](#) <sup>359</sup>,
- By creating a **new** sequence diagram from scratch, which is linked to an operation, by right-clicking the operation (in the Model Tree) and selecting [Create sequence diagram for code](#) <sup>368</sup>.

When using a reverse engineered sequence diagram as basis, ensure that the option "Show code in notes" is selected when reverse engineering the code, so you do not lose any code when you start the forward-engineering process again. This is due to the fact that UML is not able to display all the language features of VB.NET, Java and C# on the sequence diagram, and those code sections are therefore shown as code notes.

**To add plain text as code when creating a sequence diagram:**

1. Attach a note to a sequence diagram lifeline.
2. Type in the code which should be written into the final source code. Click the **Is Code** check box (in the **Properties** pane) for that note, to make it accessible.

See [Adding code to sequence diagrams](#) 368 for an example.

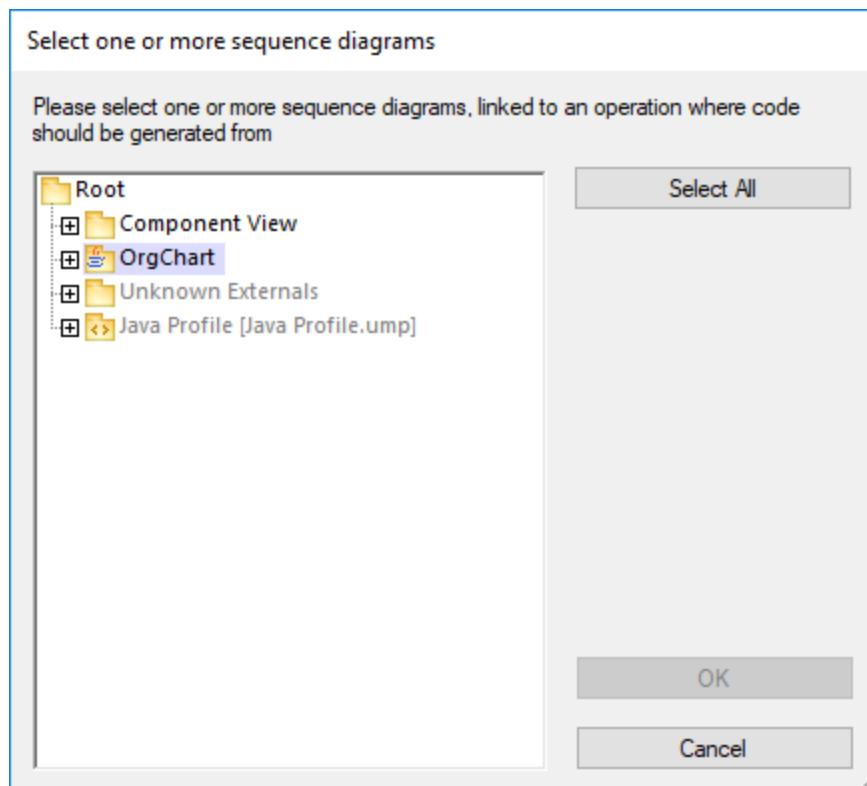
If a Sequence Diagram is to be used for code engineering automatically every time code engineering is started:

1. Select the diagram in the Model Tree or Diagram Tree window.
2. Select the **Use for forward engineering** check box in the **Properties** window.

Old code will always be lost when forward engineering code from a sequence diagram, because it will be overwritten with the new code.

**To generate code using the Project menu:**

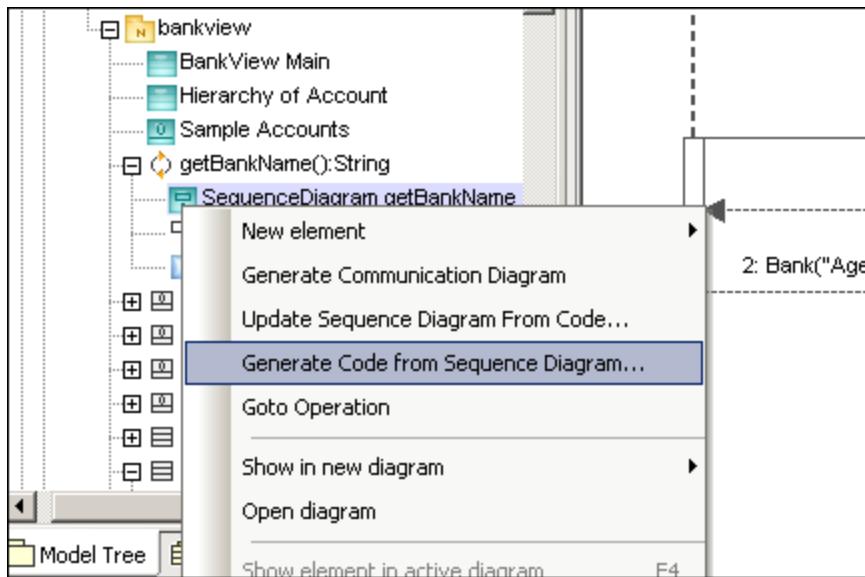
1. Select the menu option **Project | Generate Code from Sequence Diagrams**. You are now prompted to select the specific Sequence Diagram(s). Clicking the "Select All" button selects all the Sequence Diagrams in the UModel project.



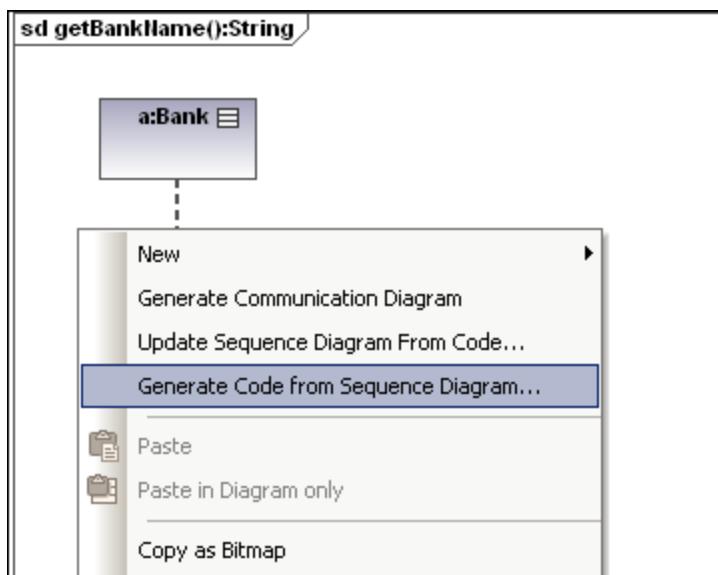
2. Click OK to generate the code. The Messages window shows the status of the code generation process.

**To generate code using the Model Tree:**

- Right click a Sequence Diagram and select **Generate Code from Sequence diagram**.

**To generate a Sequence Diagram containing code of an operation:**

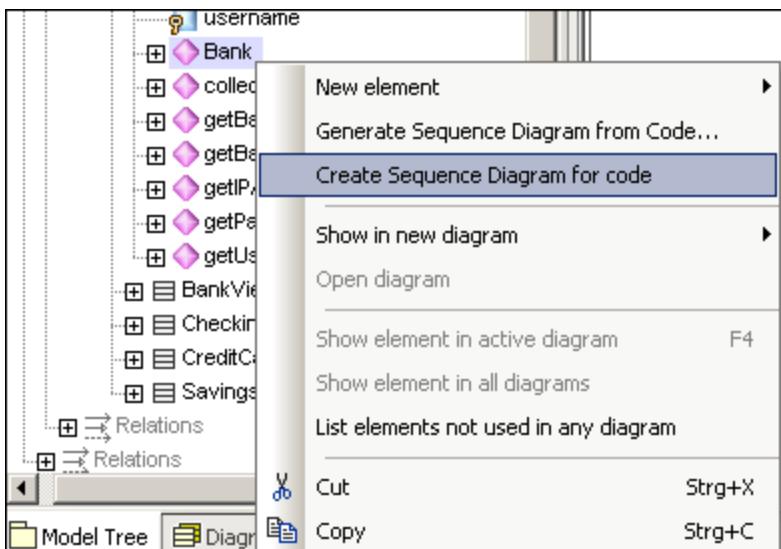
- Click into the empty space of the Sequence Diagram, that contains code of an operation.
- Select **Generate Code from Sequence diagram**.



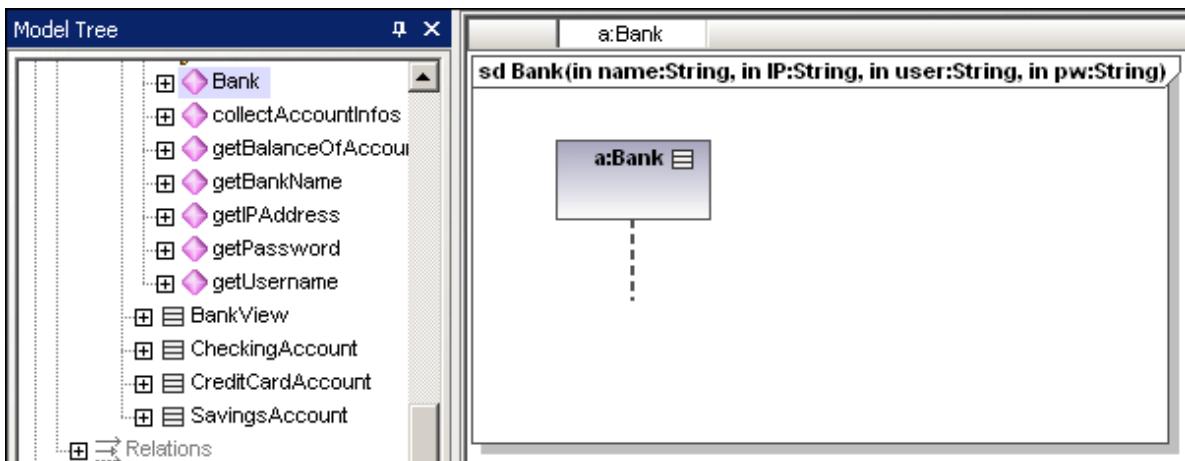
This command starts the forward-engineering process at this point.

To create a Sequence diagram for code (engineering):

- In the Model Tree, right-click an operation and select **Create Sequence diagram for code**.



You will then be prompted if you want to use the new diagram for forward engineering.



The result is a new Sequence Diagram containing the lifeline of that class.

#### 8.1.7.3.1 Adding code to sequence diagrams

Program code can be generated from new, and reverse-engineered sequence diagrams, but only for a sequence diagram linked to the "main operation".

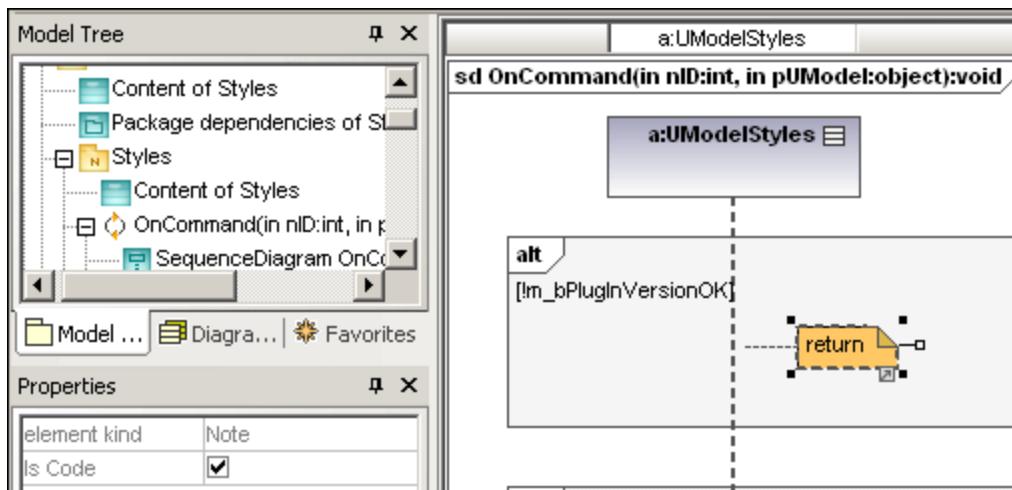
When reverse-engineering code, standard sequence diagram elements, e.g. CombinedFragments, are "mapped/assigned" to coding elements (e.g. "if" statements, loops, etc.).

For those programming statements that have no corresponding sequence diagram elements, e.g. "i = i+1", UModel makes use of "code" notes to add code to diagrams. These notes must then be linked to the lifeline.

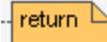
Note that UModel does not check, or parse, these code fragments. It is up to you to make sure that the code fragments are correct and will compile.

### To add code to a sequence diagram:

1. Click the **Note** icon  then click the model element where you want to insert it, e.g. CombinedFragment.
2. Enter the code fragment, e.g. return.
3. Click the Node Link handle of the inserted note and drop the cursor on the lifeline.
4. Activate the "Is Code" check box in the Properties tab to include this code fragment when generating code.

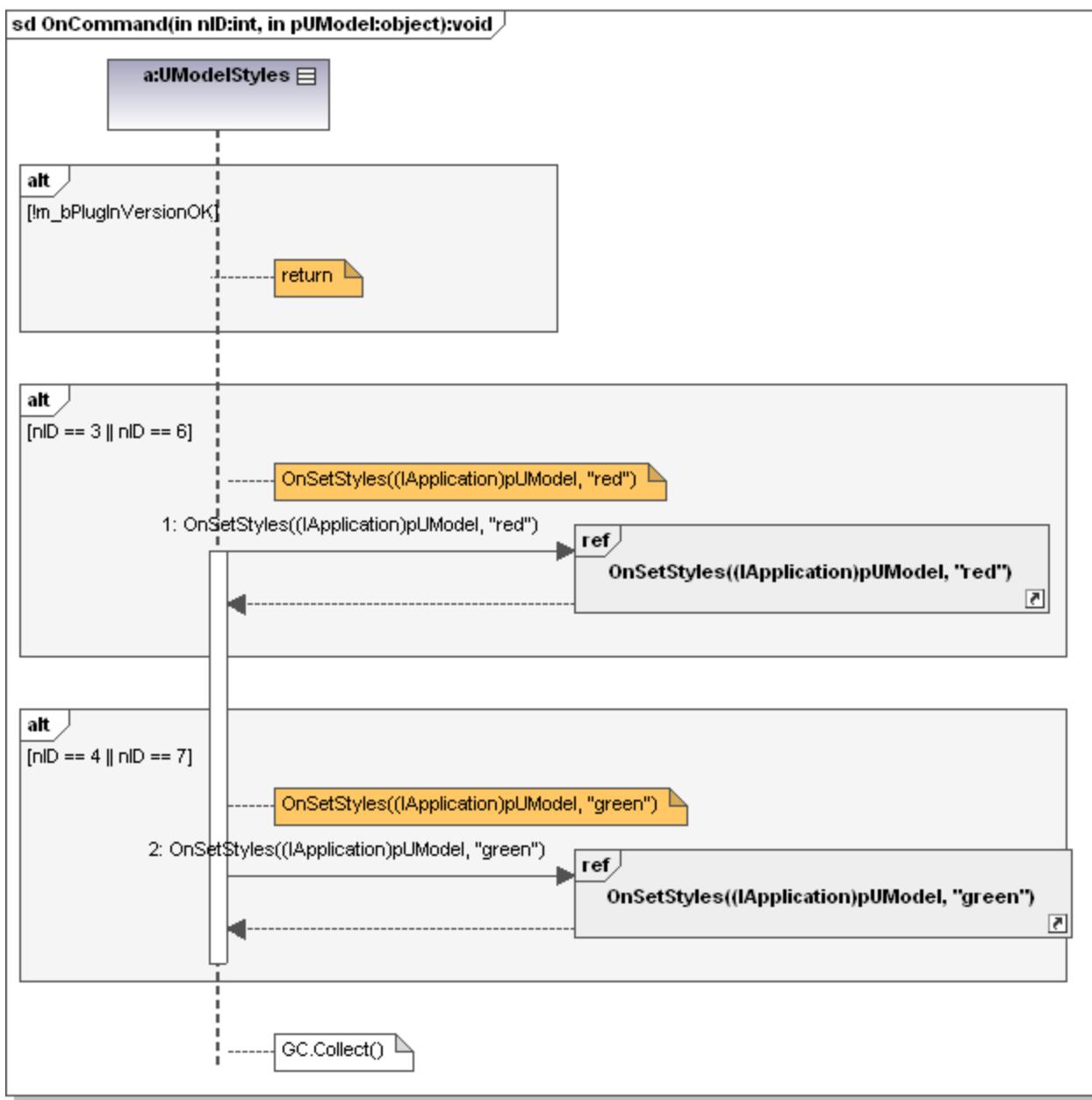


When selecting a note on a sequence diagram, which **can** be used for code generation, the property "is code" is available in the Properties window. Clicking the check box, allows you to switch between "ordinary" notes and code generation notes.

Ordinary notes:	
Code generation notes	 - shown with a darker dog-ear

Code updates occur automatically on every forward engineering process if the "Use for forward engineering" check box is active. If changes were made to the sequence diagram, the code of the operation is always overwritten.

The sequence diagram shown below was generated by right clicking the **OnCommand** operation and selecting **Generate sequence diagram from code**. The C# code of this example is available in the **C:\Users\<user>\Documents\Altova\UModel2022\UModelExamples\IDEPlugIn\Styles** folder. Use the option **Project | Import Source Project**, to import the project.



The code shown below is generated from the sequence diagram.

```

Public void OnCommand(int nID, object pUModel)
{
    //Generated by UModel. This code will be overwritten when you re-run code generation.

    if (!m_bPlugINVersionOK)
    {
        return;
    }
}

```

```

if (nID == 3 || nID == 6)
{
OnSetStyles((IAplication)pUModel, "red");

}

if (nID == 4 || nID == 7)
{
OnSetStyles((IAplication)pUModel, "green");
}
GC.Collect();

}

```

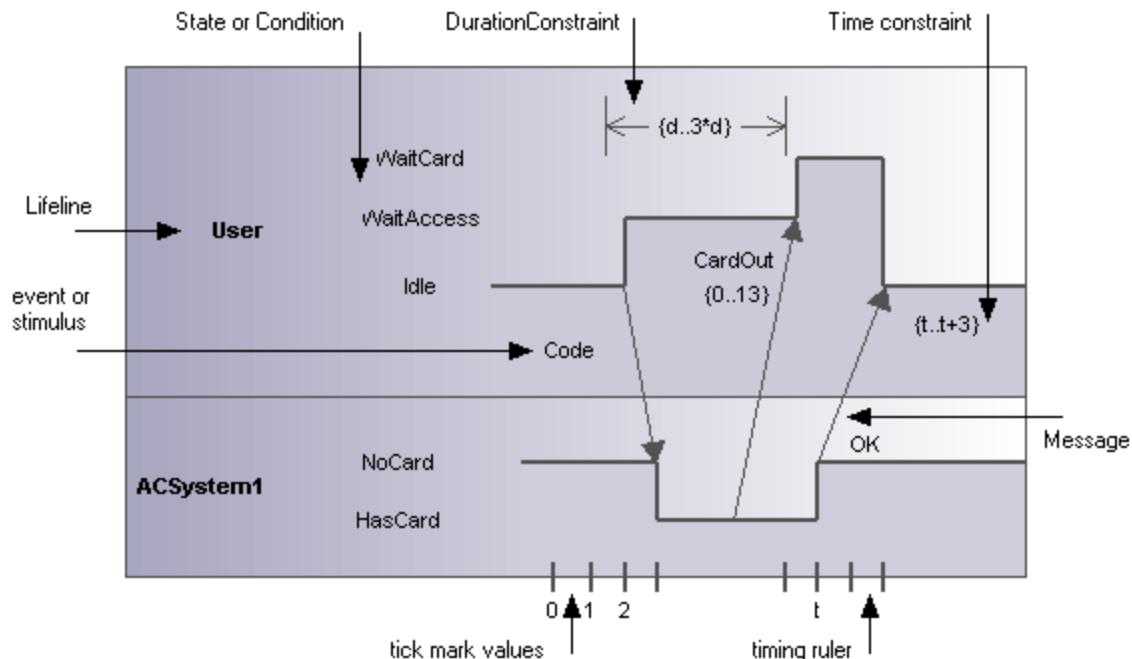
## 8.1.8 Timing Diagram

Altova website: [UML Timing diagrams](#)

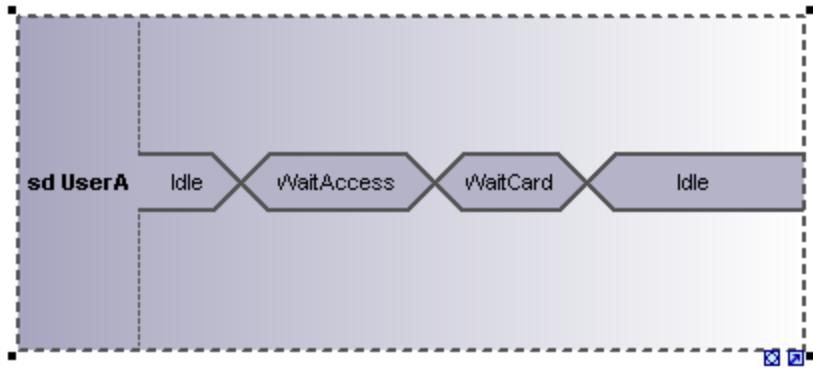
Timing diagrams depict the changes in state, or condition, of one or more interacting objects over a given period of time. States, or conditions, are displayed as timelines responding to message events, where a lifeline represents a Classifier Instance or Classifier Role.

A Timing diagram is a special form of a sequence diagram. The difference is that the axes are reversed i.e. time increases from left to right, and lifelines are shown in separate vertically stacked compartments.

Timing diagrams are generally used when designing embedded software or real-time systems.



There are two different types of timing diagram: one containing the State/Condition timeline as shown above, and the other, the General value lifeline, shown below.



### 8.1.8.1 Inserting Timing Diagram elements

#### Using the toolbar icons

1. Click the specific timing icon in the Timing Diagram toolbar.



2. Click in the Timing Diagram to insert the element. To insert multiple elements of the selected type, hold down the **Ctrl** key and click in the diagram window.

#### Dragging existing elements into the timing machine diagram

Elements occurring in other diagrams, e.g. classes, can be inserted into an Timing Diagram.

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

### 8.1.8.2 Lifeline

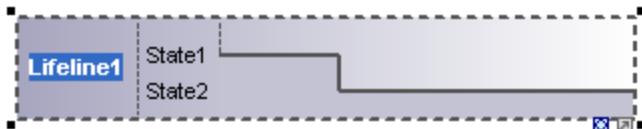
The **lifeline** element is an individual participant in an interaction, and is available in two different representations:

1. State/Condition lifeline
2. General Value lifeline

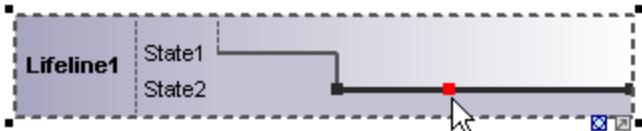
To create a **multiline** lifeline, press **Ctrl+Enter** to create a new line.

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

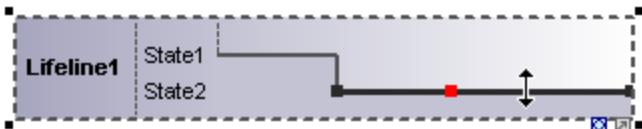
1. Click the **Lifeline (State/Condition)** icon  in the title bar, then click in the Timing Diagram to insert it.



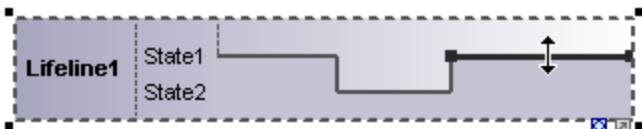
2. Enter the lifeline name to change it from the default name, Lifeline1, if necessary.
3. Place the mouse cursor over a section of one of the timelines and click left. This selects the line.
4. Move the mouse pointer to the position you want a state change to occur, and click again. Note that you will actually see the double headed arrow when you do this. A red box appears at the click position and divides the line at this point.



5. Move the cursor to the right hand side of the line and drag the line upwards.



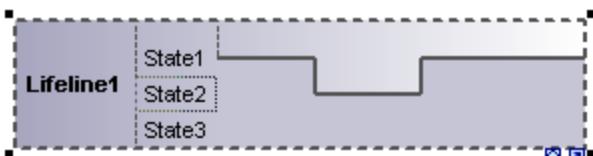
Note that lines can only be moved between existing states of the current lifeline.



Any number of state changes can be defined per lifeline. Once the red box appears on a line, clicking anywhere else in the diagram deletes it.

**To add a new state to the lifeline:**

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



**To move a state within a lifeline:**

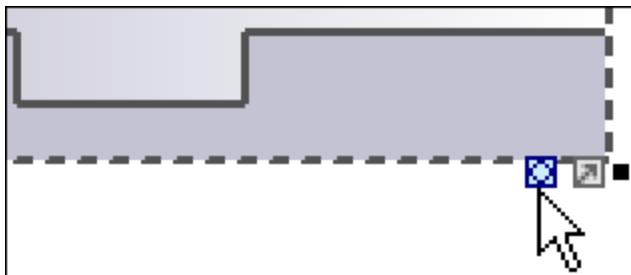
1. Click the state label that you want to move.
2. Drag it to a different position in the lifeline.

**To delete a state from a lifeline:**

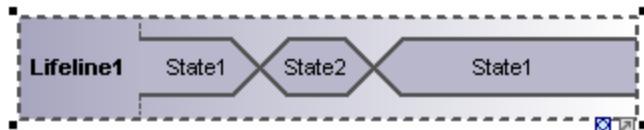
- Click the state and press the **Del.** key, or alternatively, right click and select **Delete**.

**To switch between timing diagram types:**

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



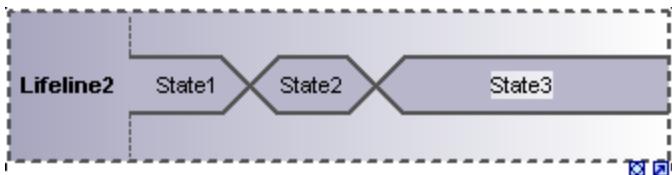
This changes the display to the General Value lifeline, the cross-over point represents a state/value change.



**Note:** Clicking the **Lifeline (General Value)** icon inserts the lifeline as shown above. You can switch between the two representations at any time.

**To add a new state to the General value lifeline:**

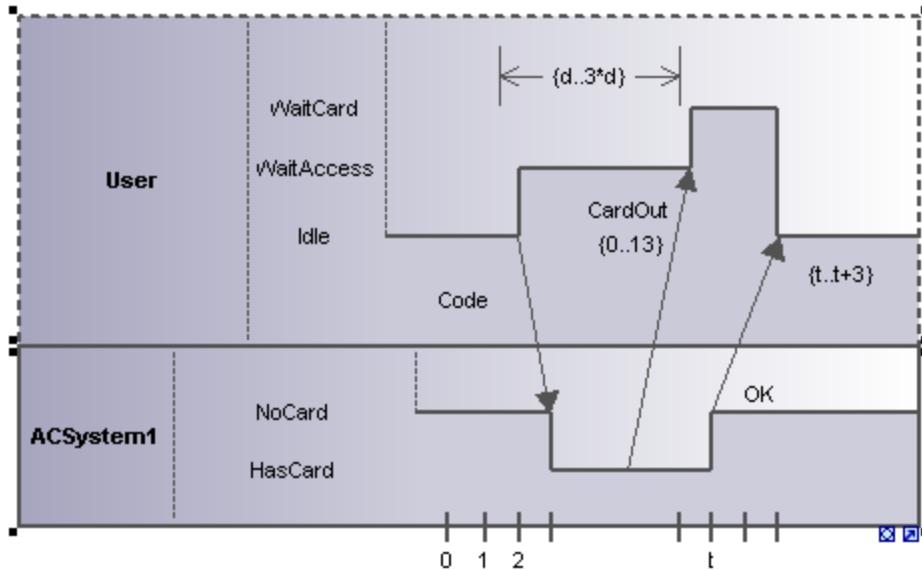
1. Right-click the lifeline and select **New | State/Condition (StateInvariant)**.
2. Edit the new name e.g. State3, and press **Enter** to confirm.



A new State is added to the lifeline.

## Grouping lifelines

Placing or stacking lifelines automatically positions them correctly and preserves any tick marks that might have been added. Messages can also be created between separate lifelines by dragging the respective message object.

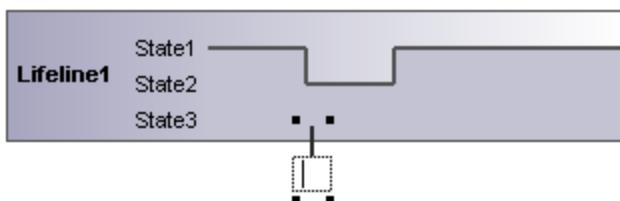


### 8.1.8.3 Tick Mark

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

#### To insert a TickMark:

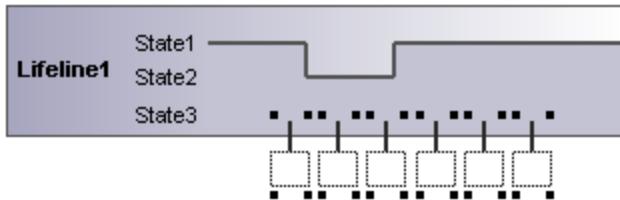
1. Click the tick mark icon and click on the lifeline to insert it.



2. Insert multiple tick marks by holding down the **Ctrl** key and repeatedly clicking at different positions on the lifeline border.
3. Enter the tick mark label in the field provided for it. Drag tick marks to reposition them on the lifeline.

### To evenly space tick marks on a lifeline:

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

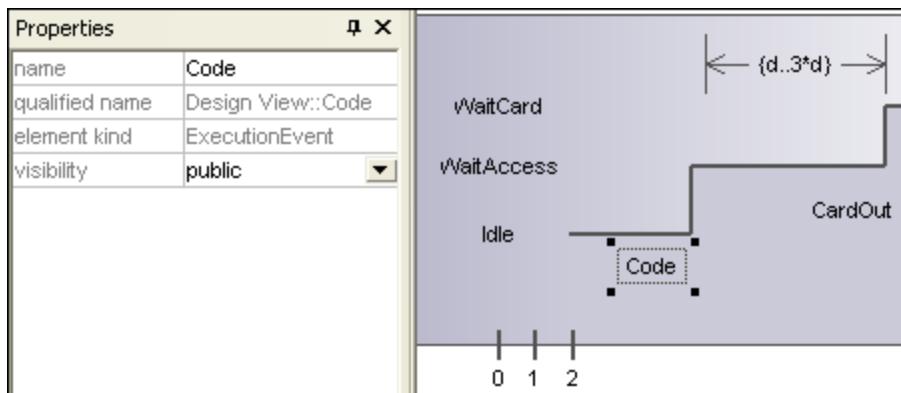


### 8.1.8.4 Event/Stimulus

The **Event/Stimulus**  ExecutionEvent is used to show the change in state of an object caused by the respective event or stimulus. The received events are annotated to show the event causing the change in condition or state.

#### To insert an Event/Stimulus:

1. Click the Event/Stimulus icon, then click the specific position in the timeline where the state change takes place.



2. Enter a name for the event, in this example the event is "Code".

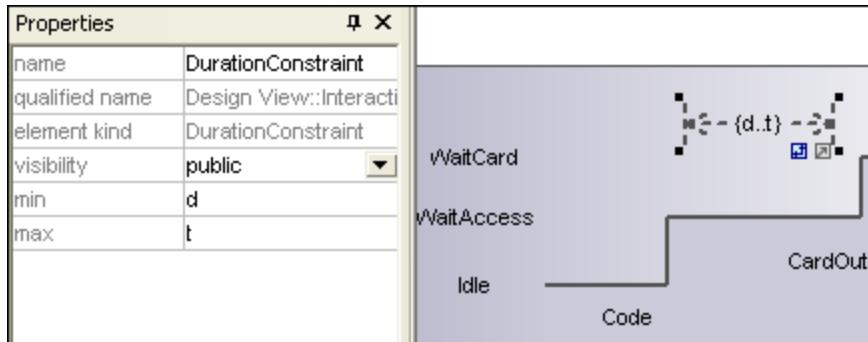
Note that the event properties are visible in the Properties tab.

### 8.1.8.5 DurationConstraint

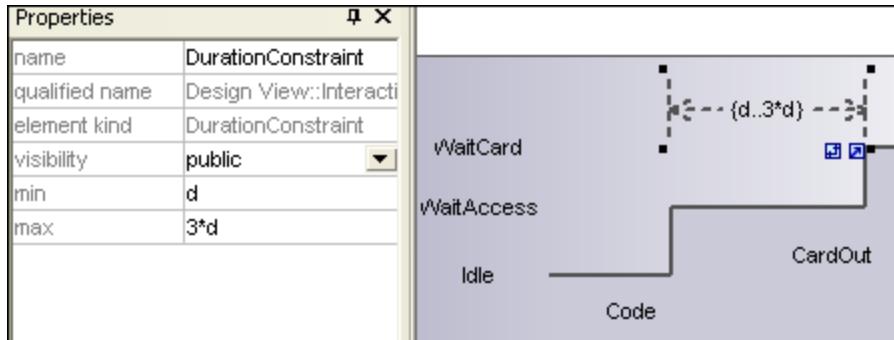
A **DurationConstraint**  defines a ValueSpecification that denotes a duration in time between a start and endpoint. A duration is often an expression representing the number of clock ticks, which may elapse during this duration.

#### To insert an DurationConstraint:

1. Click the **DurationConstraint** icon, then click the specific position on the lifeline where the constraint is to be displayed. The default minimum and maximum values, "d..t", are automatically supplied. These values can be edited by double clicking the time constraint, or by editing the values in the Properties window.

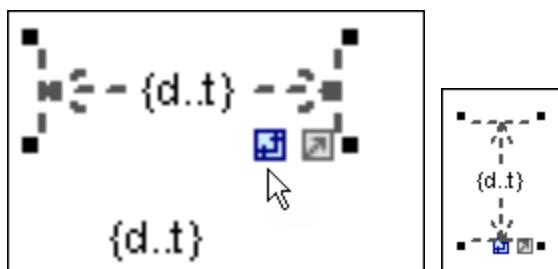


2. Use the handles to resize the object if necessary.



#### To change the orientation of the DurationConstraint:

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

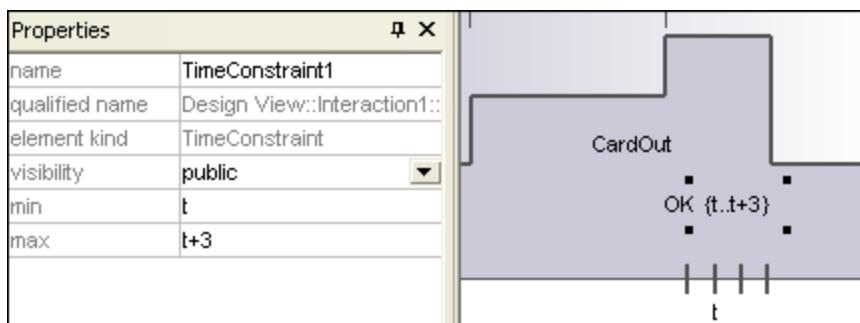


### 8.1.8.6 TimeConstraint

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

#### To insert a TimeConstraint:

- Click the **TimeConstraint** icon, then click the specific position on the lifeline where the constraint is to be displayed.



The default minimum and maximum values are automatically supplied, "d..t" respectively. These values can be edited by double clicking the time constraint, or by editing the values in the Properties window.

### 8.1.8.7 Message

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

Use the following toolbar buttons to add specific message types:

- |                                                                                     |                 |
|-------------------------------------------------------------------------------------|-----------------|
|  | Message (Call)  |
|  | Message (Reply) |

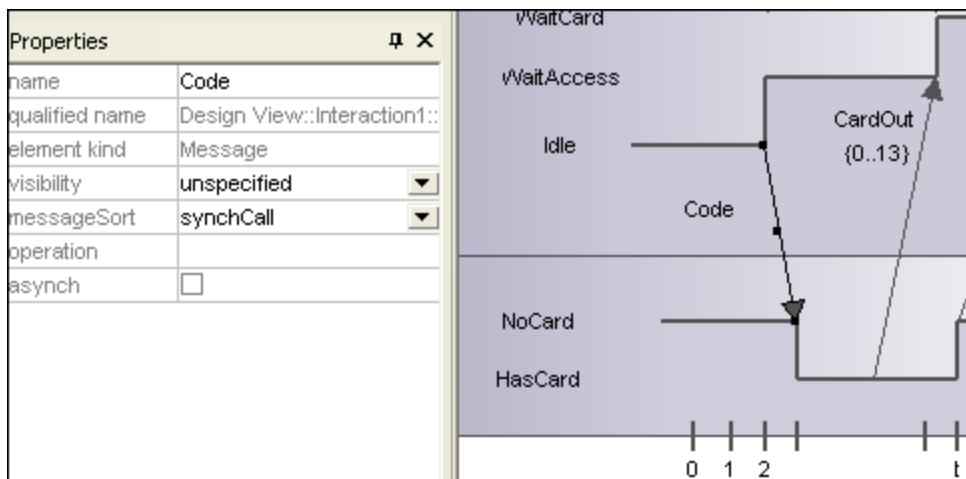


Async message (Call)

Messages are sent between sender and receiver timelines, and are shown as labeled arrows.

#### To insert a message:

1. Click the specific message icon in the toolbar.
2. Click anywhere on the timeline sender object e.g. **Idle**.
3. Drag and drop the message line onto the receiver objects timeline e.g. **NoCard**. Lifelines are highlighted when the message can be dropped.



Notes:

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

#### To delete a message:

1. Click the specific message to select it.
2. Press the **Del** key to delete it from the model, or right click it and select "Delete from diagram".

## 8.2 Structural Diagrams

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

-  [Class Diagram](#)
-  [Component Diagram](#)
-  [Composite Structure Diagram](#)
-  [Deployment Diagram](#)
-  [Object Diagram](#)
-  [Package Diagram](#)
-  [Profile Diagram](#) 404

### 8.2.1 Class Diagram

This section includes tasks and concepts applicable to Class Diagrams, as follows:

- [Customizing Class Diagrams](#) 380
- [Overriding Base Class Operations and Implementing Interface Operations](#) 387
- [Creating Getter and Setter Methods](#) 387
- [Ball and Socket Notation](#) 389
- [Adding Raised Exceptions to Methods of a Class](#) 390
- [Adding Receptions to a Class](#) 391
- [Generating Class Diagrams](#) 392

For a basic introduction to Class Diagrams, see [Class Diagrams](#) 27 in the tutorial section of this documentation.

#### 8.2.1.1 Customizing Class Diagrams

##### Expanding / hiding class compartments in a UML diagram

There are several methods of expanding the various compartments of class diagrams.

- Click on the + or - buttons of the currently active class to expand/collapse the specific compartment.
- Use the marquee (drag on the diagram background) to mark **multiple** classes, then click the expand/hide button. You can also use **Ctrl+Click** to select multiple classes.
- Press **Ctrl+A** to select **all classes**, then click the expand/collapse button, on one of the classes, to expand/collapse the respective compartments.

##### Expanding / collapsing class compartments in the Model Tree

In the Model Tree classes are subelements of packages and you can affect either the packages or the classes.

- Click the package / class you want to **expand** and:
  - Press the \* key to expand the current package/class and all sub-elements
  - Press the + key to open the current package/class.

To **collapse** the packages/classes, press the - keyboard key.

Note that you can use the standard keyboard keys, or the numeric keypad keys to achieve this.

## Changing the visibility type icons

Clicking the **visibility icon** to the left of an operation , or property , opens a drop-down list enabling you to change the visibility status. You can also change the type of visibility symbols that you want to see.

- Click a class in the diagram window, click the **Styles** tab and scroll down the list until you find the **Show Visibility** entry.



You can choose between the UModel type shown above, or the UML conformant symbols shown below.

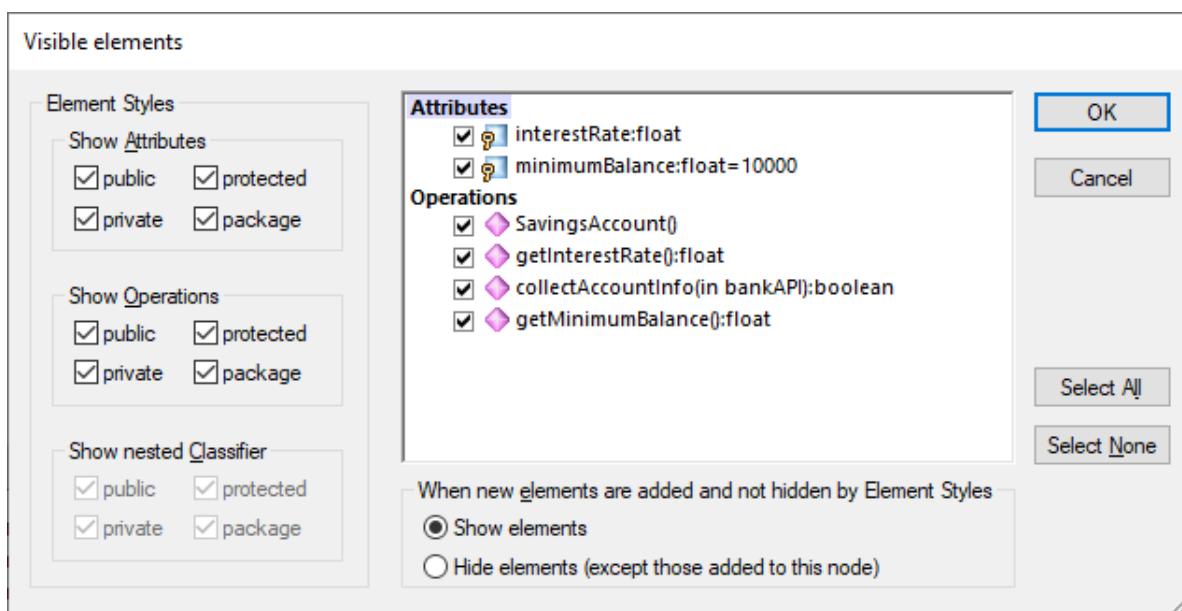


## Showing or hiding node content (class attributes, operations, slots)

In class diagrams, you can show or hide specific members of a class, such as attributes or operations. You can show or hide not only individual members but also multiple members of the same type according to their visibility. For example, you can hide only those class attributes that have private visibility. Showing or hiding is also supported for object slots (`InstanceSpecifications`) in Object diagrams.

To show or hide class members or object slots:

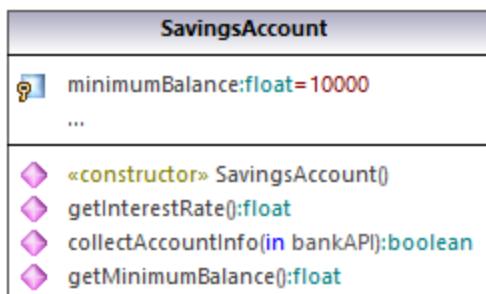
1. Right-click a class (for example, `SavingsAccount` from the example `Bank_MultiLanguage.ump` project) and select **Show/Hide Node content** from the context menu.
2. Select or clear the check box next to the members you want to show or hide, respectively.



To show or hide multiple members based on their visibility, use the check boxes in the **Element Styles** group. For example, clearing the **protected** check box in the **Show Attributes** group hides all protected attributes of the class.

**Note:** Tagged values of hidden elements are also hidden when you select the hide option.

After you confirm your preferences with **OK** and close the dialog box, any hidden members on the diagram are replaced by the ellipsis `...` symbol. To open the dialog box again, double-click the ellipsis.



The **When new elements are added and not hidden by Element Styles** option allows you to define what will be made visible when new elements are added to the class. This applies not only to elements added manually in the diagram or in the Model Tree, but also to those added automatically during the code engineering process. The valid values for this option are as follows:

<b>Show elements</b>	When a new member is added to the class, show it on the diagram. Nevertheless, if any of the options set under "Element styles" dictate that the element must be hidden, hide it.
----------------------	-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

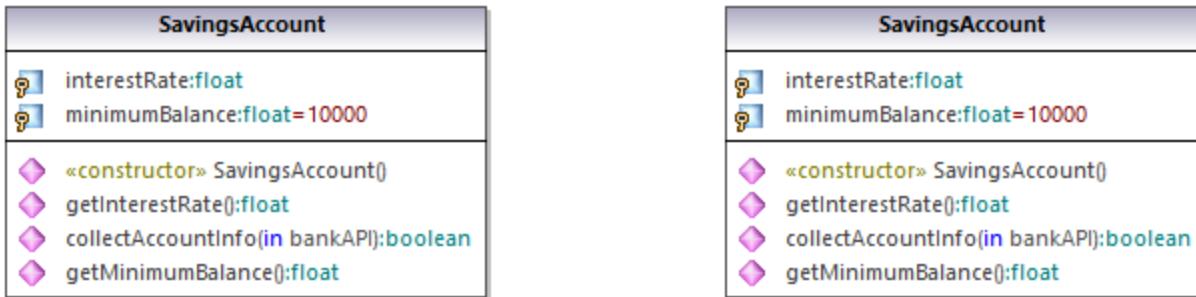
<b>Hide elements (except those added to this node)</b>	<p>Here, the term "node" refers to the current instance of the class on the diagram. (Recall that the same class can be added multiple times on the same diagram, see <a href="#">Renaming, Moving, and Copying Elements</a><sup>107</sup>.)</p> <p>When two or more instances of the same class exist on the diagram, and when a new member is added to <i>this instance</i> of the class, then hide the member in all instances of the class but show it for the current instance.</p>
--------------------------------------------------------	------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

For an example of how the options above are useful, open the **Bank\_MultiLanguage.ump** example project, and find the "Hierarchy of Account" class diagram.

Next, create a new instance of the `SavingsAccount` class, as follows:

1. Right-click the `SavingsAccount` class in the diagram and select **Copy**.
2. Right-click an empty area in the same diagram and select **Paste in this diagram only** from the context menu.

There are now two instances of the `SavingsAccount` class on the diagram.



Next, set different visibility options in each of the instances:

1. Right-click the left instance of the class, select **Show/Hide Node content**, and then select the **Show elements** option.
2. Right-click the right instance of the class, select **Show/Hide Node content**, and then select the **Hide elements (except those added to this node)** option.

Next, add a new property to the left instance (select the class and press **F7**). As illustrated below, the new property (`Property1`) is visible in the left instance but not visible in the right instance. This happens because the right-side instance of the class has the **Hide elements (except those added to this node)** option enabled.

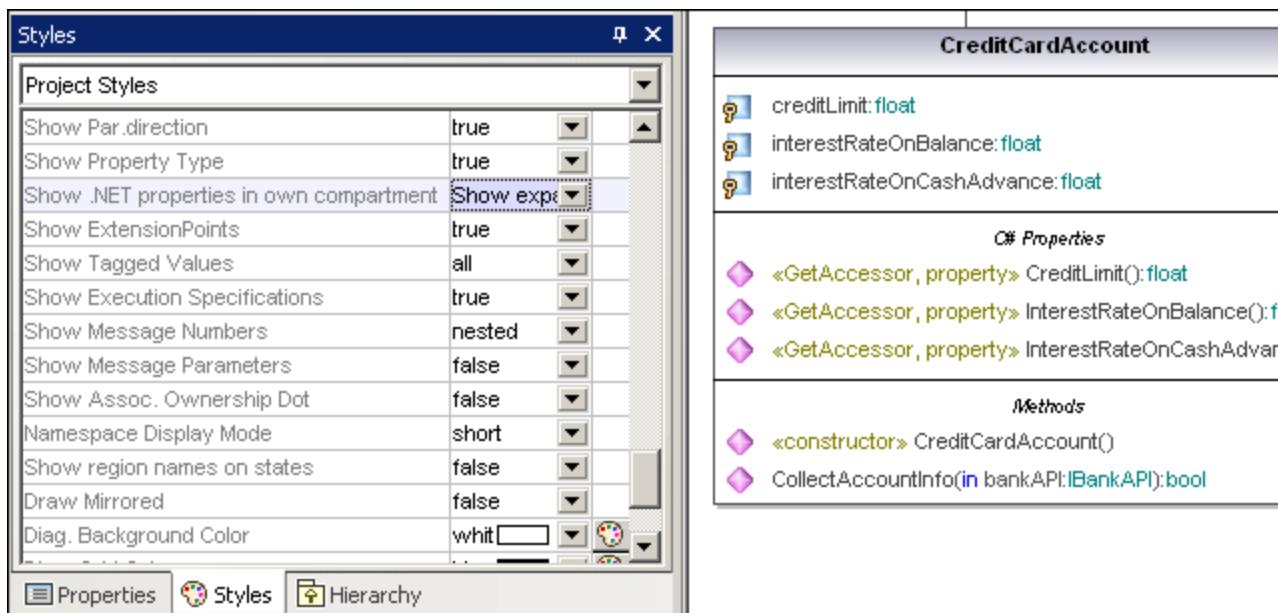


Finally, add a new property to the right-side instance of the class. As illustrated below, the new property (`Property2`) is visible in both instances. This happens because the left-side instance is configured to show new elements, while the right-side instance is the *current instance* where the property is added, so the new property is shown unconditionally.



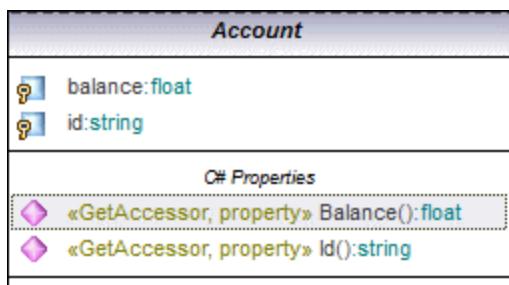
## Showing or hiding .NET compartments

To display .NET properties in their own compartment, select the "Show .NET properties in own compartment" option in the **Styles** tab.



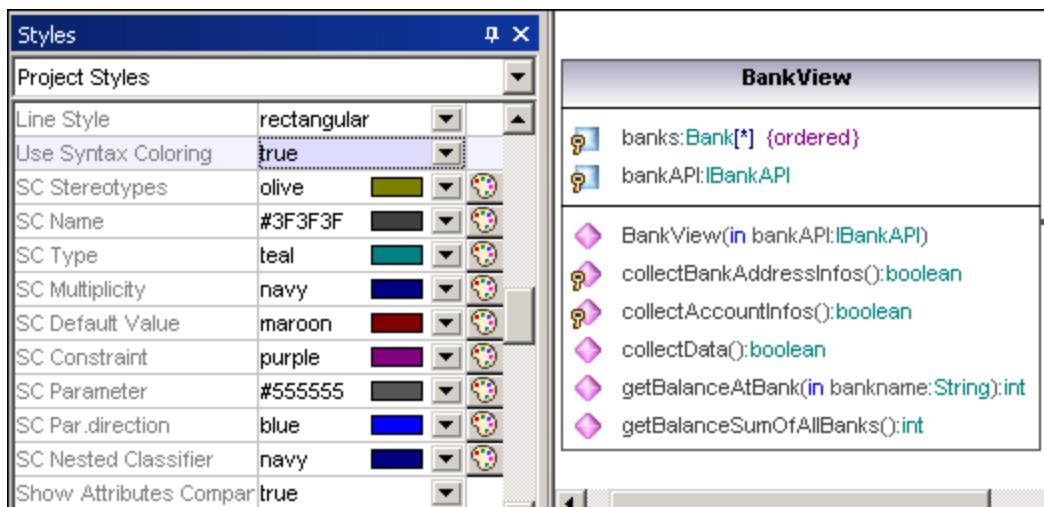
### Showing .NET properties as associations

To display .NET properties as associations, right-click a C# property as shown below, and select **Show | All .NET Properties as Associations** from the context menu.



### Changing the syntax coloring of operations/properties

UModel automatically enables syntax coloring, but lets you customize it to suit your needs. The default settings are shown below.



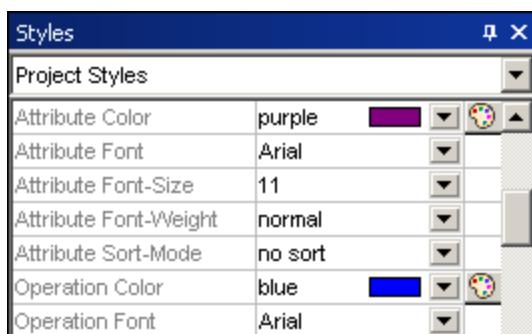
To change the default syntax coloring options (shown below):

1. Switch to the **Styles** tab and scroll the **SC** prefixed entries.
2. Change one of the "SC color" entries e.g. "SC Type" to "red".



To disable syntax coloring:

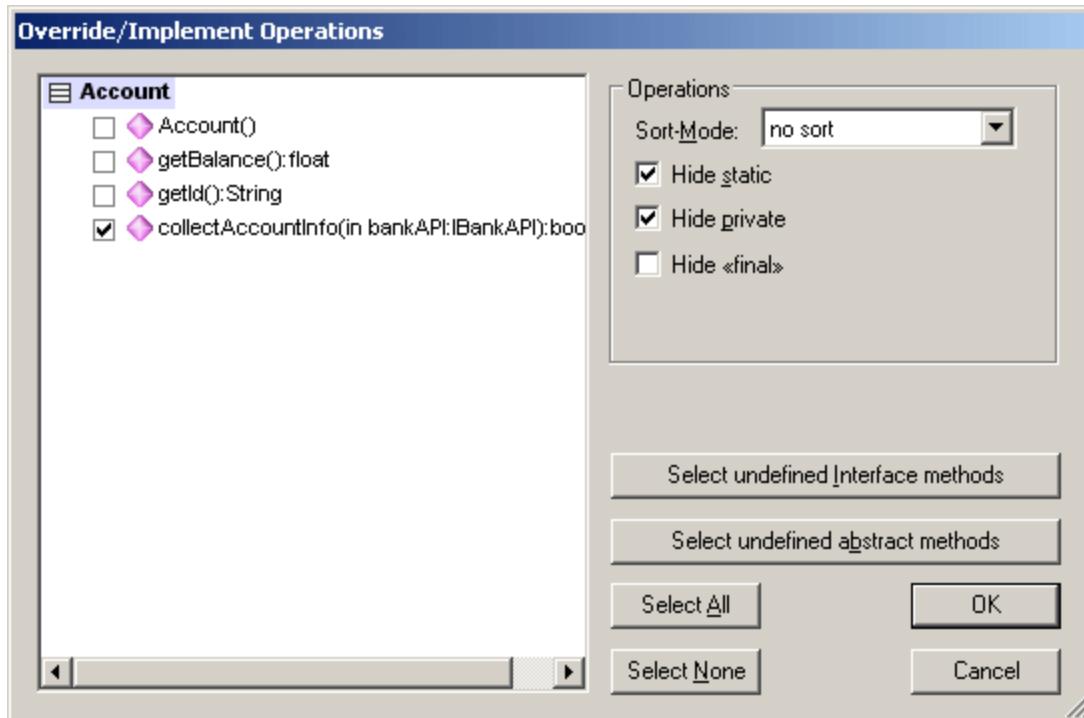
1. Switch to the **Styles** tab and change the **Use Syntax Coloring** entry to **false**.
2. Use the **Attribute Color**, or **Operation Color** entries in the **Styles** tab to customize these items in the class.



### 8.2.1.2 Overriding Base Class Operations and Implementing Interface Operations

UModel gives you the ability to override the base-class operations, or implement interface operations of a class. This can be done from the Model Tree, Favorites tab, or in Class diagrams.

1. Right-click one of the derived classes in the class diagram, e.g. `CheckingAccount`, and select **Override/Implement Operations**. This opens the dialog box shown below.



2. Select the Operations that you want to override and confirm with **OK**. The "Select undefined..." buttons select those method types in the window at left.

**Note:** When the dialog box is opened, operations of base classes and implemented interfaces that have the same signature as existing operations, are automatically checked (i.e. active).

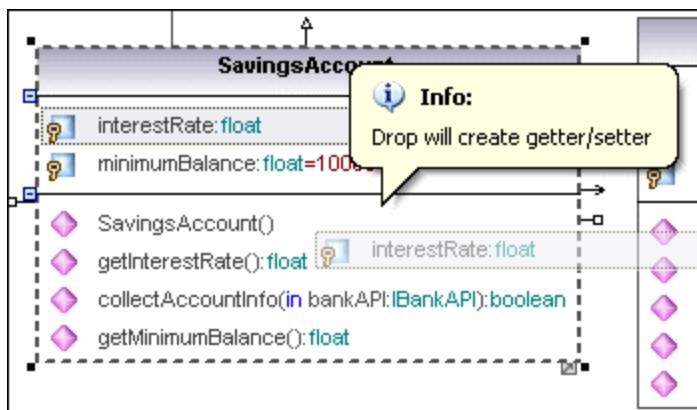
### 8.2.1.3 Creating Getter and Setter Methods

During the modeling process it is often necessary to create get/set methods for existing attributes. UModel supplies you with two separate methods to achieve this:

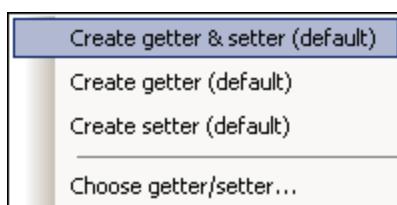
- Drag and drop an attribute into the operation compartment
- Use the context menu to open a dialog box allowing you to manage get/set methods

### To create getter/setter methods using drag and drop:

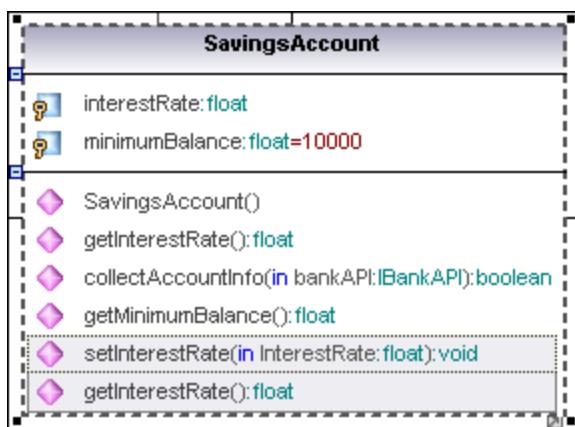
- Drag an attribute from the Attribute compartment and drop it in the Operations compartment.



A pop-up menu appears at this point allowing you to decide what type of get/set method you want to create.

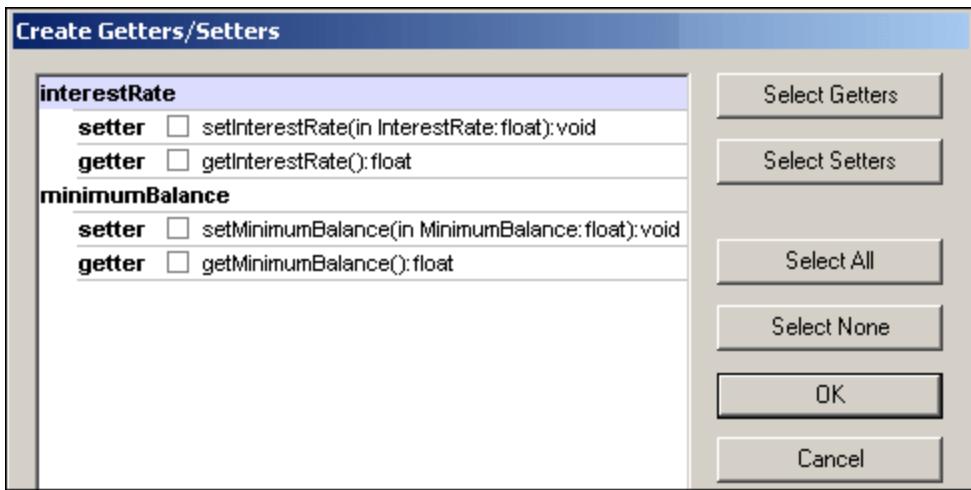


Selecting the first item creates a get and set method for `interestRate:float`.



### To create getter/setter methods using the context menu:

1. Right-click the class title, e.g. `SavingsAccount`, and select the context menu option **Create Getter/Setter Operations**. The Create Getters/Setters dialog box opens displaying all attributes available in the currently active class.

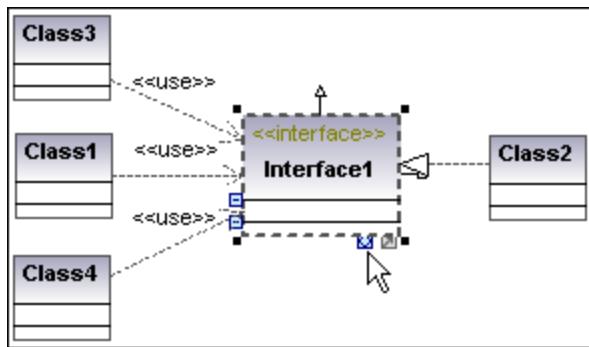


2. Use the buttons to select the items as a group, or click the getter/setter check boxes individually.

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

#### 8.2.1.4 Ball and Socket Notation

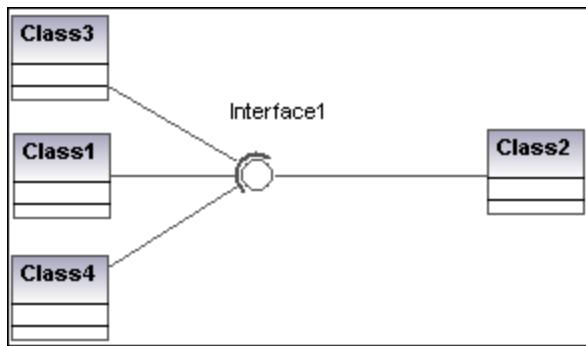
UModel supports the ball and socket notation of UML. Classes that require an interface display a "socket" and the interface name, while classes that implement an interface display the "ball".



In the shots shown above, Class2 realizes Interface1, which is used by classes 1, 3, and 4. The usage icons were used to create the usage relationship between the classes and the interface.

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

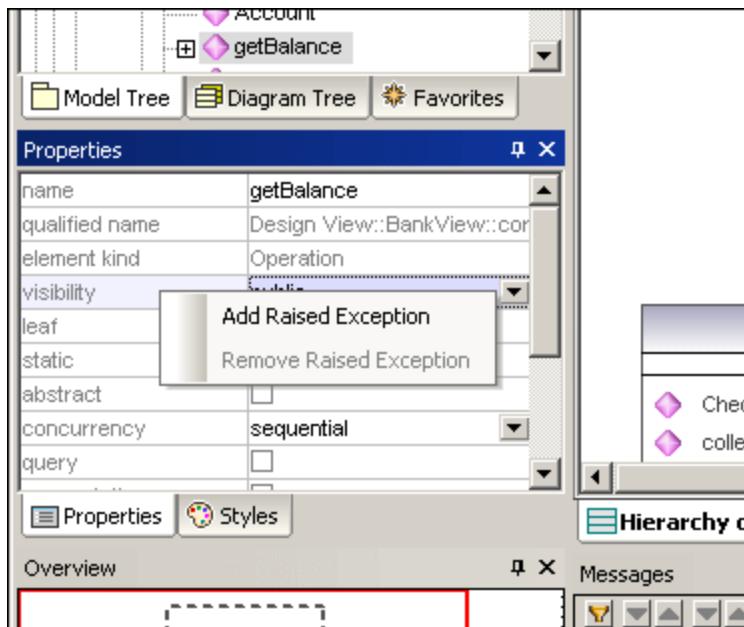
- Click the Toggle Interface notation icon at the base of the interface element.



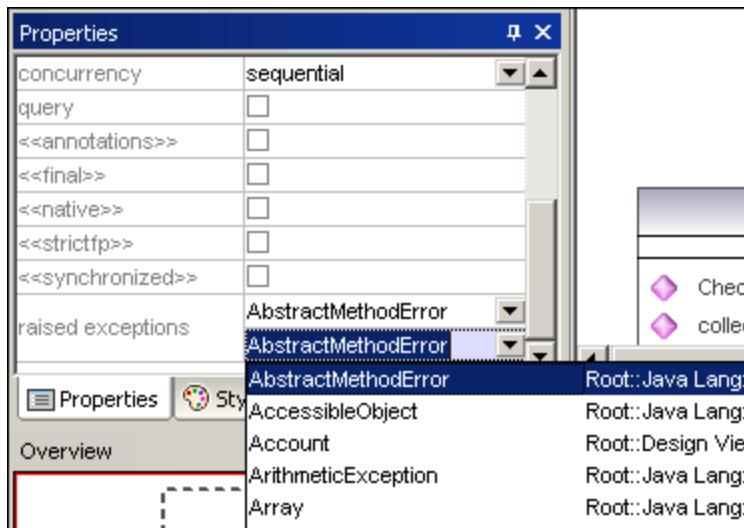
### 8.2.1.5 Adding Raised Exceptions to Methods of a Class

To add raised Exceptions to methods of a class:

1. Click the method of the class you want to add the raised exception to in the Model Tree window, e.g. `getBalance` of the `Account` class.
2. Right-click the Properties window and select **Add Raised Exception** from the pop-up menu. This adds the **raised exceptions** field to the Properties window, and automatically selects the first entry in the list.



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



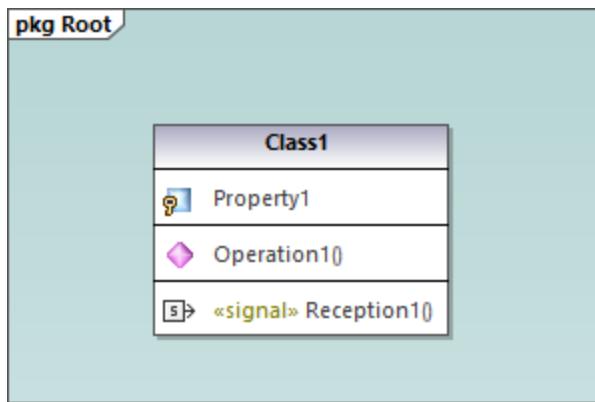
### 8.2.1.6 Adding Receptions to a Class

In addition to operations and properties, you can add Reception elements to a class.

#### To add a Reception to a class:

- Right-click the class on the diagram and select **New | Reception** from the context menu.

Receptions appear in a separate compartment on the Class diagram, similar to properties and operations, for example:

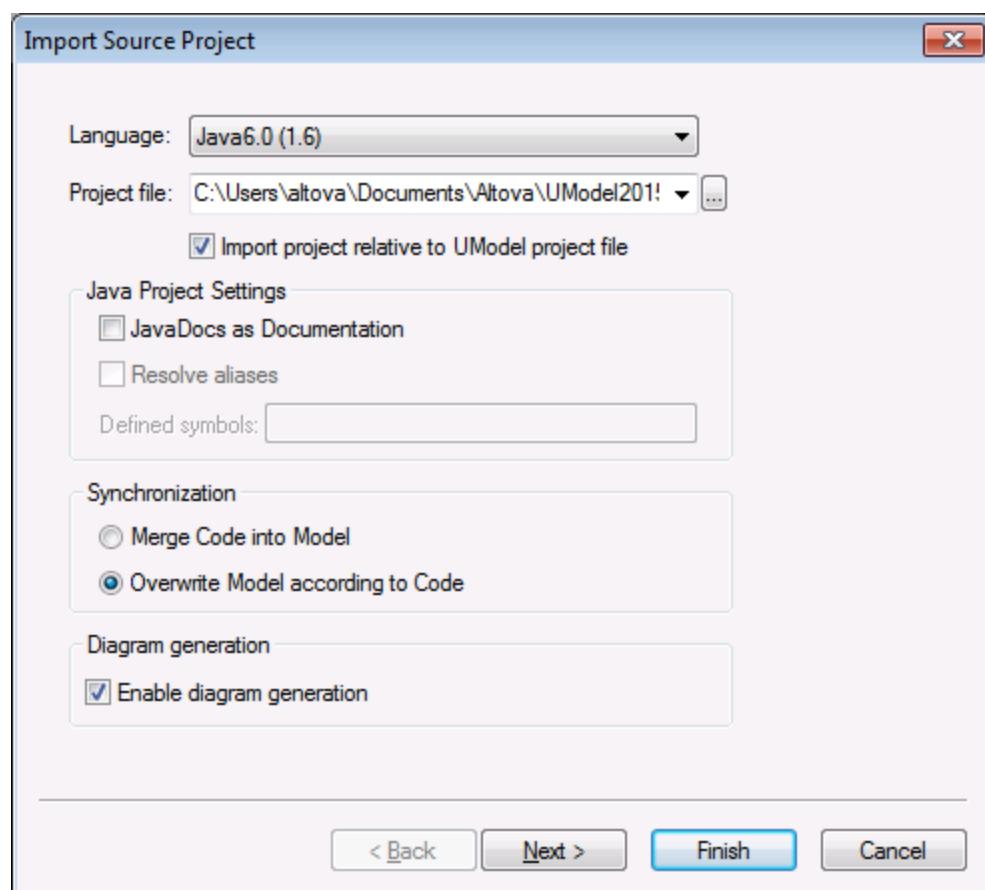


Receptions share the same styles as operations. This means that, whenever you change the style of operations, the changes affect Receptions also. For more information, see [Changing the Style of Elements](#) 117.

### 8.2.1.7 Generating Class Diagrams

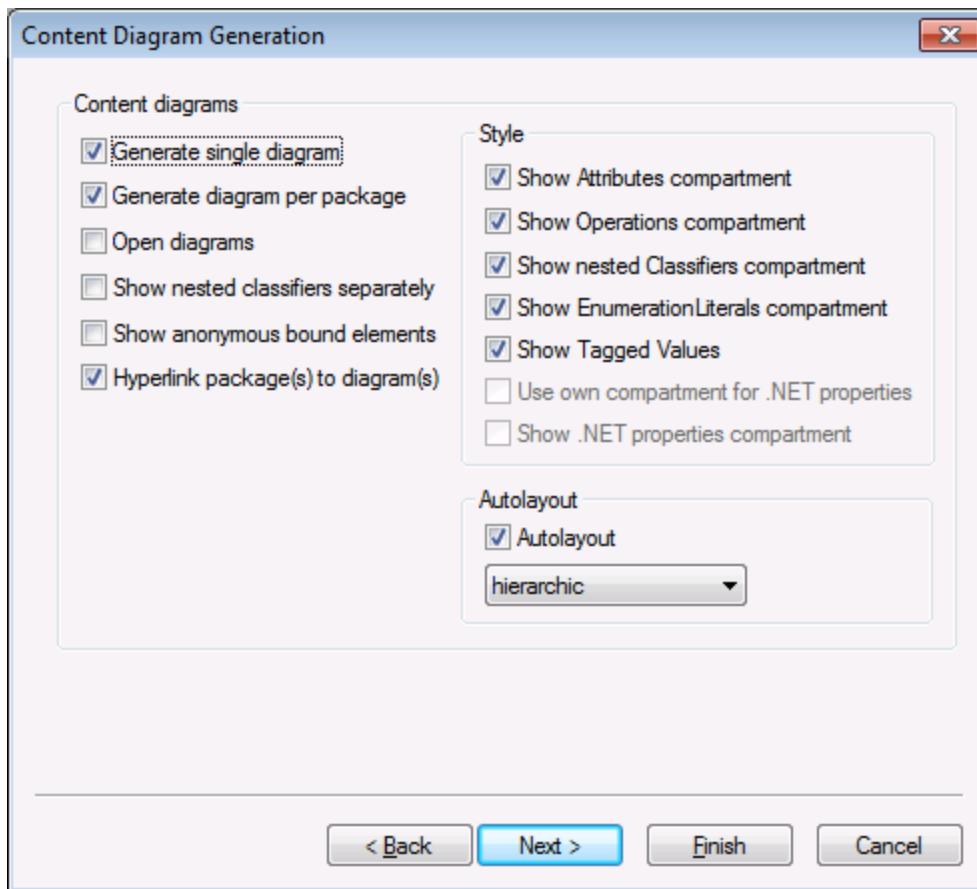
As an alternative to designing class diagrams directly in UModel, you can generate them automatically when importing source code or binaries into UModel projects (see [Importing Source Code](#)<sup>187</sup> and [Importing Java, C# and VB.NET Binaries](#)<sup>199</sup>). When following the import wizard, make sure that:

- 1) The **Enable diagram generation** check box is selected on the "Import Source Project", "Import Binary Types", or "Import Source Directory" dialog box.



Import Source Project dialog box

- 2) The **Generate single diagram** and/or the **Generate diagram per package** options are selected on the "Content Diagram Generation" dialog box.



*Content Diagram Generation dialog box*

Once the import operation is finished, any generated class diagrams are available under "Class Diagrams" in the Diagram Tree.

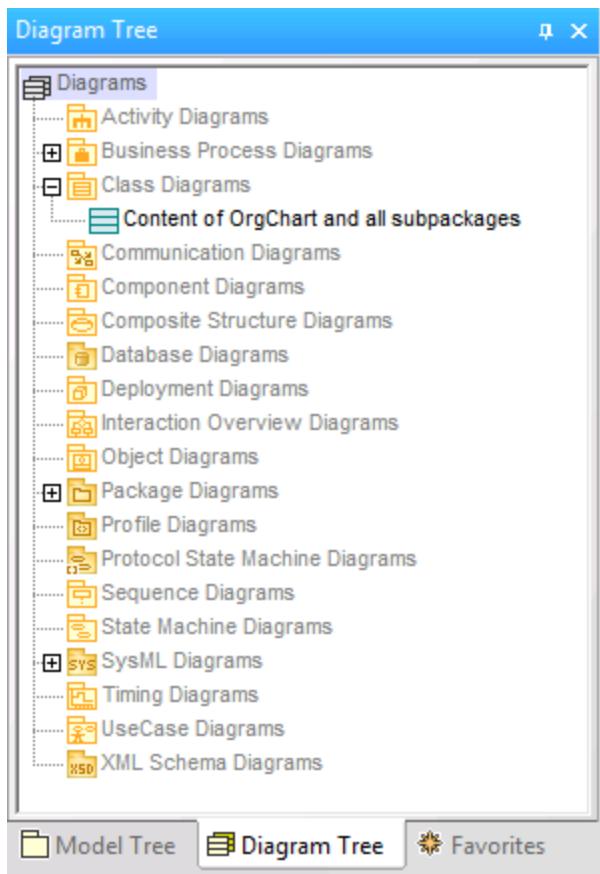
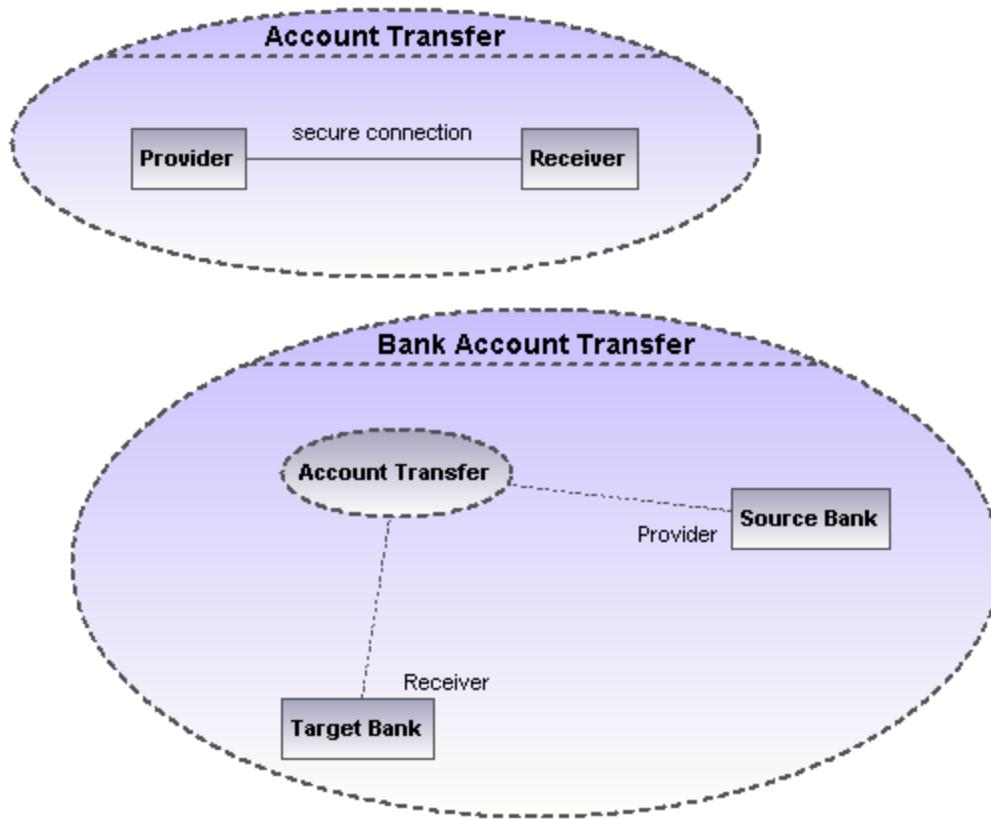


Diagram Tree

## 8.2.2 Composite Structure Diagram

Altova website: [UML Composite Structure diagrams](#)

The Composite Structure Diagram has been added in UML 2.0 and is used to show the internal structure, including parts, ports and connectors, of a structured classifier, or collaboration.



### 8.2.2.1 Inserting Composite Structure Diagram elements

#### Using the toolbar icons

1. Click the specific Composite Structure diagram icon in the toolbar.



2. Click in the Composite Structure diagram to insert the element. To insert multiple elements of the selected type, hold down the **Ctrl** key and click in the diagram window.

#### Dragging existing elements into the Composite Structure diagram

Most elements occurring in other Composite Structure diagrams, can be inserted into an existing Composite Structure diagram.

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



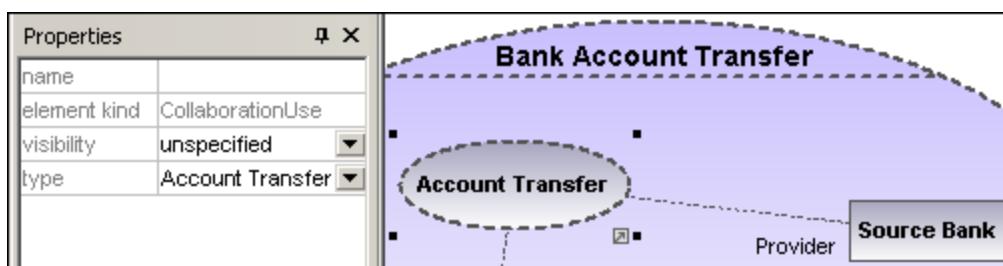
### Collaboration

Inserts a collaboration element which is a kind of classifier-instance that communicates with other instances to produce the behavior of the system.



### CollaborationUse

Inserts a Collaboration use element which represents one specific use of a collaboration involving specific classes or instances playing the role of the collaboration. A collaboration use is shown as a dashed ellipse containing the name of the occurrence, a colon, and the name of the collaboration type.



When creating dependencies between collaboration use elements, the "type" field must be filled to be able to create the role binding, and the target collaboration must have at least one part/role.



### Part (Property)

Inserts a part element which represents a set of one or more instances that a containing classifier owns. A Part can be added to collaborations and classes.



### Port

Inserts a port element which defines the interaction point between a classifier and its environment, and can be added on parts with a defined type.



### Class

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



### Connector

Inserts a Connector element which can be used to connect two or more instances of a part, or a port. The connector defines the relationship between the objects and identifies the communication between the roles.

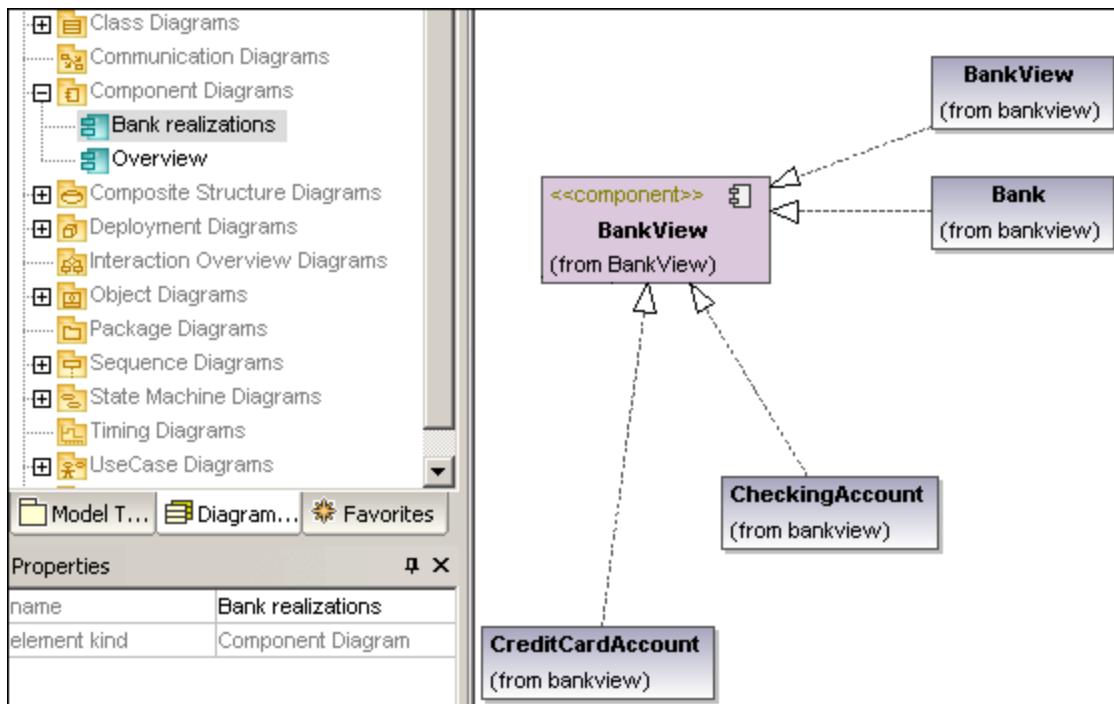


### Dependency (Role Binding)

Inserts the Dependency element, which indicates which connectable element of the classifier or operation, plays which role in the collaboration.

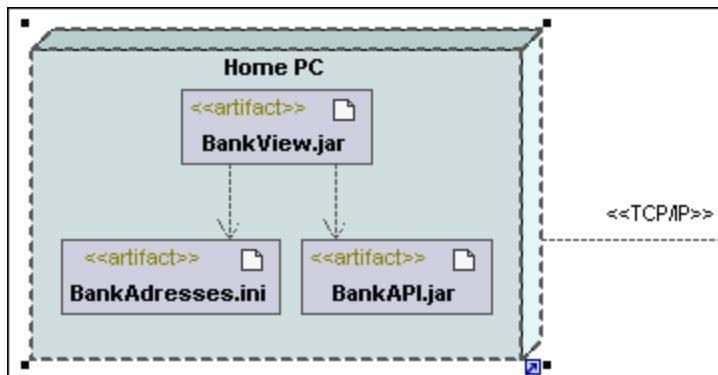
## 8.2.3 Component Diagram

Please see the [Component Diagrams](#) 49 section in the tutorial for more information on how to add component elements to the diagram.



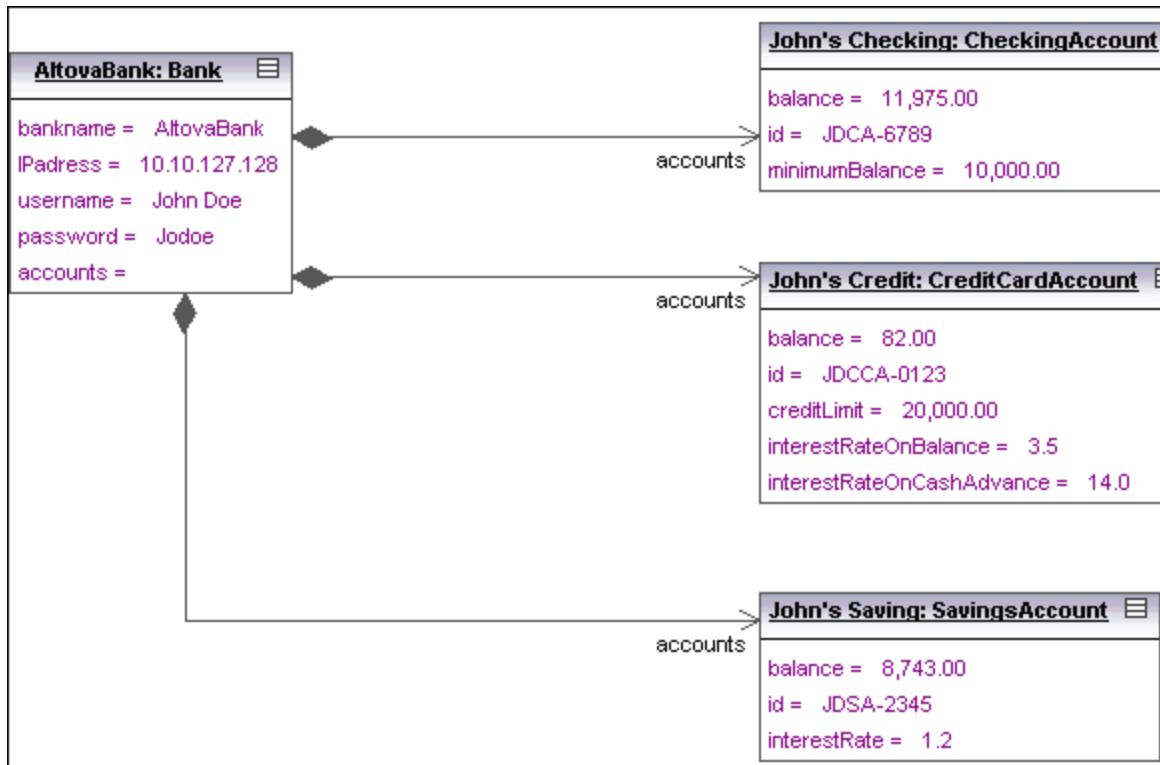
## 8.2.4 Deployment Diagram

Please see the [Deployment Diagrams](#) 55 section in the tutorial for more information on how to add nodes and artifacts to the diagram.



## 8.2.5 Object Diagram

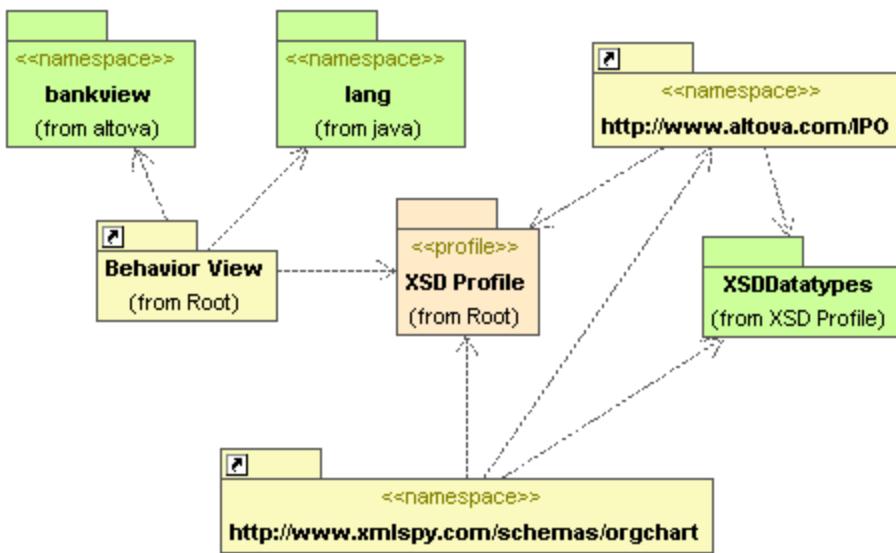
Please see the [Object Diagrams](#) (42) section in the tutorial for more information on how to add new objects/instances to the diagram.



## 8.2.6 Package Diagram

Package diagrams display the organization of packages and their elements, as well as their corresponding namespaces. UModel additionally allows you to create a hyperlink and navigate to the respective package content.

Packages are depicted as folders and can be used on any of the UML diagrams, although they are mainly used on use-case and class diagrams.



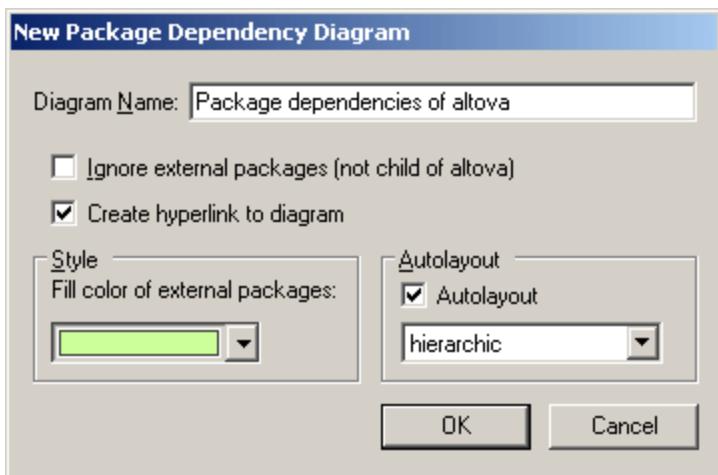
## Automatic Package Dependency diagram generation

You can generate a package dependency diagram for any package that already exists in the Model Tree.

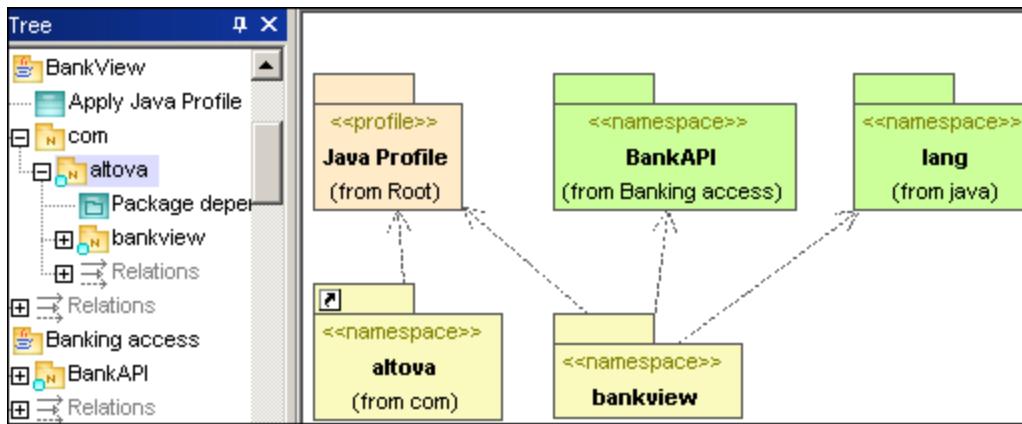
Dependency links between packages are created if there are any references between the modeling elements of those packages. E.g. Dependencies between classes, derived classes, or if attributes have types that are defined in a different package.

### To generate a package dependency diagram:

1. Right click a package in the Model Tree, e.g. altova, and select **Show in new Diagram | Package Dependencies...**. This opens the New Package Dependency Diagram dialog box.



2. Select the specific options you need and click OK to confirm.



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

### 8.2.6.1 Inserting Package Diagram elements

#### Using the toolbar icons

1. Click the specific icon in the Package Diagram toolbar.



2. Click in the diagram to insert the element. To insert multiple elements of the selected type, hold down the **Ctrl** key and click in the diagram window.

#### Dragging existing elements into the Package Diagram

Elements occurring in other diagrams, e.g. other packages, can be inserted into a Package diagram.

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



#### Package

Inserts the package element into the diagram. Packages are used to group elements and also to provide a namespace for the grouped elements. Being a namespace, a package can import individual elements of other packages, or all elements of other packages. Packages can also be merged with other packages.



#### Profile

Inserts the Profile element, which is a specific type of package that can be applied to other packages.

The Profiles package is used to extend the UML meta model. The primary extension construct is the Stereotype, which is itself part of the profile. Profiles must always be related to a reference meta model such as UML, they cannot exist on their own.



### Dependency

Inserts the Dependency element, which indicates a supplier/client relationship between modeling elements, in this case packages, or profiles.



### PackageImport

Inserts an <<import>> relationship which shows that the elements of the included package will be imported into the including package. The namespace of the including package gains access to the included namespace; the namespace of the included package is not affected.

**Note:** Elements defined as "private" within a package, cannot be merged or imported.



### PackageMerge

Inserts a <<merge>> relationship which shows that the elements of the merged (source) package will be imported into the merging (target) package, including any imported contents of the merged (source) package.

If the same element exists in the target package then these elements' definitions will be expanded by those from the target package. Updated or added elements are indicated by a generalization relationship back to the source package.

**Note:** Elements defined as "private" within a package, cannot be merged or imported.



### ProfileApplication

Inserts a Profile Application which shows which profiles have been applied to a package. This is a type of package import that states that a Profile is applied to a Package.

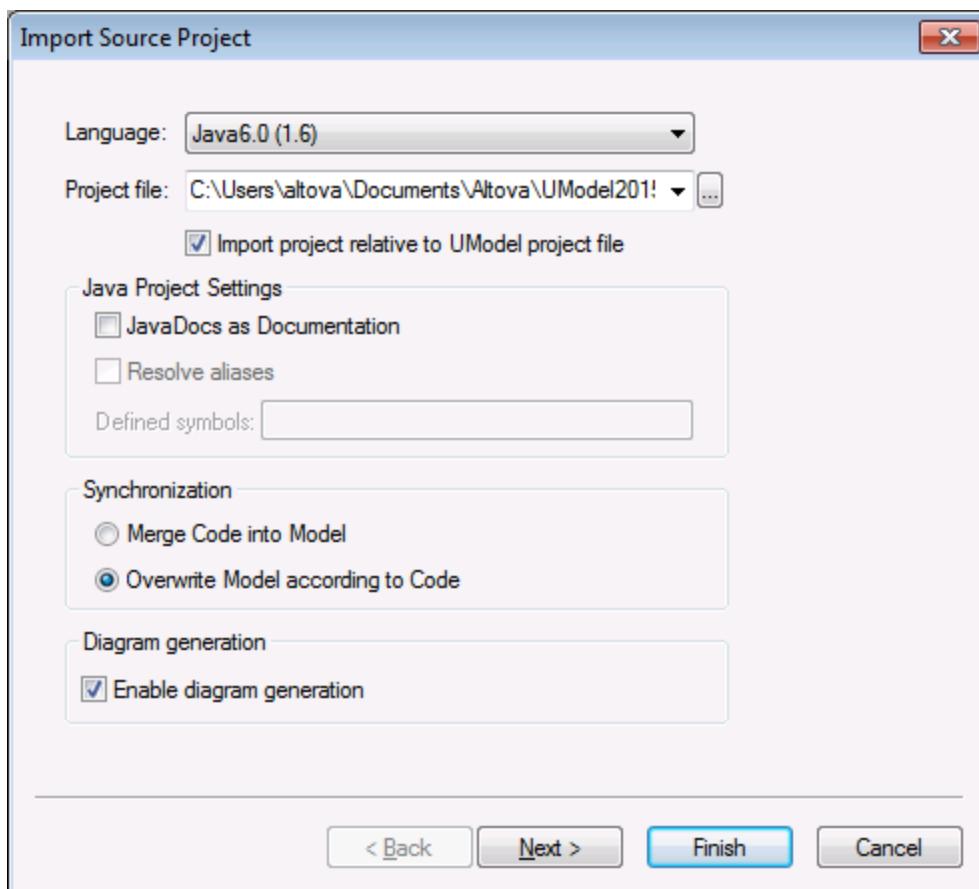
The Profile extends the package it has been applied to. Applying a profile, using the **ProfileApplication** icon, means that all stereotypes that are part of it, are also available to the package.

Profile names are shown as dashed arrows from the package to the applied profile, along with the <<apply>> keyword.

## 8.2.6.2 Generating Package Diagrams

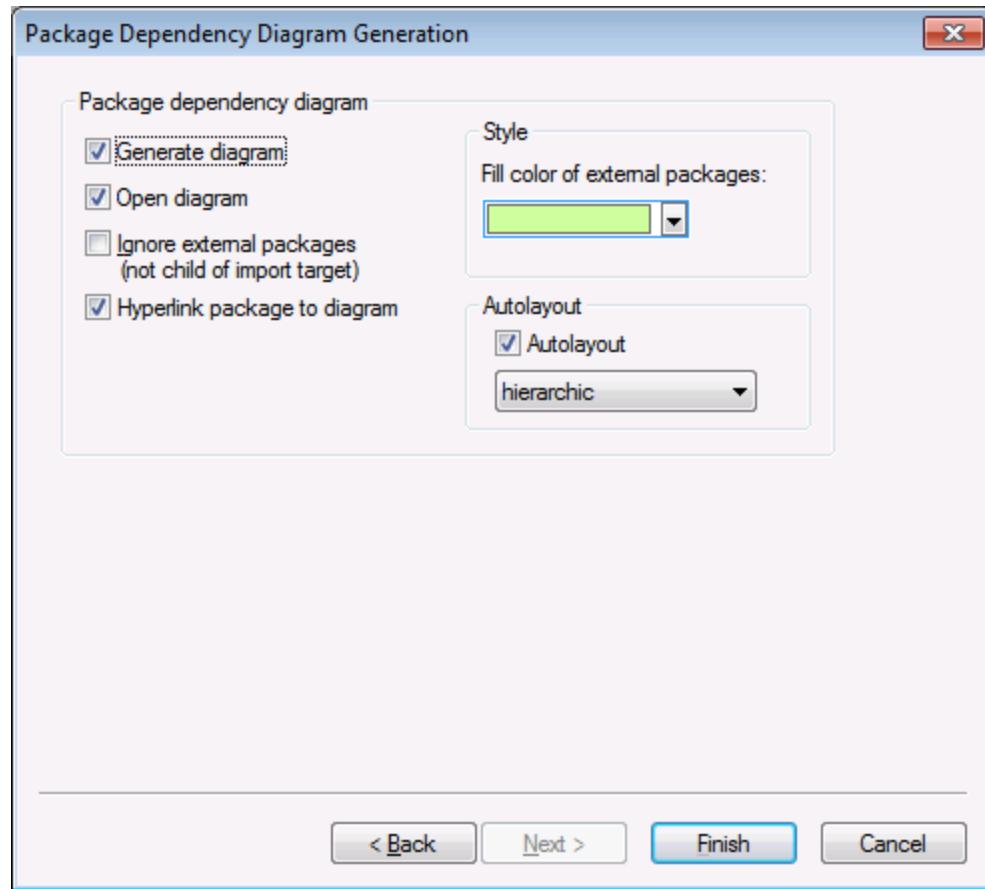
You can instruct UModel to generate package diagrams when importing source code or binaries into the UModel project (see [Importing Source Code](#)<sup>187</sup> and [Importing Java, C# and VB.NET Binaries](#)<sup>199</sup>). When following the import wizard, make sure that:

- 1) The **Enable diagram generation** check box is selected on the "Import Source Project", "Import Binary Types", or "Import Source Directory" dialog box.



*Import Source Project dialog box*

2) The **Generate diagram** option is selected on the "Package Dependency Diagram Generation" dialog box.



*Package Dependency Diagram Generation dialog box*

Once the import operation is finished, any generated package diagrams are available under "Package Diagrams" in the Diagram Tree.

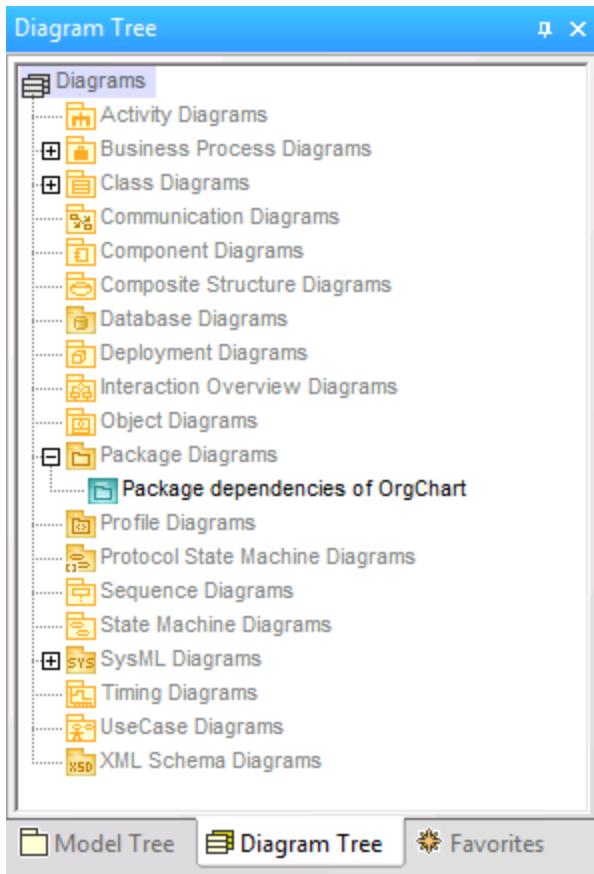


Diagram Tree

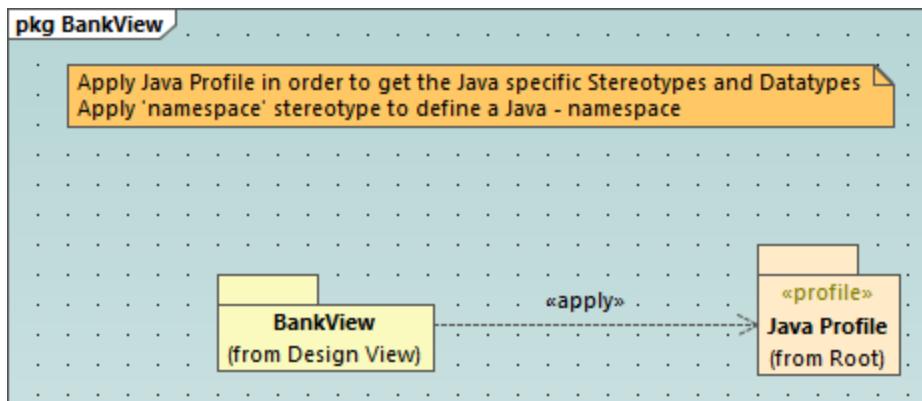
### 8.2.7 Profile Diagram

Altova website: [UML profile diagrams](#)

In UML, profiles are a way to extend UML to a specific platform or domain. Unlike a package, a profile is in the meta-model and consists of "meta" building blocks that extend or constrain something. This is possible with the help of the following extension mechanisms included into a profile: stereotypes, tagged values, and constraints.

In UModel, the profile diagram is where you can conveniently create your own stereotypes, tagged values and constraints bundled as a custom profile. Profiles enable you to extend or adapt UML to your specific domain or customize the appearance of elements in your modeling projects. For example, you may want to define custom styles or add custom icons for UML elements such as classes, interfaces, and so on.

Importantly, the profile diagram is where you can *apply a profile* to a package. For example, the profile diagram below illustrates a **ProfileApplication** relationship between the package **BankView** and the Java profile built into UModel. You can find this diagram in the following sample project: **C:\Users\<username>\Documents\Altova\UModel2022\UModelExamples\BankView\_Java.ump**; it is called "Apply Java Profile".

*Profile diagram*

The applied Java profile means that any class or interface that is part of the **BankView** package (or will be added to this package in future) must look like a Java class or interface and all its members must exhibit behavior specific to that language. For example:

- All Java data types that exist in the profile are available for selection from a drop-down list when you design a class in a class diagram, see also [Class Diagrams](#)<sup>27</sup>.
- All Java-specific stereotypes defined in the profile, such as «annotations», «final», «static», «strictfp», and so on, are visible as properties in the Properties window when you select an element.

This chapter describes how you can extend UModel projects by means of custom profiles and stereotypes. For information about using the UModel built-in profiles, see [Applying UModel Profiles](#)<sup>154</sup> and [Stereotypes and Tagged Values](#)<sup>140</sup>.

### 8.2.7.1 Creating and Applying Custom Profiles

The instructions below show you how to create a custom UModel profile and apply it to a package. This is typically required if you need to create and apply stereotypes beyond those included in the default UModel profiles. For information about applying the default UModel profiles, see [Applying UModel Profiles](#)<sup>154</sup>.

#### To create a custom profile:

1. Right-click the package where you would like to create the new profile, (for example, "Root"), and select **New element | Profile** from the context menu.
2. Create all the elements that should be part of this profile, such as stereotypes, data types, and so on. You can do this either in the Model Tree window or from a profile diagram. For example, to create a new stereotype in the model, right-click the profile and select **New element | Stereotype** from the context menu. See also [Creating Stereotypes](#)<sup>406</sup>.
3. Optionally, create a profile diagram (right-click the profile and select **New diagram | Profile diagram** from the context menu). To add all the required elements to the diagram, use the standard UModel menu commands and toolbars, see [How to Model...](#)<sup>103</sup>.

If you would like to create the profile from a profile diagram, make sure that the diagram is owned by

(created under) a profile, or by a package inside a profile.

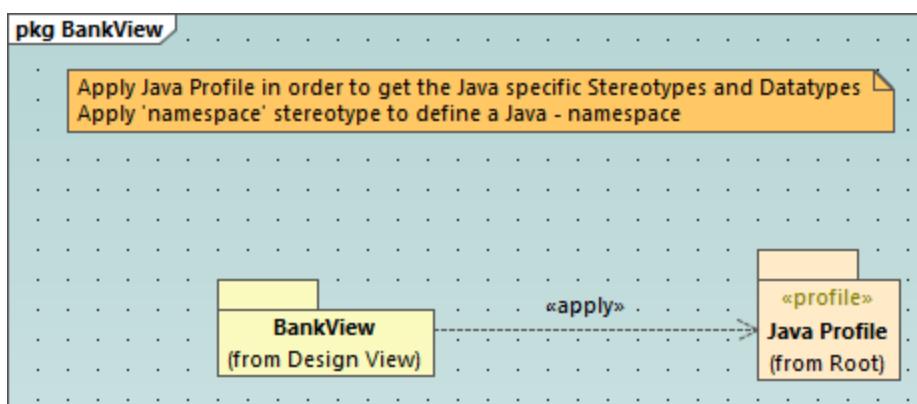
In addition, if you would like to reuse the profile across multiple UModel projects, do the following:

1. Share any packages that you want to make reusable. (Right-click the package or the profile itself, and select **Subproject | Share package** from the context menu.)
2. Save the project to a directory from where you can later include it as a subproject, see [Including Subprojects](#) 158.

So far, you have created a profile but have not added (or applied) it to any package. By applying a profile to a package, you make all of the extension mechanisms of that profile (such as stereotypes, data types, and so on) available to elements of the package.

#### To apply a custom profile to a package:

1. Create a new UModel project, or open an existing one.
2. Do one of the following:
  - a. Create your custom profile in the existing project, as shown above.
  - b. Include a custom profile from an existing project using the menu command **Project | Include Subproject**. Note that either the entire profile or its packages under must be shared in order to be reusable, see [Sharing Packages and Diagrams](#) 160.
3. Right-click the profile and select **New diagram | Profile diagram** from the context menu.
4. Add some package(s) and the custom profile to the diagram.
5. Draw a **ProfileApplication**  relationship from the package to the profile. For example, the profile diagram below illustrates a **ProfileApplication** relationship between the package **BankView** and the Java profile built into UModel. As illustrated below, profile applications are shown as dashed arrows from the package to the applied profile, along with the <>apply>> keyword.



#### 8.2.7.2 Creating Stereotypes

When you model projects using any of the UModel built-in profiles (such as C#, Java, VB.NET, XML schema, and so on), you shouldn't typically need to create any custom stereotypes. Instead, you can just apply the existing stereotypes to your model's elements, as described in [Applying Stereotypes](#) 142.

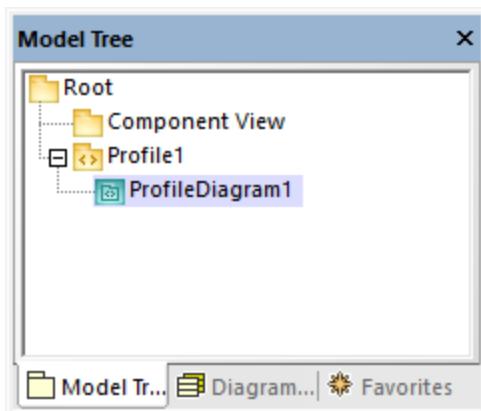
However, if you would like to add custom icons to elements or customize their appearance based on the applied stereotype, this can be achieved by creating custom stereotypes. Note the following prerequisites:

- Stereotypes must be owned by a profile or a package inside a profile. Therefore, in order to create a stereotype, you must create a profile first (or a package inside an existing profile).
- After creating the profile, you must apply it to the package where you need to use the custom stereotypes, as described in [Creating and Applying Profiles](#) <sup>405</sup>.

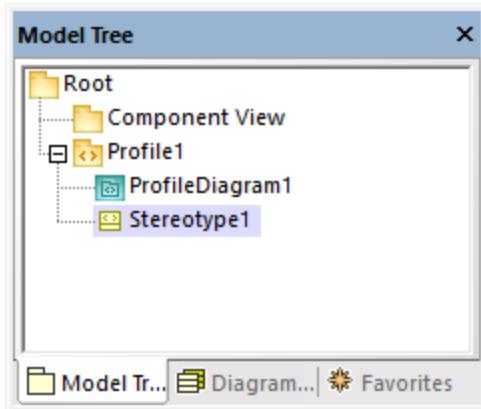
Once you have created a profile, you can start adding stereotypes to it. This can be done either directly in the Model Tree window, or from a profile diagram. If you would like to create stereotypes from a profile diagram, make sure that the diagram is owned by (created under) a profile, or by a package inside a profile, as shown below.

#### To create a stereotype:

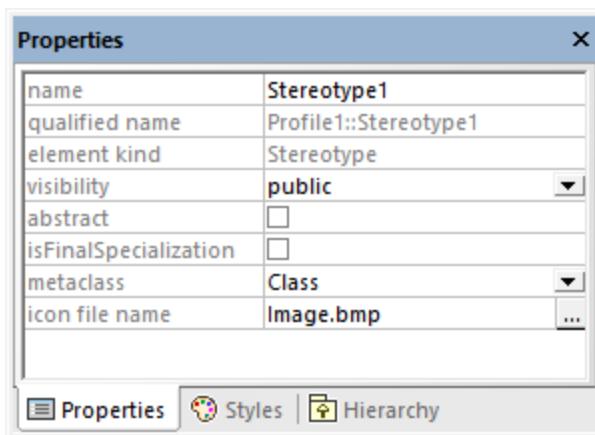
1. If you haven't done so already, create a profile, see [Creating and Applying Custom Profiles](#) <sup>405</sup>.
2. Optionally, right-click the profile and select **New diagram | Profile diagram** from the context menu. This creates a new profile diagram under the current profile—it will help you visualize in one place all the stereotypes, data types, and other elements that you will subsequently add to the profile.



3. Right-click the profile in the Model Tree window, and select **New element | Stereotype** from the context menu.



4. Optionally, set the stereotype properties in the Properties window. For example, if you set the stereotype's **metaclass** to "Class", the stereotype will apply to classes only. Likewise, you can set a custom icon for the stereotype by clicking the **Ellipsis**  button next to **icon file name**.



#### Notes

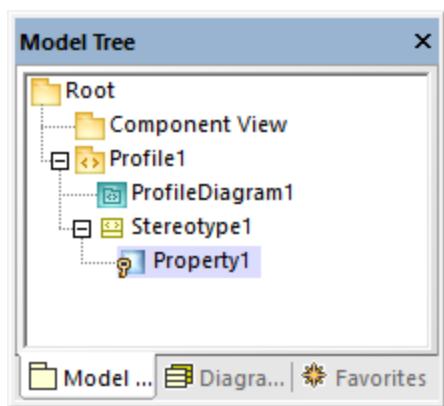
- If the image path is relative, it must be relative to the UModel project's folder.
- To use custom icons with transparent background, set their background color to RGB value 82,82,82.
- To display stereotypes for association relationships, set the **Show MemberEnd stereotypes** property to "true" in the **Styles** window.

### Adding stereotype attributes (properties)

The stereotype created above is very simple and does not have any attributes (properties) associated with it. It is, however, possible to add properties to a stereotype. Such properties will become tagged values when this stereotype is applied to some element in future.

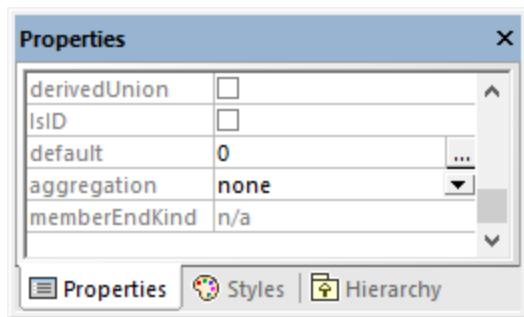
#### To add attributes (properties) to a stereotype:

1. Click the stereotype in the Model Tree window or on the diagram.
2. Do one of the following
  - a. Right-click the stereotype and select **New | Property** from the context menu.
  - b. Press **F7**.



You can set the data type of each property from the Properties window, by selecting a value from the **type** list. Any data type previously defined in the same profile as the stereotype is available for selection. If the profile doesn't contain any data types yet, you can define one by right-clicking the profile diagram, and selecting **New | Data type** from the context menu.

To set the default value of a property, enter that value in the **default** field of the Properties window. For example, the stereotype property illustrated below has "0" as default value:



The data type of a stereotype attribute (property) can also be an enumeration, see [Example: Creating and Applying Stereotypes](#) 409.

### 8.2.7.3 Example: Creating and Applying Stereotypes

This example provides a step-by-step demo of the stereotype creation process. It shows you how to achieve the following goals:

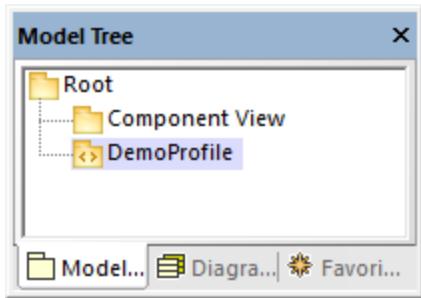
- Create a stereotype
- Create stereotype attributes (properties) that become tagged values when applied to an element
- Define a stereotype attribute as an enumeration
- Set a default value for a stereotype attribute
- Apply the stereotype to elements in the model.

The example is accompanied by a sample project file called **StereotypesDemo.ump**, available at the following path: **C:\Users\<username>\Documents\Altova\UModel2022\UModelExamples\Tutorial**. If you follow the instructions below literally, you will create a similar project.

## Create a new profile

As mentioned above, a stereotype must be owned by a profile; therefore, let's first create a profile.

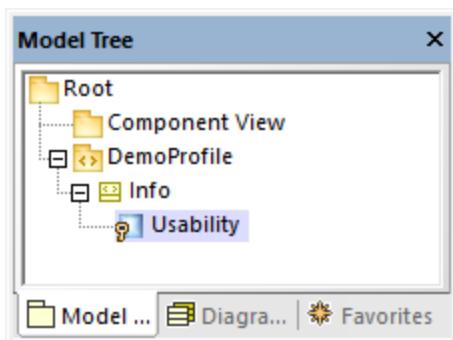
1. Create a new UModel project.
2. Right-click the "Root" package and add a new profile by selecting **New element | Profile** from the context menu.
3. Rename the new profile to "DemoProfile".



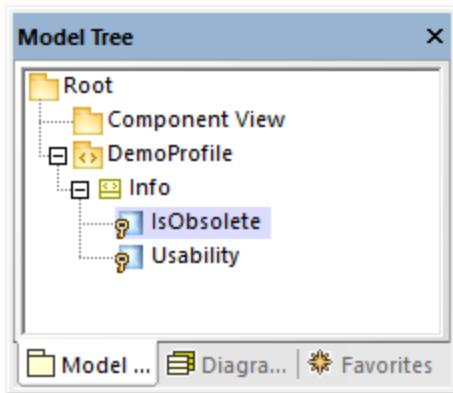
## Create a stereotype

For the scope of this tutorial, you will create a stereotype with two attributes: "Usability" and "IsObsolete". The "IsObsolete" attribute will be defined as an enumeration. The enumeration will consist of two values, "Yes" and "No", where "No" is the default value.

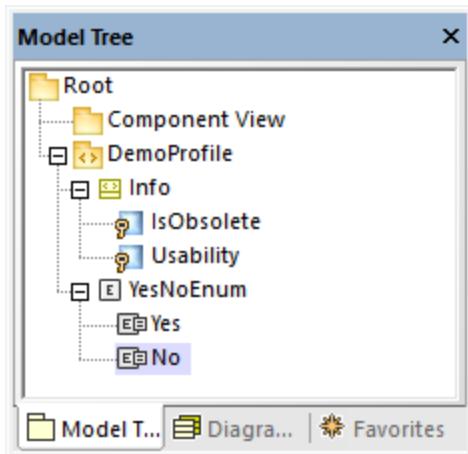
1. Right-click the profile and select **New element | Stereotype** from the context menu. A new stereotype has been added to the profile.
2. Rename the new stereotype to "Info".
3. Right-click the stereotype and select **New element | Property** from the context menu. This adds a new property.
4. Rename the new property to "Usability".



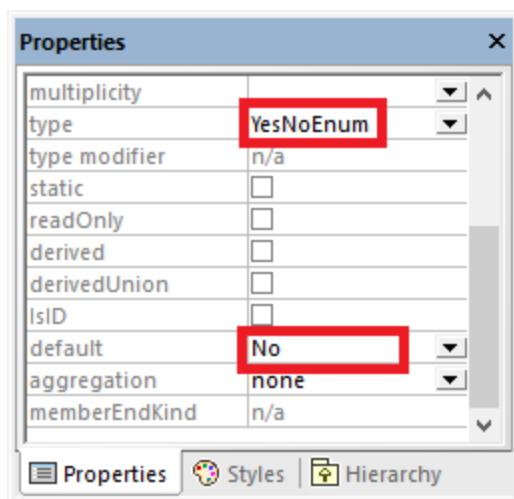
5. Repeat the steps above to create a new property called "IsObsolete".



6. Right-click the "DemoProfile" and select **New Element | Enumeration** from the context menu. Rename the enumeration to "YesNoEnum".
7. Right-click the enumeration and select **New Element | EnumerationLiteral** from the context menu. Rename the enumeration literal to "Yes".
8. Repeat the step above and create an enumeration literal called "No".



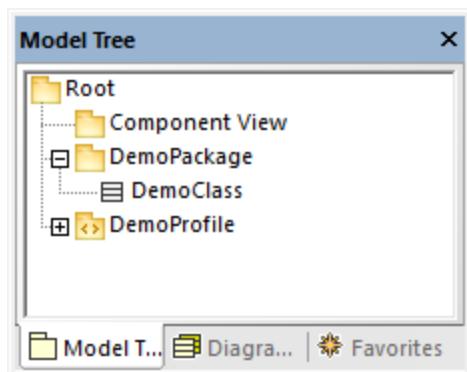
9. Click the "IsObsolete" property and change its type to `YesNoEnum`. Also, set the **default** property to "No"



## Create a new package

In order to illustrate how the custom stereotype can be used, let's create a simple package containing only one class.

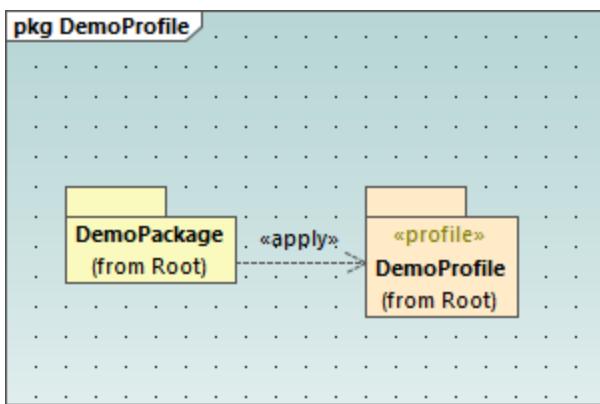
1. Right-click the "Root" package and add a new package by selecting **New element | Package** from the context menu.
2. Rename the new package to "DemoPackage".
3. Add a class to the package (in this example, "DemoClass").



## Apply the profile to a package

As you recall from Step 1, the stereotype was created inside a profile. In this step, we apply the profile to a package, so that the stereotype becomes "visible" to the package.

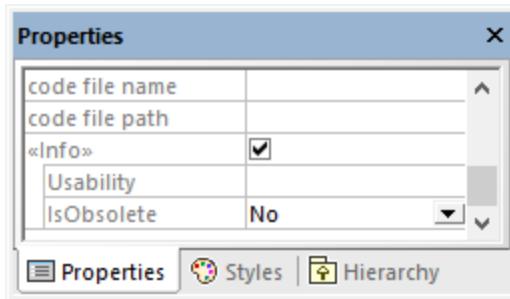
1. Right-click the "DemoProfile" in the Model Tree window and select **New diagram | Profile diagram** from the context menu.
2. Drag both the "DemoPackage" package and the "DemoProfile" profile from the Model Tree window into the diagram.
3. Click the **ProfileApplication** toolbar button, and draw a **ProfileApplication** relationship from the package to the profile.



## Apply the stereotype to classes

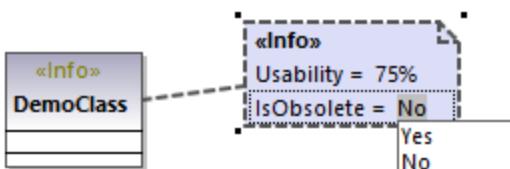
You can now apply the stereotype to a class.

1. Right-click the "DemoPackage" and select **New diagram | Class diagram** from the context menu.
2. Drag the class "DemoClass" onto the diagram.
3. Click the class and select the **«Info»** stereotype in the Properties window. Notice that the "IsObsolete" property is pre-filled with its default value.



4. Enter a value for the "Usability" property ("75%", in this example).

The class on the diagram now has a "Tagged values" section which displays the stereotype attributes and their values. You can change these values either from the Properties window, or directly from the diagram.



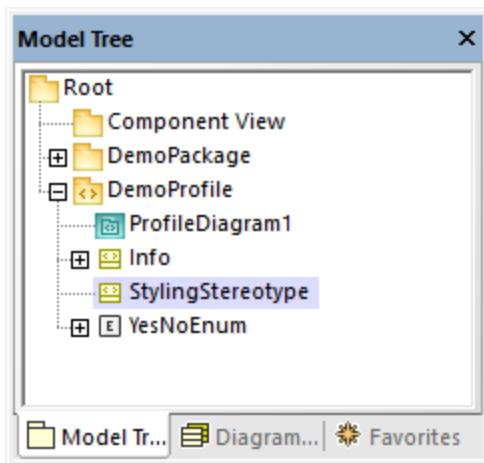
### 8.2.7.4 Example: Customizing Icons and Styles

This example shows you how to customize the appearance of a class in UModel with the help of stereotypes. After following this example, you will learn how to add custom icons to elements and change the style of all elements that use the same stereotype.

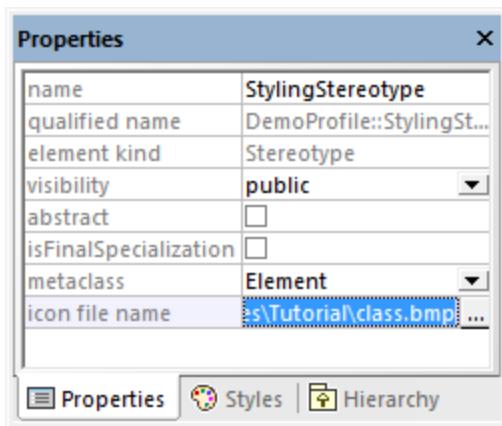
The class that will be customized in this example is in the **StereotypesDemo.ump** project, available at the following path: **C:\Users\<username>\Documents\Altova\UModel2022\UModel\Examples\Tutorial**. This is a simple demo project which includes a custom profile under which we will create the stereotype. For an example that shows you how to create profiles and stereotypes from scratch, see [Example: Creating and Applying Stereotypes](#) 409.

Let's first create the stereotype to be used for styling:

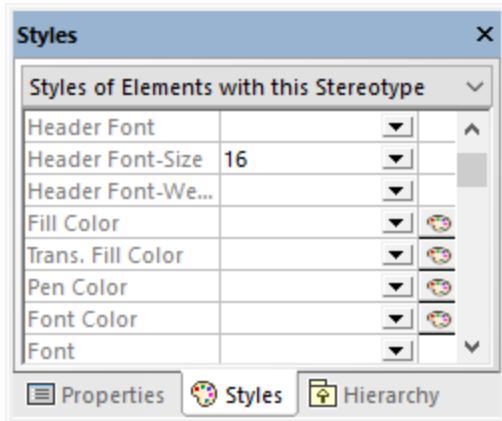
1. Open the **StereotypesDemo.ump** project.
2. Right-click the "DemoProfile" profile in the model tree, and select **New Element | Stereotype** from the context menu.
3. Rename the stereotype to "StylingStereotype".



To add a custom image to the stereotype, click the stereotype, and then click the Ellipsis button next to icon file name property in the Properties window. Select the following sample image: **C:\Users\<username>\Documents\Altova\UModel2022\UModel\Examples\Tutorial\class.bmp**.

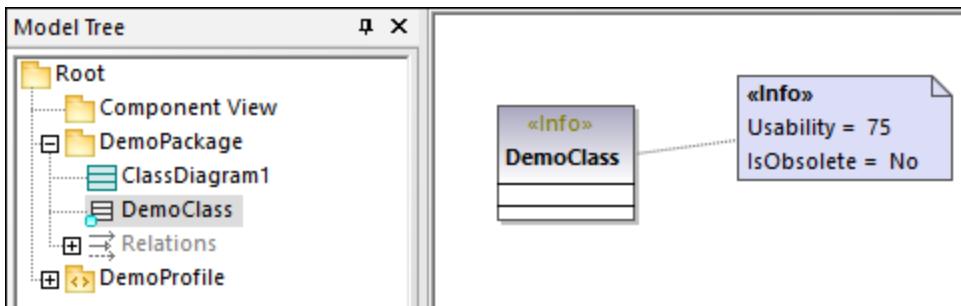


Next, click the **Styles** tab of the Properties window. Select **Styles of Elements with this Stereotype** from the top list, and change the **Header Font Size** property to "16".

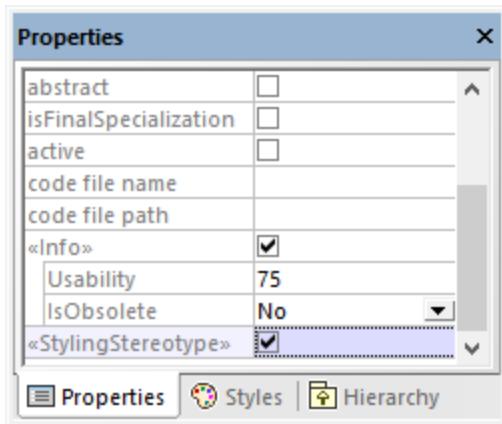


Finally, apply the stereotype to a class.

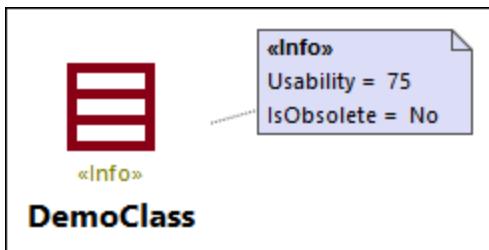
1. Open the class diagram "ClassDiagram1". You will find this diagram under the "DemoPackage" in the Model Tree view.



2. Click the "DemoClass" class, and then select the **«StylingStereotype»** check box in the Properties window.

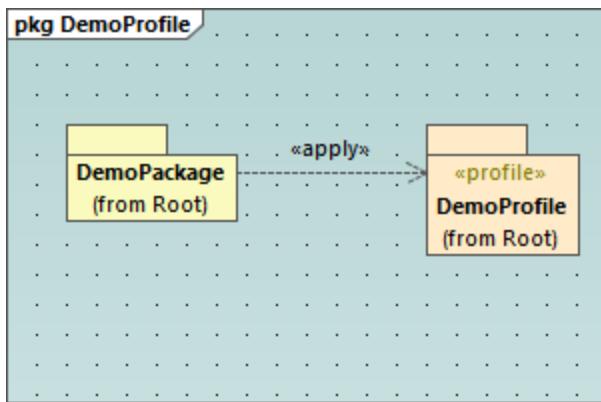


The appearance of the class on the diagram is now changed according to the applied stereotype:



## Remarks

The demo project contains a profile diagram, "ProfileDiagram1". In this diagram, notice that the "DemoProfile" is applied to the "DemoPackage" with a **ProfileApplication** relationship. This makes the stereotype available to the package, see also [Creating and Applying Custom Profiles](#) 405.



You have now learned how to change the appearance of elements using stereotypes. You can use the same technique in other projects. Just keep in mind that the profile where you create the stereotype must be applied to the target package, as shown above.

## 8.3 Additional Diagrams

The additional diagram kinds supported by **UModel Basic Edition** are as follows:



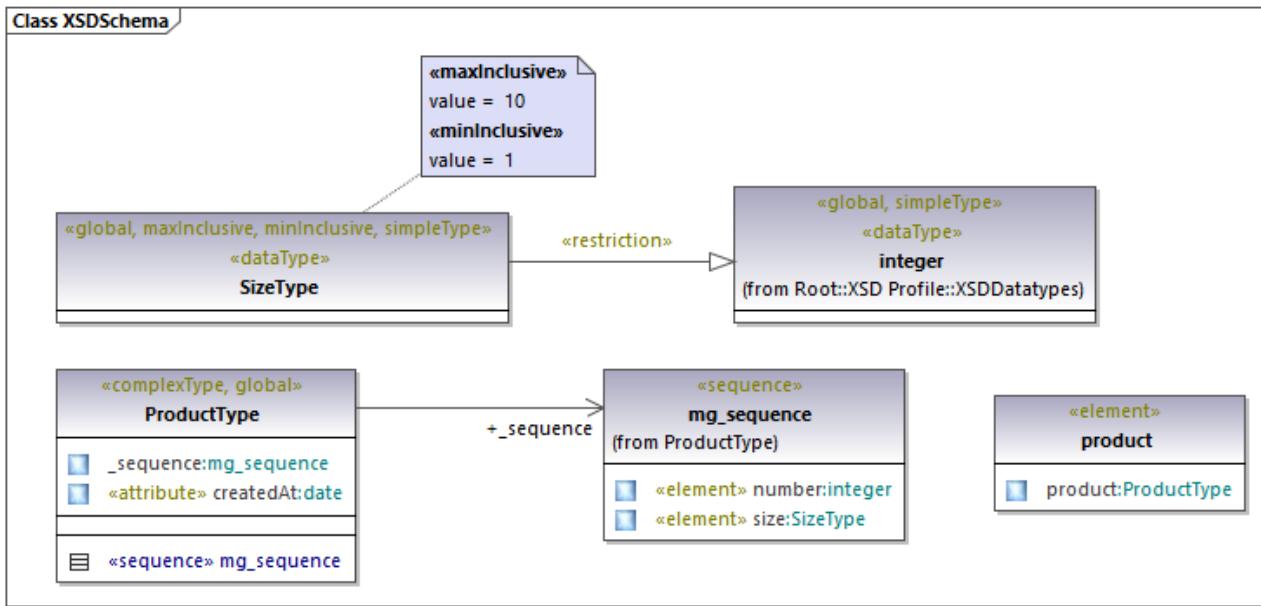
### 8.3.1 XML Schema Diagrams

Altova website: [XML Schemas in UML](#)

UModel supports the import and generation of W3C XML schemas as well as their forward and reverse engineering. In case of XML Schemas, "forward and reverse engineering" means that you can import a schema (or multiple schemas from a directory) into UModel, view or modify the model, and write the changes back to the schema file. When you synchronize data from the model to a schema file, the schema file is always overwritten by the model.

**Note:** The XML Schema must be valid before it can be imported into UModel. XML Schemas are not validated when you create or import them in UModel, or when you run a project syntax check. Nevertheless, UModel checks whether the XML schema is well-formed when importing it.

XML Schema diagrams display schema components in UML notation. For example, simple types are shown in UModel as data types with the «simpleType» stereotype. Complex types are shown as classes with the «complexType» stereotype. Various schema details are represented as [Tagged Values](#), while schema annotations are represented as comments. For a mapping table that illustrates how all the XML schema components map to UModel elements, see [XML Schema Mappings](#).



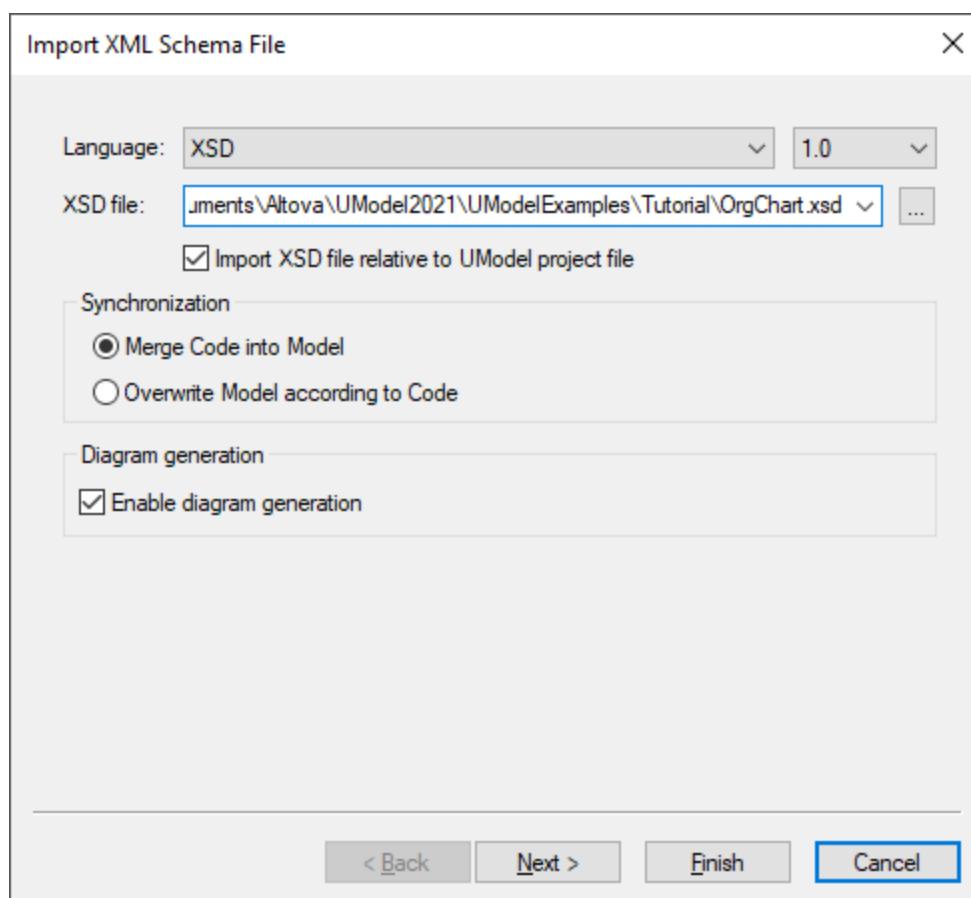
Example XML Schema diagram

### 8.3.1.1 Importing XML Schemas

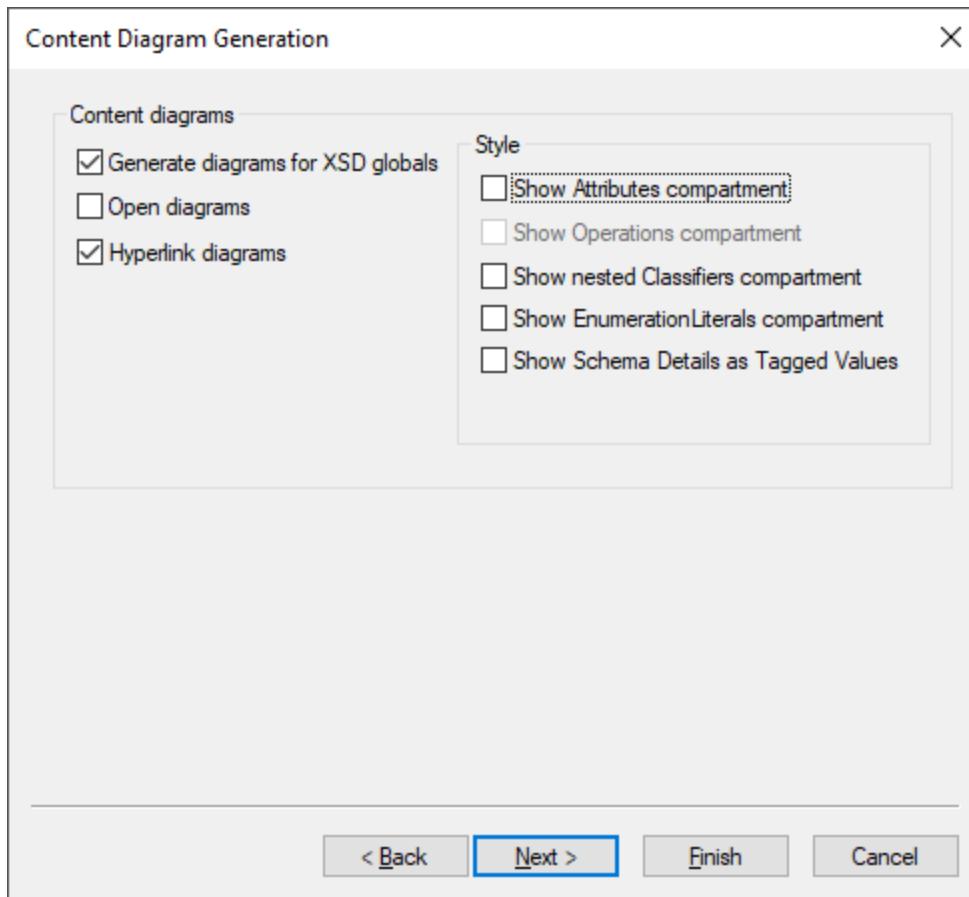
You can import either a single schema file into UModel, or all schemas from a directory. If a schema includes or imports other schemas, these are imported into the model as well.

#### To import a single XML Schema:

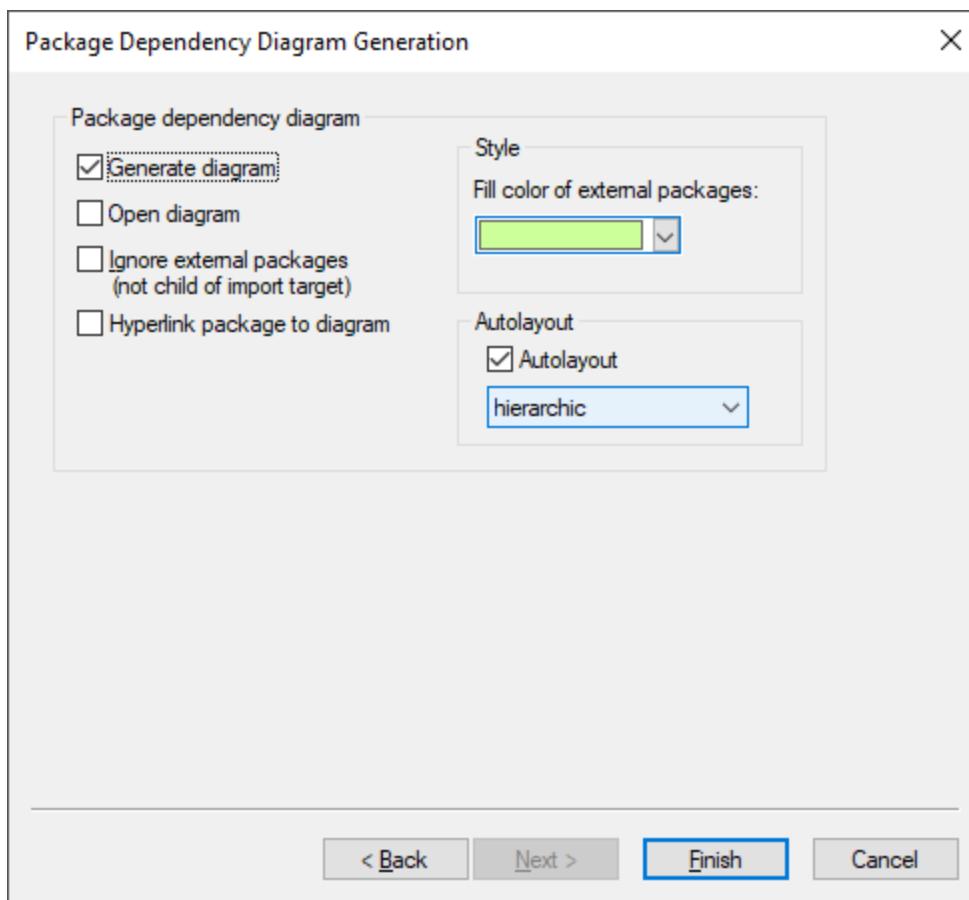
1. Select the menu command **Project | Import XML Schema file**.
2. Click **Browse** and select the source schema to import. For the scope of this example, you can use the following schema: **C**:  
`\Users\<username>\Documents\Altova\UModel2022\UModelExamples\Tutorial\OrgChart.xsd.`



3. To generate diagrams from the schema, make sure that the **Enable diagram generation** check box is selected and click **Next**.

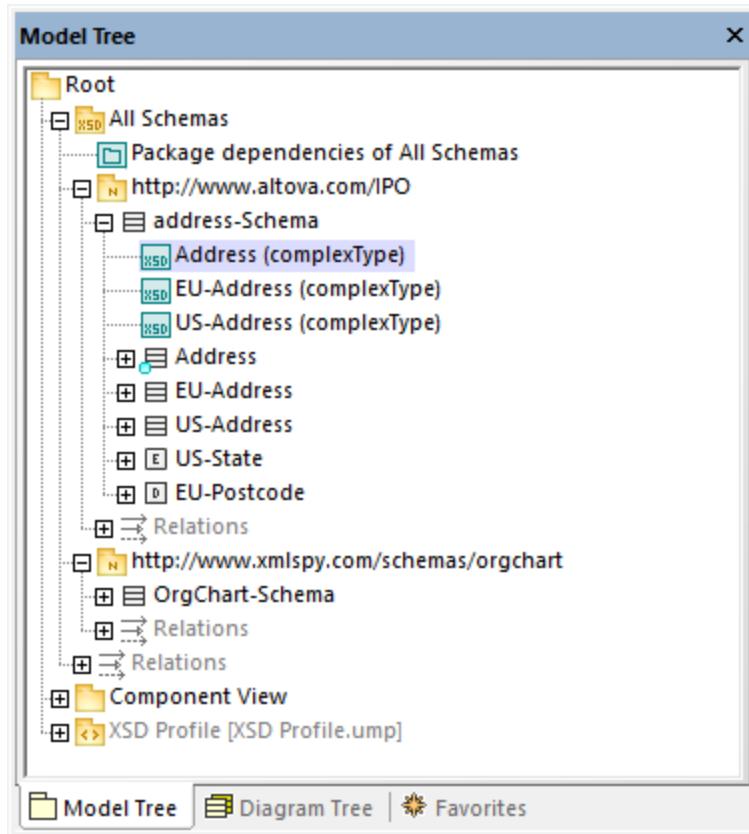


4. To create a separate diagram for each global component in the schema like in this example, select the **Generate diagrams for XSD globals** option. To open all generated diagrams after import, select **Open diagrams**. Options from the "Style" group let you define the compartments that appear by default in diagrams for each schema component. The **Show schema details as tagged values** (141) option displays the schema details as **Tagged Values**.
5. Click **Next**. To generate a Package dependency diagram like the one in this example, select the **Generate Diagram** check box.



6. Click **Finish**.

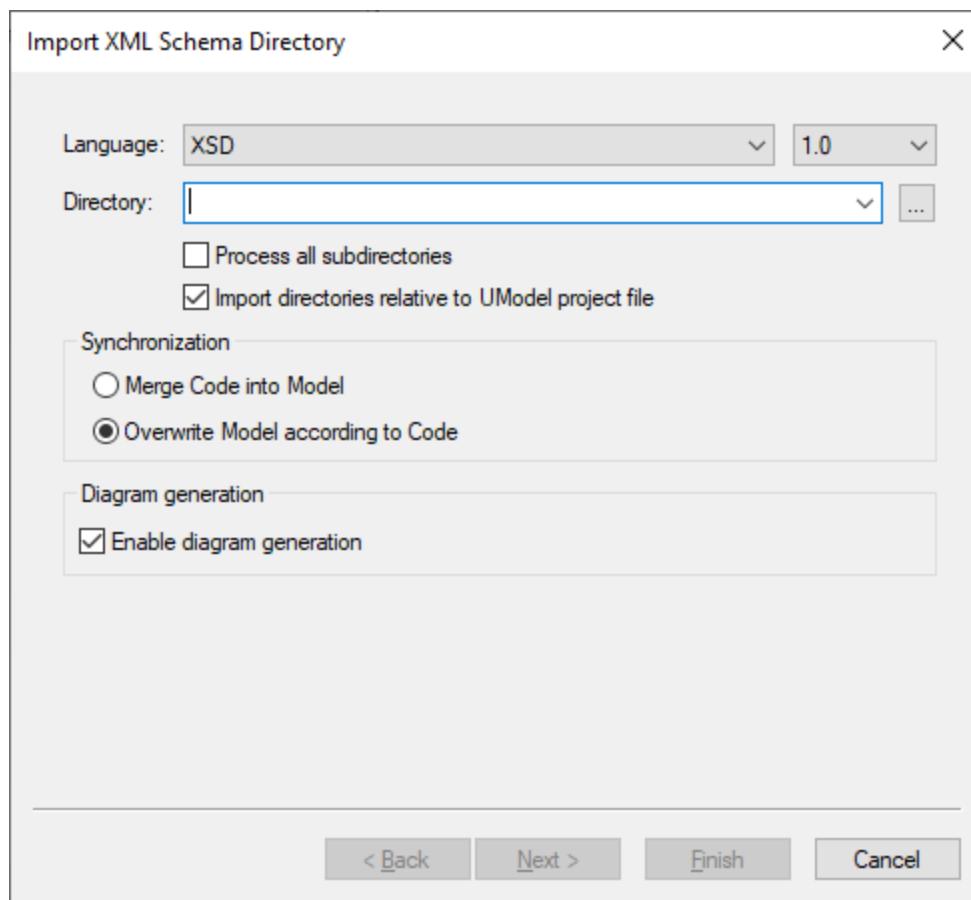
Once UModel completes importing the schema, a new package called **All Schemas** is created and set automatically as the "XSD Namespace Root". The **OrgChart.xsd** schema used in this example imports types from another namespace, more specifically, from the **ipo.xsd** schema. Consequently, both schemas appear in the Model Tree window after import, under their respective namespaces:



If you have selected the **Generate diagrams for XSD globals** check box, all XSD global components generate an XML Schema diagram, and the diagrams appear under the respective namespace packages, like the "Address (complexType)" diagram in the image above.

#### To import multiple XML Schemas:

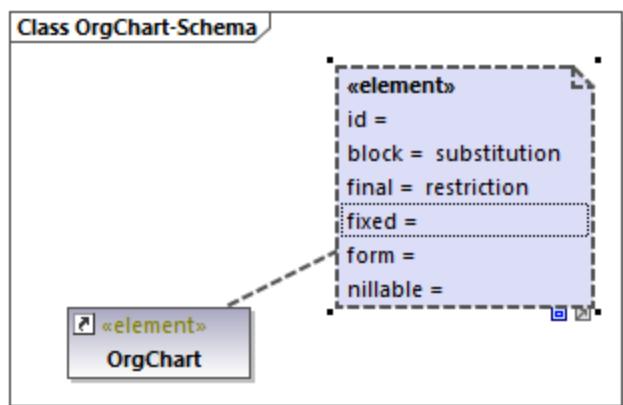
1. Select the menu command **Project | Import XML Schema directory**.



- To import schemas from all subdirectories of the selected directory, select the **Process all subdirectories** check box. The rest of the import process is the same as described above for a single XML schema.

### Changing the display of tagged values

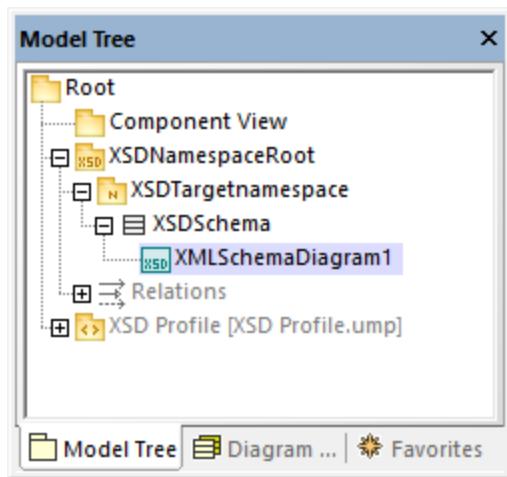
After importing an XML schema, certain schema details may appear as tagged values on the diagram, if you have selected the **Show Schema Details as Tagged Values** option during the import.



You can configure whether such details are to be shown or hidden from the diagram. To do this, right-click the element and select **Tagged Values | <option>** from the context menu. You can configure the display of tagged values not only individually for each element, but also globally at project level. For more information, see [Showing or Hiding Tagged Values](#) <sup>144</sup>.

### 8.3.1.2 Modeling XML Schemas

New XML Schema projects in UModel have the structure illustrated below. This structure is created automatically the first time when you add an XML Schema diagram to a new UModel project.



The "Root" and "Component View" packages are common to any UModel Project and cannot be deleted. "Root" is the topmost level under which any other packages are added, and "Component View" is used for code engineering (in this case, importing or generating schema files).

The "XSDNamespaceRoot" package includes all the namespaces used by your schema(s). To turn a package into an XSD Namespace Root, right-click it and select **Code Engineering | Set as XSD Namespace Root** from the context menu. If you import an existing XML schema into the project, this package is called "All schemas" by default.

The "XSDTargetnamespace" package is an XML Schema namespace. Multiple such namespaces may exist under the same XSD Namespace Root. To turn a package into a namespace, first select the package, and then select the «namespace» property (stereotype) in the Properties window.

"XSDSchema" is a schema, or, in UML terms, a class with the «schema» property (stereotype) selected in the Properties window.

**XMLSchemaDiagram1** is the actual diagram that describes the schema's model. You can create XML Schema diagrams under an XSD Namespace Root, under an XML Schema Namespace, or under an XML Schema. In the example project illustrated above, the diagram is created under the XML schema.

The **XSD Profile** enables all the types and structures required to work with XML Schema in the project. If your project does not have this profile, you will be prompted to include it whenever you create a new XML Schema diagram. You can also add the XSD profile to a project explicitly, see [Applying UModel Profiles](#) <sup>154</sup>.

## Creating XML Schema diagrams

To create a new XML schema diagram:

1. Do one of the following:
  - a. Right-click a package in the [Model Tree Window](#)<sup>79</sup> and select **XML Schema Diagram** from the context menu.
  - b. Right-click "Diagrams" or "XML Schema Diagrams" in the [Diagram Tree Window](#)<sup>83</sup> and select **New Diagram | XML Schema diagram** from the context menu. A dialog box opens asking you to select the owner of the diagram. Select a package where the diagram should be stored, and click **OK**.
2. If the current UModel project does not include the XSD profile, a dialog box opens asking you to include it. Click **OK** to include the XSD profile into the current project, see also [Applying UModel Profiles](#)<sup>154</sup>.

## Adding new XML Schema elements

To add XML schema elements to a diagram:

- Click a specific toolbar button, and then click inside the XML Schema diagram.



To insert multiple elements of the same type, hold down the **Ctrl** key and click multiple times in the diagram.

As stated above, XML Schema diagrams can be created at various levels in the project's structure. If the diagram is at a level which does not allow placing a particular element, certain toolbar buttons are not meaningful and they show a tooltip with information instead of adding the element.

The table below lists all the toolbar buttons and their purpose.

	<b>XSD Target Namespace</b>	Adds an XSD target namespace. Clicking this button is meaningful if the diagram was created directly under an XSD Namespace Root.
	<b>XSD Schema</b>	Adds an XML Schema Definition (XSD). Clicking this button is meaningful if the diagram was created under an XSD target namespace.
	<b>Element (global)</b>	Adds a global element to the diagram. When you add an element, a property with the same name as the element is automatically generated in the attributes compartment. Set the property type to set the element's type.
	<b>Group</b>	Adds a named model group to the diagram.
	<b>Complex Type</b>	Adds a global complex type to the diagram. In UML terms, this is a class that has the «global» and «complexType»

		stereotypes applied.
 CS	<b>Complex Type with Simple Content</b>	Adds a global complex type with simple content. In UML terms, this is a data type that has the «global», «complexType», and «simpleContent» stereotypes applied.
 S	<b>Simple Type</b>	Adds a global simple type.
 SL	<b>List</b>	Adds a list type.
 SU	<b>Union</b>	Adds a union type.
 SE	<b>Enumeration</b>	Adds an enumeration.
 A	<b>Attribute</b>	Adds an attribute.
 AG	<b>Attribute group</b>	Adds an attribute group.
 N	<b>Notation</b>	Adds a notation type.
 IM	<b>Import</b>	Adds an import relationship.
 IN	<b>Include</b>	Adds an include relationship.
 RE	<b>Redefine</b>	Adds a redefine relationship.
 R	<b>Restriction</b>	Adds a restriction relationship.
 E	<b>Extension</b>	Adds an extension relationship.
 S	<b>Substitution</b>	Adds a substitution relationship.
 C	<b>Comment</b>	Adds a comment. Comments are converted to annotations when you generate the schema file from the model. You can specify the annotation type by selecting the required stereotype from the Properties window.
 N	<b>Note</b>	Adds an explanatory note.
 NL	<b>Note link</b>	Links a note to some other element on the diagram.

For step-by-step schema modeling instructions, see [Example: Create and Generate an XML Schema](#) 425.

### 8.3.1.3 Example: Create and Generate an XML Schema

This example shows you how to model a new XML Schema with UModel, step by step. After modeling the schema visually using UML, you will generate the schema file. More specifically, you will learn how to create and generate the **product.xsd** schema listed below.

```
<?xml version="1.0" encoding="UTF-8"?>
<xss:schema targetNamespace="http://www.altova.com/umodel"
xmlns:xss="http://www.w3.org/2001/XMLSchema" xmlns:prod="http://www.altova.com/umodel">
  <xss:simpleType name="SizeType">
```

```

<xs:restriction base="xs:integer">
  <xs:maxInclusive value="10"/>
  <xs:minInclusive value="1"/>
</xs:restriction>
</xs:simpleType>
<xs:complexType name="ProductType">
  <xs:sequence>
    <xs:element name="number" type="xs:integer">
      </xs:element>
    <xs:element name="size" type="prod:SizeType">
      </xs:element>
    </xs:sequence>
    <xs:attribute name="createdAt" type="xs:date">
      </xs:attribute>
  </xs:complexType>
  <xs:element name="product" type="prod:ProductType">
    </xs:element>
</xs:schema>

```

*product.xsd*

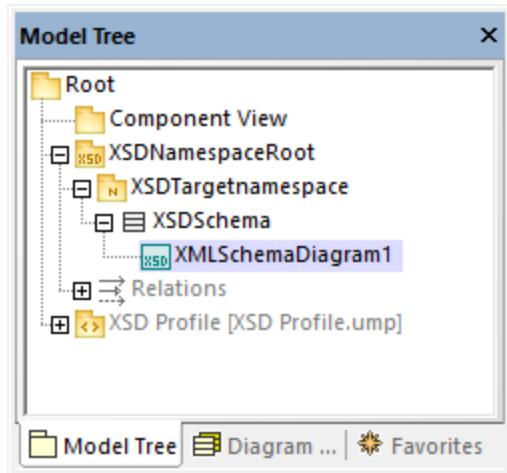
As shown above, the **product.xsd** schema has two namespace declarations:

1. The default XML Schema namespace <http://www.w3.org/2001/XMLSchema> mapped to the "xs" prefix.
2. The secondary namespace <http://www.altova.com/umodel> mapped to the "prod" prefix, which is also the target namespace.

Also, the XML schema has a global product element, a complex type `ProductType` and a simple type `SizeType`.

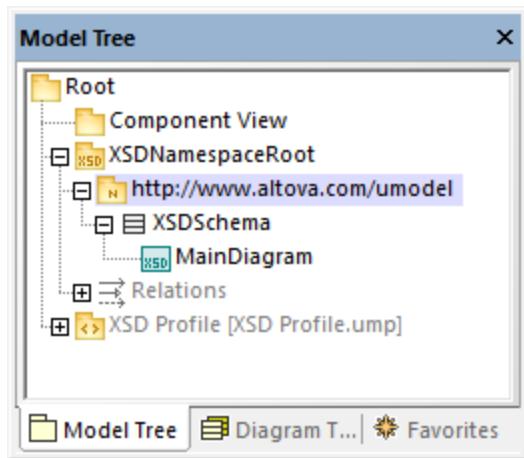
## Declaring namespaces and file encoding

To proceed, create a new UModel project. Right-click the **Root** package, and select **New Diagram | XML Schema Diagram** from the context menu. When prompted to include the UModel XSD Profile, click **OK**.



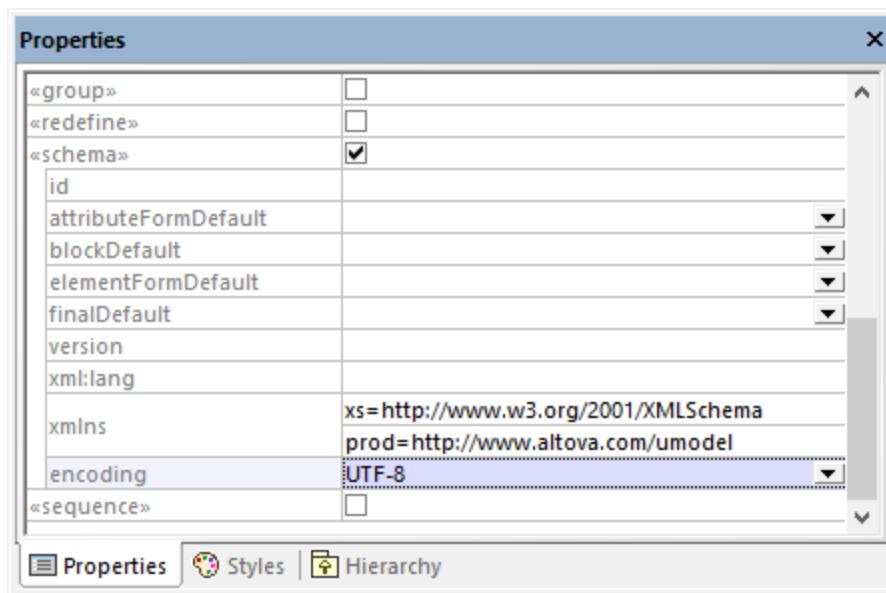
In the [Model Tree Window](#)<sup>79</sup>, rename "XMLSchemaDiagram1" to "MainDiagram". This is the diagram where most schema components will be created, except for namespace declarations.

Next, rename "XSDTargetNamespace" to "http://www.altova.com/umodel" (recall that this is the required target namespace). This declares the target namespace of the new schema.



The two "xmlns" namespaces and the UTF-8 encoding can be set as follows:

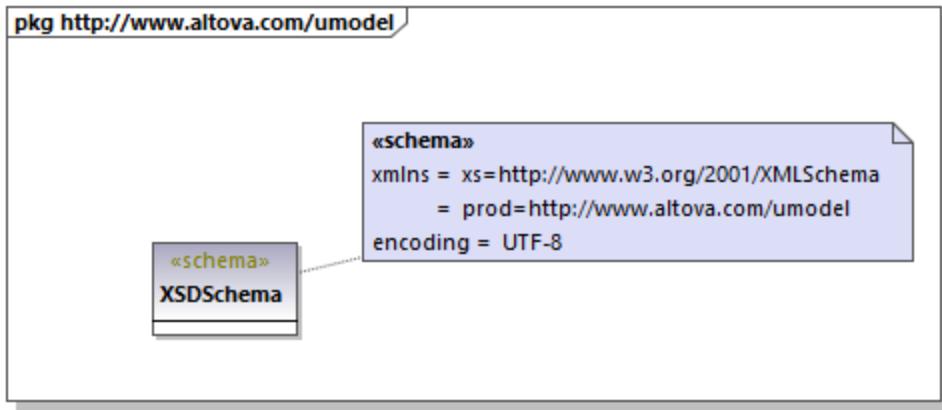
1. Select the **XSDSchema** schema in the Model Tree.
2. In the Properties window, right-click the **xmlns** property and select **Add Tagged Value | xmlns**.
3. Edit the **xmlns** and **encoding** properties as shown below.



Optionally, you can quickly generate a new XML Schema diagram at namespace level that presents the same information visually, as follows:

1. In the Model Tree, right-click the namespace "http://www.altova.com/umodel" and select **New Diagram | XML Schema diagram** from the context menu.

2. When a message box with the following text appears: "Do you want to add the 'XML Schema Diagram' to a new 'XSD Schema?'?", click **No**.
3. Drag the XML Schema from the Model Tree into the diagram.

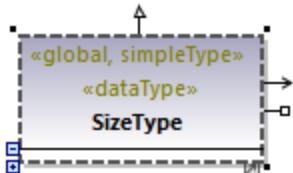


As shown above, the namespace and encoding are stored as [Tagged Values](#) <sup>141</sup> and can be edited from the diagram window as well.

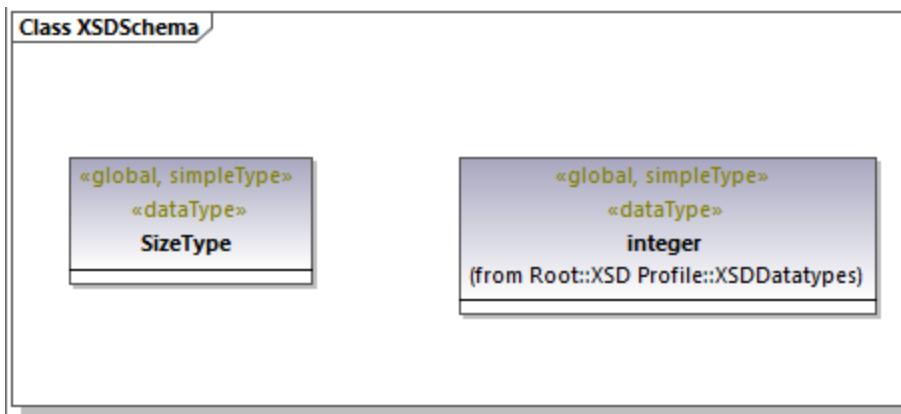
## Add a simple type

The following steps create the `SizeType` simple type to the XML schema. This is a type that restricts the base `xs:integer` type; therefore, we will add the base type to the diagram as well, and create a restriction relationship.

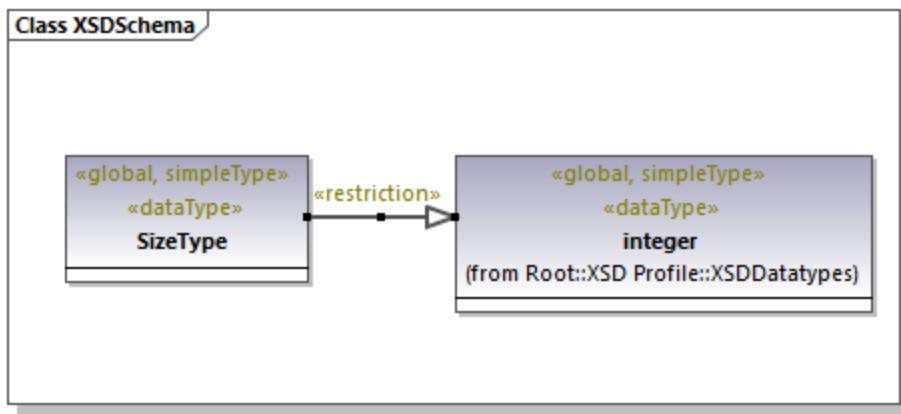
1. Double-click the **MainDiagram** in the Model Tree to open it.
2. Click the **XSD Simple Type** toolbar button, and then click inside the diagram.
3. Rename the newly added simple type to `SizeType`.



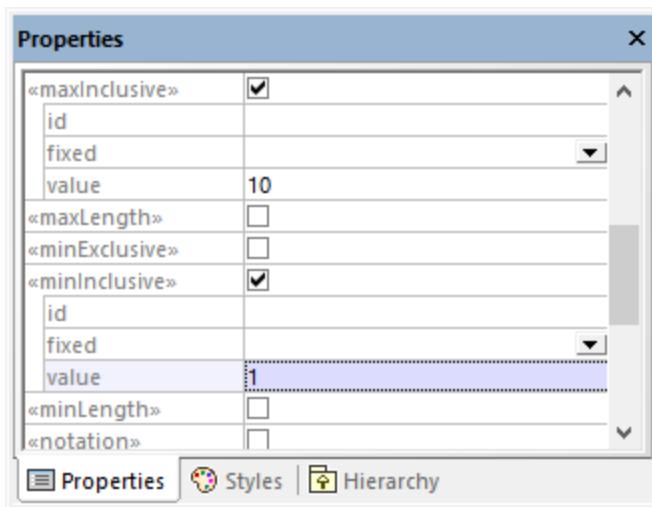
4. Click inside the Model Tree and press **Ctrl+F**. The Find dialog box appears. Start typing "integer" and locate the `integer` type from the "XSDDataTypes" package of the "XSD Profile".
5. Drag the `integer` type into the diagram.



6. Click the **Restriction** toolbar button and drag the cursor from `SizeType` to `integer`. This creates the restriction relationship; see also [Creating Relationships](#).



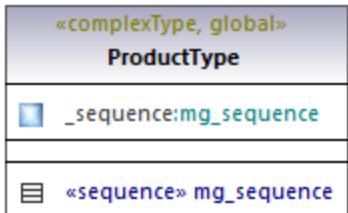
7. To define the `minInclusive` and `maxInclusive` values, select the simple type and edit the properties with the same name in the Properties window.



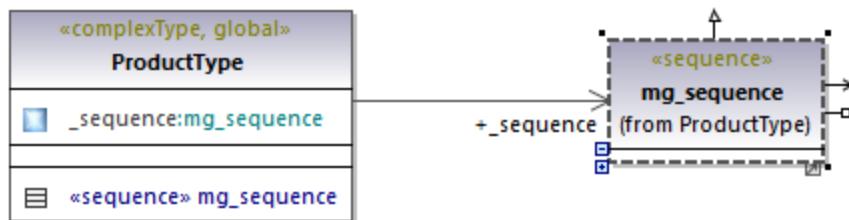
## Add a complex type

The following steps add the `ProductType` complex type to the XML schema. All these steps take place in the **MainDiagram** as well.

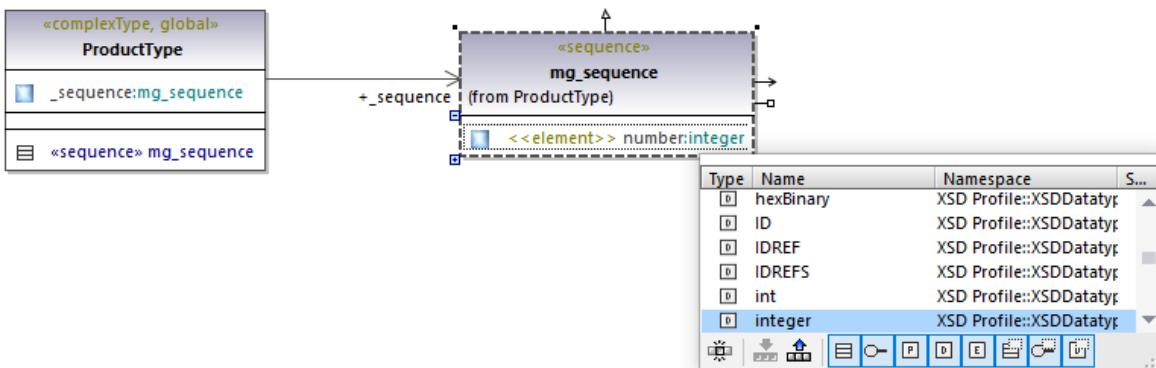
1. Click the **XSD Complex Type**  toolbar button, and then click inside the diagram.
2. Rename the complex type to `ProductType`.
3. Right-click the complex type and select **New | XSD Sequence** from the context menu.



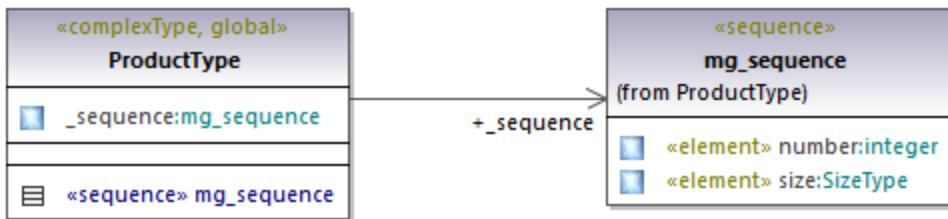
4. Drag the «sequence» class away from the complex type and into the diagram.



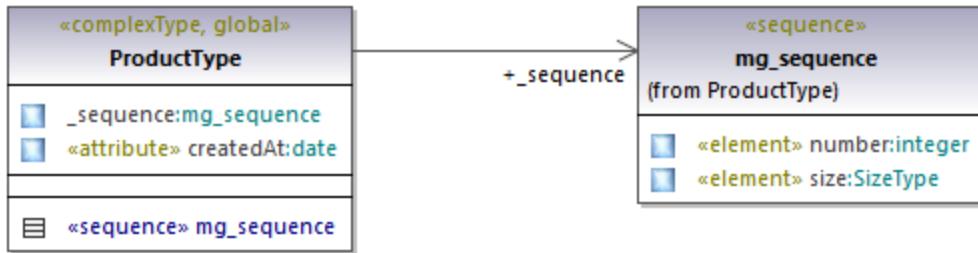
5. Right-click the sequence and select **New | XSD Element (local)**.
6. Change the element's name to **number** and set the type to `integer`. The `integer` type is a base XML Schema type from the XSD Profile. For instructions about setting an element's type, see [Type Autocompletion in Classes](#) <sup>(127)</sup>.



7. Using the same steps as above, create the element `size` of type `SizeType`. Note that `SizeType` is the simple type created previously.



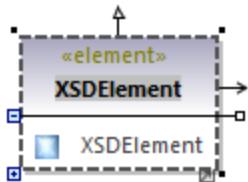
8. Right-click the complex type on the diagram and select New | XSD Attribute (local) from the context window.
9. Change the attribute's name to **createdAt** and the type to `date`.



## Add an element

Now that all the required types of the schema have been defined, you can add a product element of type `ProductType`, as follows:

1. Click the **XSD Element (global)** toolbar button, and then click inside the diagram. Notice that a class with the `<<element>>` stereotype and a single property is added.

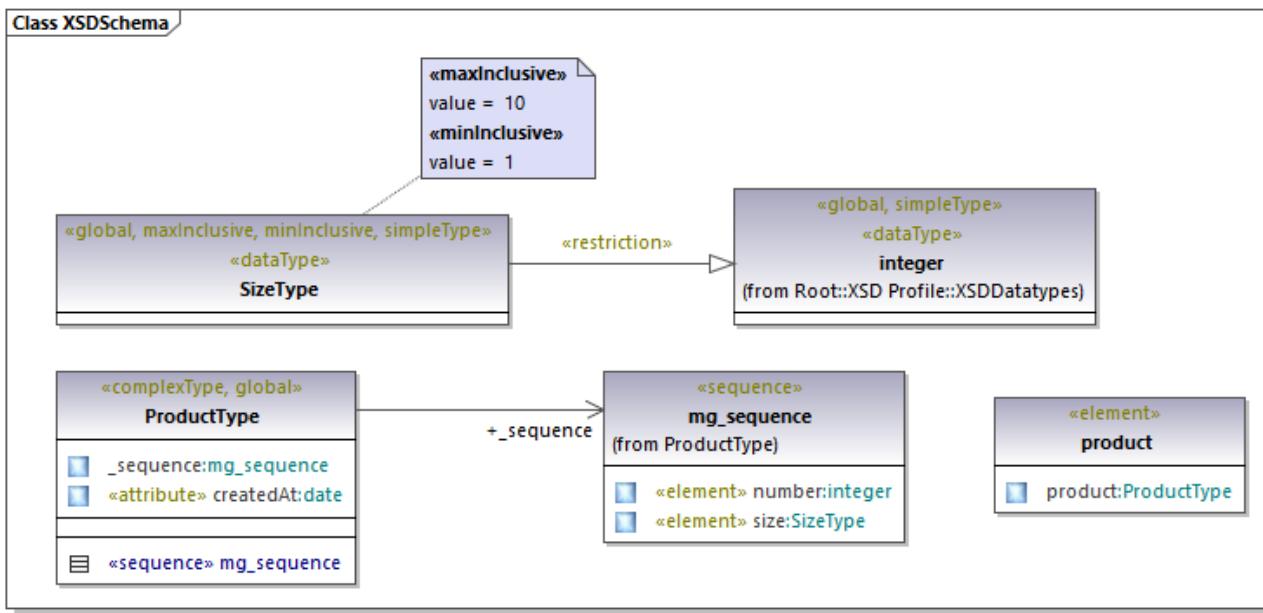


2. Rename the property to **product** and change its type to `ProductType`.



## Completed design

The steps above conclude the design part of the schema. By now, your full schema design should look as follows:

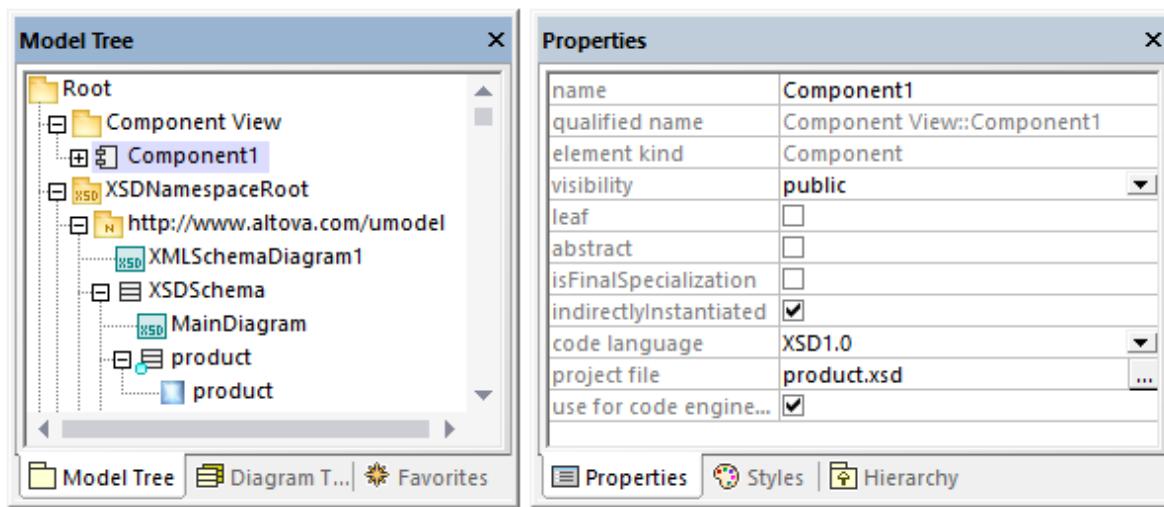


## Enable code engineering

To make it possible to generate a schema file from the model, let's now add a code engineering component that provides the schema generation details. The code engineering component is similar to other UModel project kinds, see also [Adding a Code Engineering Component](#).

Right-click the "Component View" package in the Model Tree and add a new element of type **Component**. Make sure to change the component's properties as shown below:

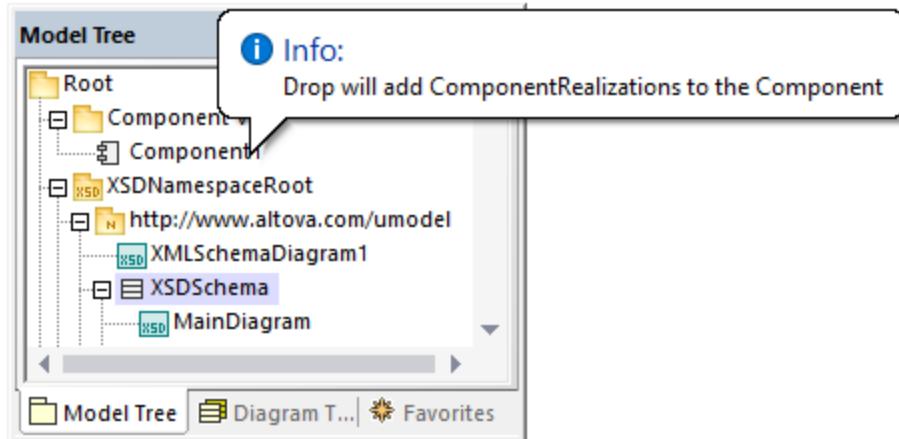
1. The **use for code engineering** property must be enabled.
2. The **code language** property of the code engineering component must be set to "XSD 1.0".
3. The **project file** property of the code engineering component must point to the schema file that is to be generated (in this example, **product.xsd**).



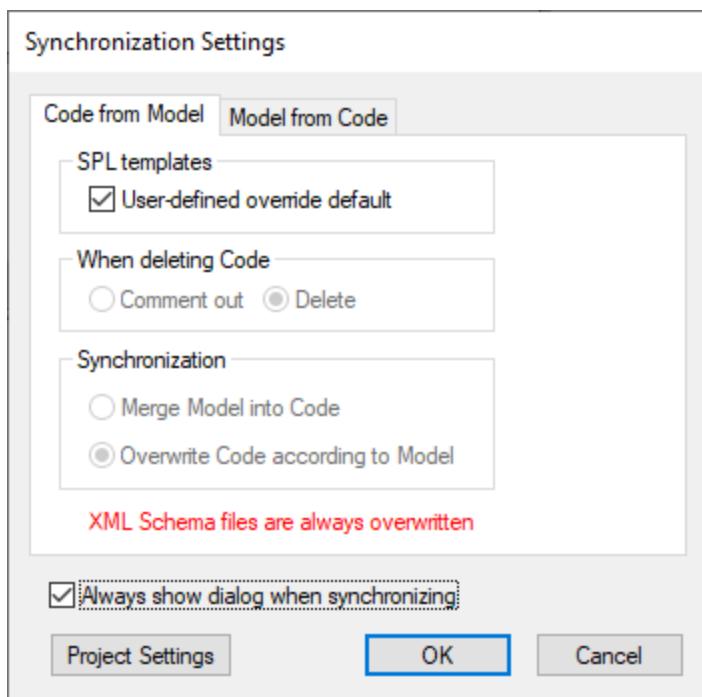
**Note:** If a **project file** property is missing, enter **product.xsd** in the **directory** property and press **Enter**. A message box should now appear asking you to refer to a project file instead. Click **Yes** to confirm.

Finally, the XML Schema must be realized by the code engineering component, as described in [Adding a Code Engineering Component](#)<sup>165</sup>. For the scope of this example, the quickest way to create the **ComponentRealization** relationship is as follows:

- In the Model Tree, drag the **XSDSchema** schema over the code engineering component (**Component1**) and drop it when a tooltip appears such as the one below:



You can now generate the schema file. To do this, either press **F12** or select the **Project | Overwrite Program Code from UModel project** menu command. Note that merging is not supported in case of XML Schemas; therefore, the dialog box shows a message in red to state this fact.



The new XML schema will be generated in the same folder as your UModel project.

## 9 XMI - XML Metadata Interchange

 [Altova website: Exchanging UModel projects using XMI](#)

You can export UModel projects to XML Metadata Interchange (XMI) files, and import XMI files as UModel projects. This provides interoperability with other UML tools that support XMI. The supported XMI versions are as follows:

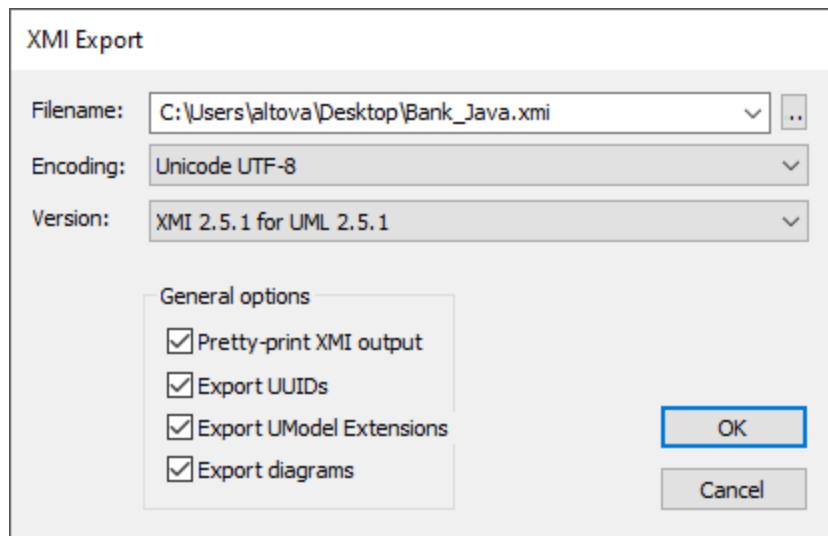
- XMI 2.1 for UML 2.0
- XMI 2.1 for UML 2.1.2
- XMI 2.1 for UML 2.2
- XMI 2.1 for UML 2.3
- XMI 2.4.1 for UML 2.4.1
- XMI 2.4.1 for UML 2.5
- XMI 2.5.1 for UML 2.5.1

**To import an XMI file into UModel:**

- On the **File** menu, click **Import from XMI File**.

**To export a UModel project to an XMI file:**

- On the **File** menu, click **Export to XMI File**.



Notes:

- During the export process, all included files, even those defined as [include by reference](#)<sup>160</sup>, are exported.
- If you intend to re-import generated XMI code into UModel, make sure that you select the **Export UModel Extensions** check box.

The sections below describe options available when exporting projects to XMI.

### Pretty-print XMI output

If you select this option, the XMI file will be generated with XML tag indentation and carriage returns.

### Export UIDs

XMI defines three versions of element identification: IDs, UUIDs and labels.

- IDs are unique within the XMI document, and are supported by most UML tools. UModel exports these type of IDs by default, i.e. none of the check boxes need activated.
- UUID are Universally Unique Identifiers, and provide a mechanism to assign each element a global unique identification, GUID. These IDs are globally unique, i.e. they are not restricted to the specific XMI document. UUIDs are generated by selecting the "Export UUIDs" check box.
- UUIDs are stored in the standard canonical UUID/GUID format (e.g "6B29FC40-CA47-1067-B31D-00DD010662DA", "550e8400-e29b-41d4-a716-446655440000",...)
- Labels are not supported by UModel.

**Note:** The XMI import process automatically supports both types of IDs.

### Export UModel Extensions

XMI defines an "extension mechanism" which allows each application to export its tool-specific extensions to the UML specification. Other UML tools will, however, only be able to import the standard UML data (ignoring the UModel extensions). This UModel extension data will be available when importing into UModel.

Data such as the file names of classes, or element colors, are not part of the UML specification and thus have to be deleted in XMI, or be saved in "Extensions". If they have been exported as extensions and re-imported, all file names and colors will be imported as defined. If extensions are not used for the export process, then these UModel-specific data will be lost.

When importing an XMI document, the format is automatically detected and the model generated.

### Export diagrams

Exports UModel diagrams as "Extensions" in the XMI file. The option **Export UModel Extensions** must be selected before you can save the diagrams as extensions.

## 10 Source Control

The source control support in UModel is available through the Microsoft Source Control Plug-in API (formerly known as the MSSCCI API), versions 1.1, 1.2 and 1.3. This enables you to run source control commands such as "Check in" or "Check out" directly from UModel to virtually any source control system that lets native or third-party clients connect to it through the Microsoft Source Control Plug-in API.

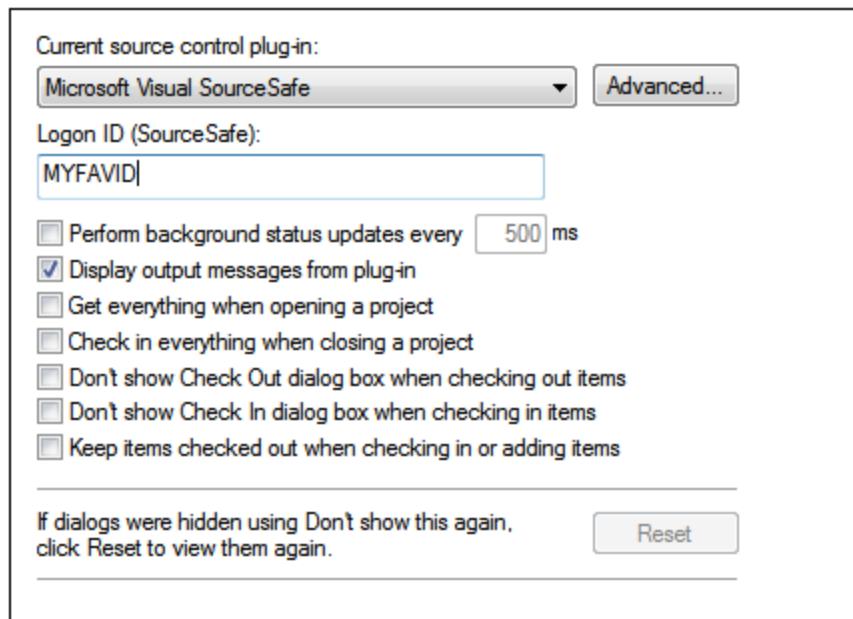
You can use as your source control provider any commercial or non-commercial plug-in that supports the Microsoft Source Control Plug-in API, and can connect to a compatible version control system. For the list of source control systems and plug-ins tested by Altova, see [Supported Source Control Systems](#)<sup>440</sup>.

### Installing and configuring the source control provider

To view the source control providers available on your system, do the following:

1. On the **Tools** menu, click **Options**.
2. Click the **Source Control** tab.

Any source control plug-ins compatible with the Microsoft Source Code Control Plug-in API are displayed in the **Current source control plug-in** drop-down list.



If a compatible plug-in cannot be found on your system, the following message is displayed:

"Registration of installed source control providers could not be found or is incomplete."

Some source control systems might not install the source control plug-in automatically, in which case you will need to install it separately. For further instructions, refer to the documentation of the respective source control system. A plug-in (provider) compatible with the Microsoft Source Code Control Plug-in API is expected to be registered under the following registry entry on your operating system:

HKEY\_LOCAL\_MACHINE\SOFTWARE\SourceCodeControlProvider\InstalledSCCPProviders

Upon correct installation, the plug-in becomes available automatically in the list of plug-ins available to UModel.

## Accessing the source control commands

The commands related to source control are available in the **Project | Source Control** menu.

## Resource / Speed issues

Very large source control databases might be introducing a speed/resource penalty when automatically performing background status updates.

You might be able to speed up your system by disabling (or increasing the interval of) the **Perform background status updates every ... seconds** option in the **Source Control** tab accessed through **Tools | Options**.

**Note:** The **64-bit** version of your Altova application automatically supports any of the supported 32-bit source control programs listed in this documentation. When using a 64-bit Altova application with a 32-bit source control program, the **Perform background status updates every ... seconds** option is automatically grayed-out and cannot be selected.

## Differencing with Altova DiffDog

You can configure many source control systems (including Git and TortoiseSVN) so that they use Altova DiffDog as their differencing tool. For more information about DiffDog, see <https://www.altova.com/difffdog>. For DiffDog documentation, see <https://www.altova.com/documentation.html>.

## 10.1 Setting Up Source Control

The mechanism for setting up source control and placing files in a UModel project under source control is as follows:

1. If this hasn't been done already, install the source control system (see [Supported Source Control Systems](#)<sup>440</sup>) and set up the source control database (repository) to which you wish to save your work.
2. Create a local workspace folder that will contain the working files that you wish to place under source control. The folder that contains all your workspace folders and files is called the local folder, and the path to the local folder is referred to as the local path. This local folder will be bound to a particular folder in the repository.
3. In your Altova application, create an application project folder to which you must add the files you wish to place under source control. This organization of files in an application project is abstract. The files in a project reference physical files saved locally, preferably in one folder (with sub-folders if required) for each project.
4. In the source control system's database (also referred to as source control or repository), a folder is created that is bound to the local folder. This folder (called the bound folder) will replicate the structure of the local folder so that all files to be placed under source control are correctly located hierarchically within the bound folder. The bound folder is usually created when you add a file or an application project to source control for the first time.

## 10.2 Supported Source Control Systems

The list below shows the Source Control Servers (SCSs) supported by UModel, together with their respective Source Control Clients (SCCs). The list is organized alphabetically by SCS. Note the following:

- Altova has implemented the Microsoft Source Control Plug-in API (versions 1.1, 1.2, and 1.3) in UModel, and has tested support for the listed drivers and revision control systems. It is expected that UModel will continue to support these products if, and when, they are updated.
- Source Code Control clients not listed below, but which implement the Microsoft Source Control Plug-in API, should also work with UModel.

Source Control System	Source Code Control Clients
AccuRev 4.7.0 Windows	AccuBridge for Microsoft SCC 2008.2
Bazaar 1.9 Windows	Aigenta Unified SCC 1.0.6
Borland StarTeam 2008	Borland StarTeam Cross-Platform Client 2008 R2
Codice Software Plastic SCM Professional 2.7.127.10 (Server)	Codice Software Plastic SCM Professional 2.7.127.10 (SCC Plugin)
Collabnet Subversion 1.5.4	<ul style="list-style-type: none"> <li>Aigenta Unified SCC 1.0.6</li> <li>PushOK SVN SCC 1.5.1.1</li> <li>PushOK SVN SCC x64 version 1.6.3.1</li> <li>TamTam SVN SCC 1.2.24</li> </ul>
ComponentSoftware CS-RCS (PRO) 5.1	ComponentSoftware CS-RCS (PRO) 5.1
Dynamsoft SourceAnywhere for VSS 5.3.2 Standard/Professional Server	Dynamsoft SourceAnywhere for VSS 5.3.2 Client
Dynamsoft SourceAnywhere Hosted	Dynamsoft SourceAnywhere Hosted Client (22252)
Dynamsoft SourceAnywhere Standalone 2.2 Server	Dynamsoft SourceAnywhere Standalone 2.2 Client
Git	PushOK GIT SCC plug-in (see <a href="#">Source Control with Git</a> <small>(462)</small> )
IBM Rational ClearCase 7.0.1 (LT)	IBM Rational ClearCase 7.0.1 (LT)
March-Hare CVSNT 2.5 (2.5.03.2382)	Aigenta Unified SCC 1.0.6
March-Hare CVS Suite 2008	<ul style="list-style-type: none"> <li>Jalindi Igloo 1.0.3</li> <li>March-Hare CVS Suite Client 2008 (3321)</li> <li>PushOK CVS SCC NT 2.1.2.5</li> <li>PushOK CVS SCC x64 version 2.2.0.4</li> <li>TamTam CVS SCC 1.2.40</li> </ul>
Mercurial 1.0.2 for Windows	Sergey Antonov HgSCC 1.0.1
Microsoft SourceSafe 2005 with CTP	Microsoft SourceSafe 2005 with CTP

<b>Source Control System</b>	<b>Source Code Control Clients</b>
Microsoft Visual Studio Team System 2008/2010 Team Foundation Server	Microsoft Team Foundation Server 2008/2010 MSSCCI Provider
Perforce 2008 P4S 2008.1	Perforce P4V 2008.1
PureCM Server 2008/3a	PureCM Client 2008/3a
QSC Team Coherence Server 7.2.1.35	QSC Team Coherence Client 7.2.1.35
Reliable Software Code Co-Op 5.1a	Reliable Software Code Co-Op 5.1a
Seapine Surround SCM Client/Server for Windows 2009.0.0	Seapine Surround SCM Client 2009.0.0
Serena Dimensions Express/CM 10.1.3 for Win32 Server	Serena Dimensions 10.1.3 for Win32 Client
Softimage Alienbrain Server 8.1.0.7300	Softimage Alienbrain Essentials/Advanced Client 8.1.0.7300
SourceGear Fortress 1.1.4 Server	SourceGear Fortress 1.1.4 Client
SourceGear SourceOffsite Server 4.2.0	SourceGear SourceOffsite Client 4.2.0 (Windows)
SourceGear Vault 4.1.4 Server	SourceGear Vault 4.1.4 Client
VisualSVN Server 1.6	<ul style="list-style-type: none"><li>• Aigenta Unified SCC 1.0.6</li><li>• PushOK SVN SCC 1.5.1.1</li><li>• PushOK SVN SCC x64 version 1.6.3.1</li><li>• TamTam SVN SCC 1.2.24</li></ul>

## 10.3 Source Control Commands

The following sections use Visual SourceSafe to show the source control features of UModel. The examples in this section use the **Bank\_CSharp.ump** UModel project (and associated code files) available in the C:\Users\<username>\Documents\Altova\UModel2022\UModelExamples folder. Note that a Source Control project is not the same as a UModel project. Source Control projects are directory dependent, whereas UModel projects are logical constructions without direct directory dependence.

To access the Source Control commands, do one of the following:

- Use the menu command **Project | Source Control**
- Use the **context** menu in the Model Tree
- Click the source control toolbar buttons in the Source Control toolbar. Use **Tools | Customize | Toolbars** to activate the toolbar.

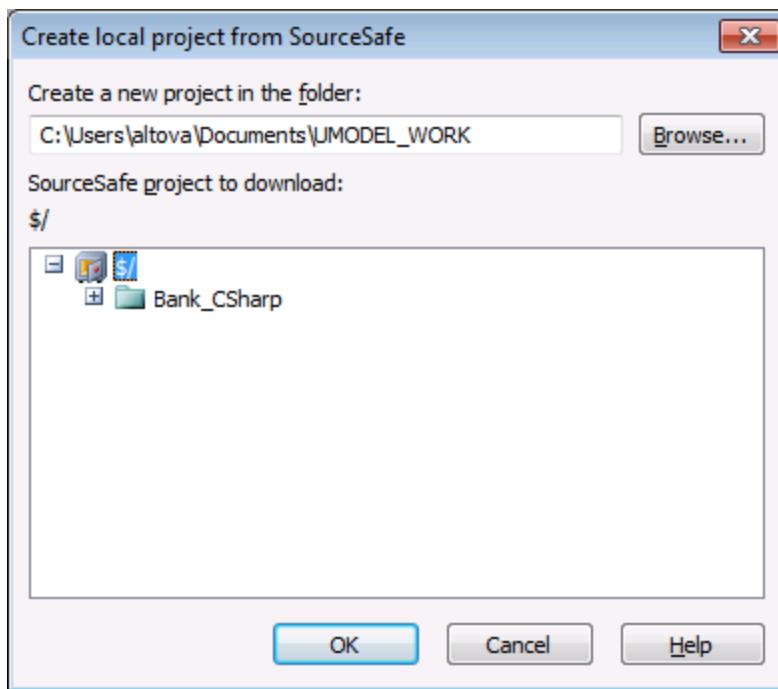
The description of the version control commands that follow apply to the standalone version of UModel. The Visual Studio and Eclipse versions of UModel use the version control functionality and menu items available in those IDEs.

<a href="#">Open from Source Control</a>	<sup>442</sup>
<a href="#">Enable Source Control</a>	<sup>445</sup>
<a href="#">Get Latest Version</a>	<sup>446</sup>
<a href="#">Get</a>	<sup>446</sup>
<a href="#">Get Folder(s)</a>	<sup>447</sup>
<a href="#">Check Out</a>	<sup>448</sup>
<a href="#">Check In</a>	<sup>450</sup>
<a href="#">Undo Check Out...</a>	<sup>450</sup>
<a href="#">Add to Source Control</a>	<sup>452</sup>
<a href="#">Remove from Source Control</a>	<sup>454</sup>
<a href="#">Share from Source Control</a>	<sup>455</sup>
<a href="#">Show History</a>	<sup>456</sup>
<a href="#">Show Differences</a>	<sup>458</sup>
<a href="#">Show Properties</a>	<sup>459</sup>
<a href="#">Refresh Status</a>	<sup>460</sup>
<a href="#">Source Control Manager</a>	<sup>460</sup>
<a href="#">Change Source Control</a>	<sup>460</sup>

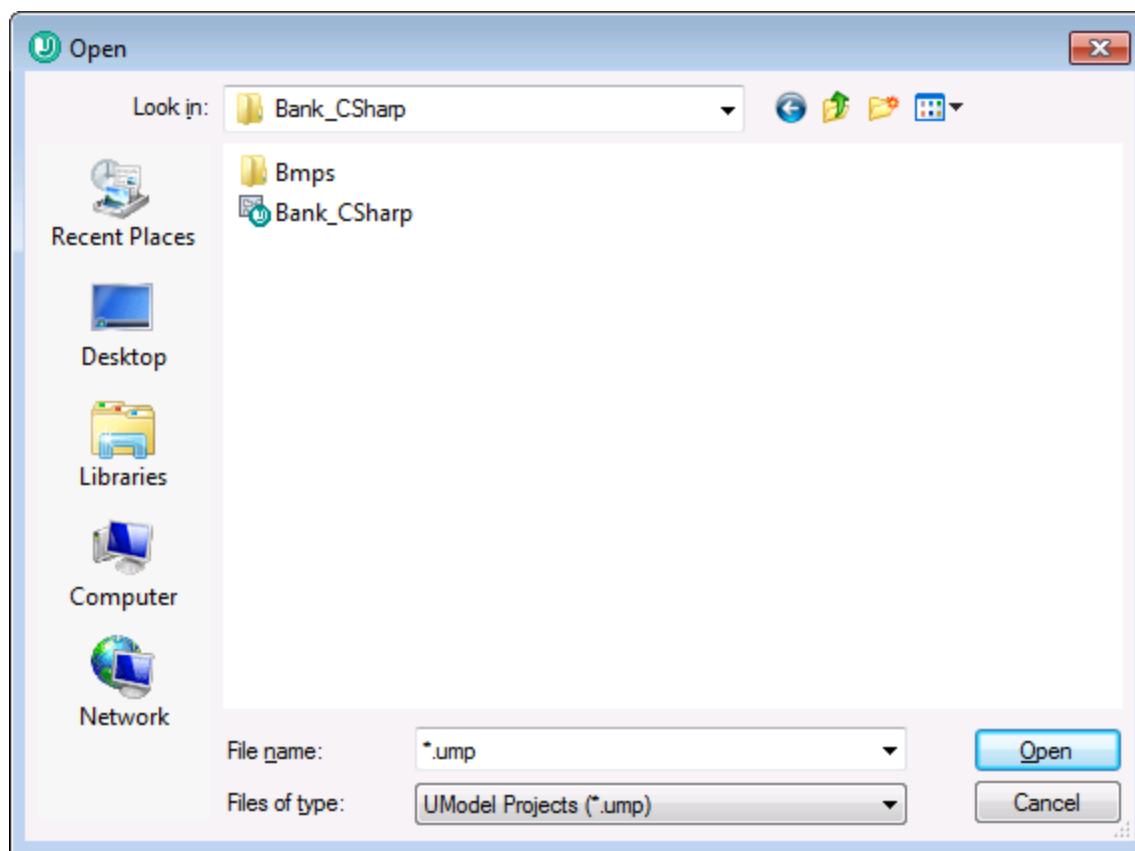
### 10.3.1 Open from Source Control

The Open from Source Control command creates a local project from an existing source control database, and places it under source control, SourceSafe in this case.

1. Select **Project | Source Control | Open from Source Control**.  
The Login dialog box is opened, enter your login details to continue.  
The "Create local project from SourceSafe" dialog box appears.
2. Define the directory to contain the new local project e.g. c:\temp\ssc. This becomes the **Working directory**, or the Check Out Folder.



3. Select the SourceSafe project you want to download e.g. Bank\_CSharp.  
If the folder you define here does not exist at the location, a dialog box opens prompting you to create it.
4. Click **Yes** to create the new directory.  
The Open dialog box is now visible.



5. Select the **Bank\_CSharp.ump** UModel project file and click Open.

**Bank\_CSharp.ump** now opens in UModel, and the file is placed under source control. This is indicated by the lock symbol visible on the Root folder in the Model Tree window. The Root folder represents both the project file and the working directory for source control operations.



The BankCSharp directory has been created locally, you can now work with these files as you normally would.

Note:

To place under source control the code files generated when synchronizing code, see: [Add to Source Control](#)<sup>452</sup>

## Source control symbols



or



The lock symbol denotes that the file, or folder is under source control, but is currently not checked out.



or



The red check mark denotes checked out, i.e. the UModel project file (or code file) has been checked out for editing. The asterisk in the Application title bar denotes that changes have been made to the file, and you will be prompted to save it when you exit.



or



The arrow symbol shows that the file(s) have been checked out by someone else in the network, or by you into a different working directory

### 10.3.2 Enable Source Control

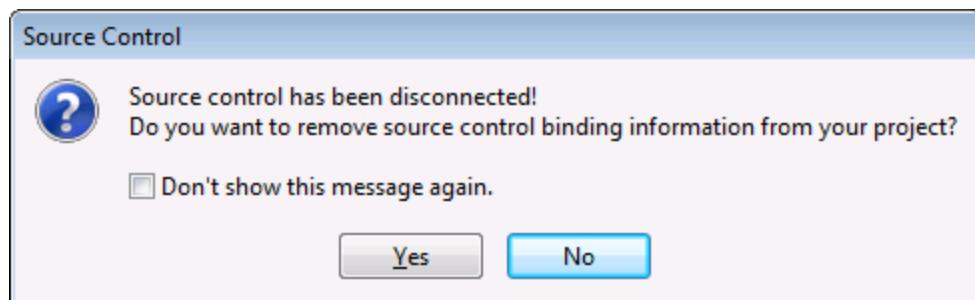
This command allows you to enable or disable source control for a UModel project and is available through the Project menu item, i.e. **Project | Source Control | Enable Source Control**. Selecting this option on any file or folder, enables/disables source control for the whole UModel project.

#### To enable Source Control for a project:

1. Select the menu option **Project | Source Control** and activate/check the **Enable source control** check box of the fly-out menu. The previous check in/out status of the various files are retrieved and displayed in the Model Tree window.

#### To disable Source Control for a project:

1. Select the menu option **Project | Source Control** and uncheck the **Enable source control** check box.



You are now prompted if you want to remove the binding information from the project.

To **provisionally** disable source control for the project, select **No**.

To **permanently** disable source control for the project, select **Yes**.

### 10.3.3 Get Latest Version

**Retrieves and places** the latest source control version of the selected file(s) in the working directory. The files are retrieved as read-only and are not checked out.

If the affected files are currently checked out, different things occur depending on the specific version control plugin: nothing happens, new data are merged into your local file, or your changes are overwritten.

This command works in a similar fashion to the Get command, but does not display the "Source control - Get" dialog box. It is therefore not possible to specify Advanced get options.

Note that this command automatically performs a recursive get latest version operation when performed on a folder, i.e. it affects all other files below the current one in the package hierarchy.

**To get the latest version of a file:**

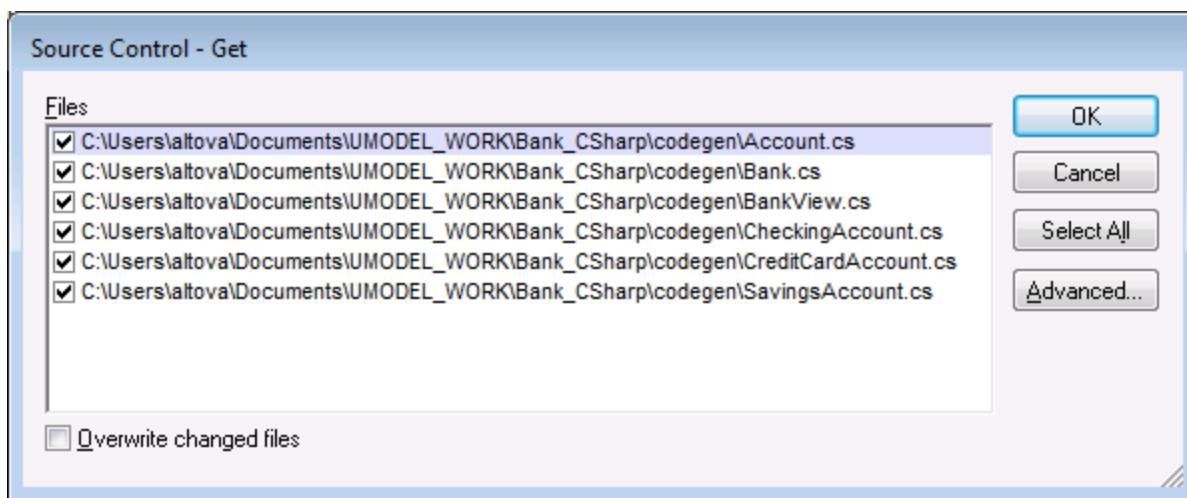
1. Select the file(s) you want to get the latest version of in the Model Tree.
2. Select **Project | Source Control | Get Latest Version**.

### 10.3.4 Get

Retrieves a read-only copy of the selected files and places them in the working folder. The files are not checked-out for editing per default.

**Using Get:**

- Select the files you want to get in the Model Tree.
- Select **Project | Source Control | Get**.



#### Overwrite changed files

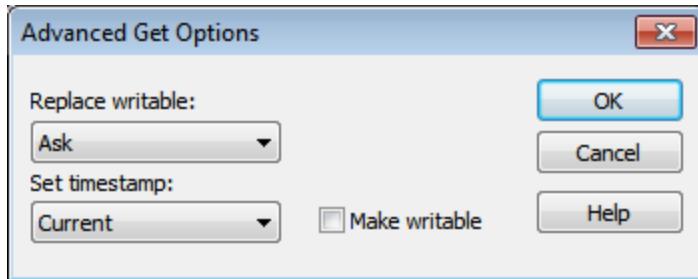
Overwrites those files that have been changed locally with those from the source control database.

#### Select All

Selects all the files in the list box.

#### Advanced

Allows you to define the **Replace writable** and **Set timestamp** options in the respective combo boxes.



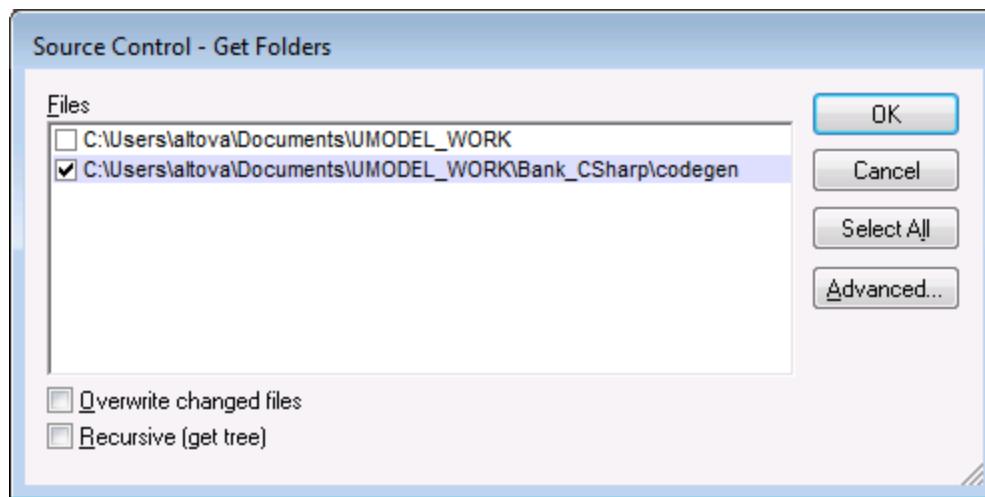
The "Make writable" check box removes the read-only attribute of the retrieved files.

### 10.3.5 Get Folder(s)

Retrieves read-only copies of files in the selected folders and places them in the working folder. The files are not checked-out for editing per default.

#### Using Get Folders:

- Select the folder you want to get in the Model Tree.
- Select **Project | Source Control | Get Folders**.



#### Overwrite changed files

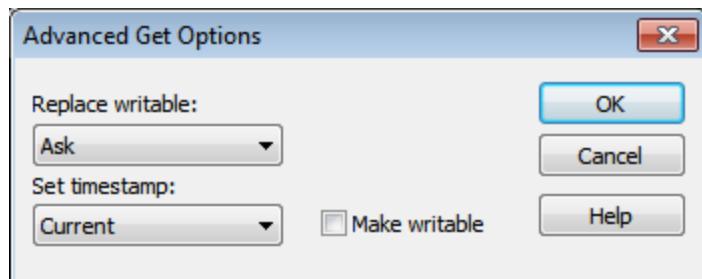
Overwrites those files that have been changed locally with those from the source control database.

#### Recursive (get tree)

Retrieves all files of the folder tree below the selected folder.

#### Advanced

Allows you to define the **Replace writable** and **Set timestamp** options in the respective combo boxes.



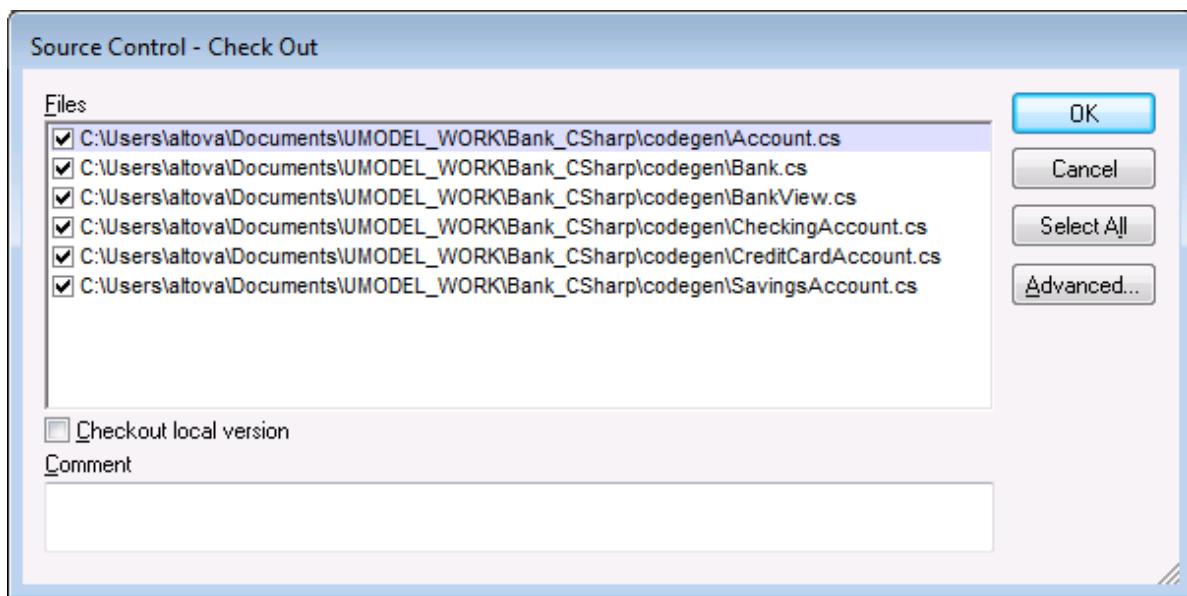
The "Make writable" check box removes the read-only attribute of the retrieved files.

## 10.3.6 Check Out

This command **checks out** the latest version of the selected files and places writable copies in the working directory. The files are flagged as "checked out" for all other users.

#### To Check Out files:

- Select the file or folder you want to check out in the Model Tree.
- Select **Project | Source Control | Check Out**.



**Note:** You can change the number of files to check out, by activating the individual check boxes in the Files list box.

Select the option **Checkout local version** to check out only the local versions of the files, not those from the source control database.

The following items can be checked out:

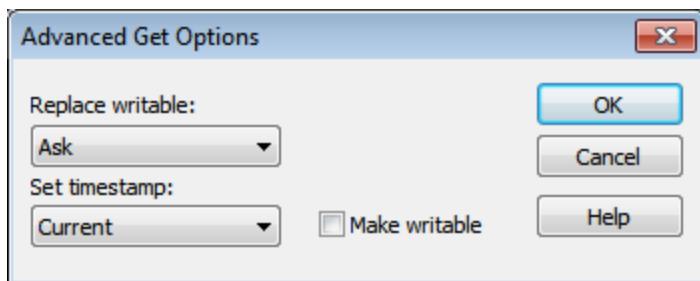
- Single files, click on the respective files (CTRL + click, in the Model Tree)
- Folders, click on the folders (CTRL + click, in the Model Tree)



The red check mark denotes that the file/folder has been checked out.

## Advanced

Allows you to define the **Replace writable** and **Set timestamp** options in the respective combo boxes.



The "Make writable" check box removes the read-only attribute of the retrieved files.

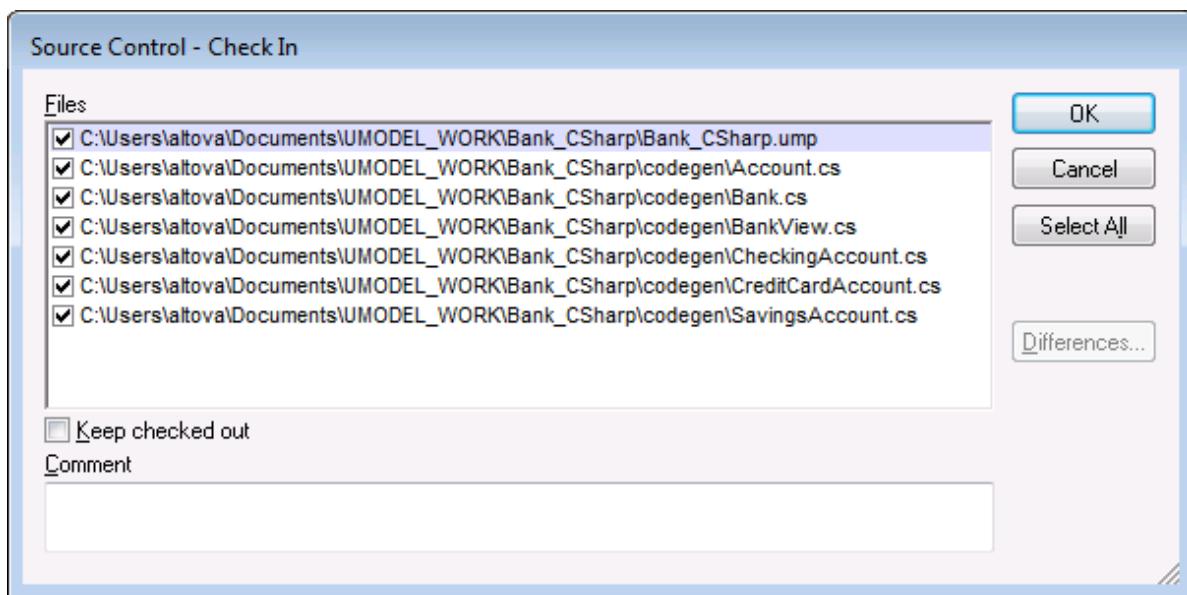
### 10.3.7 Check In

This command **checks in** the previously checked out files, i.e. your locally updated files, and places them in the source control database.

#### To Check In files:

- Select the files in the Model Tree
- Select **Project | Source Control | Check In**.

**Shortcut:** Right-click a checked out item in the project window, and select "Check in" from the Context menu.

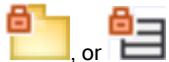


#### Note:

You can change the number of files to check in, by activating the individual check boxes in the Files list box.

The following items can be checked in:

- Single files, click on the respective files (CTRL + click, in Model Tree)
- Folders, click on the folders (CTRL + click, in Model Tree)



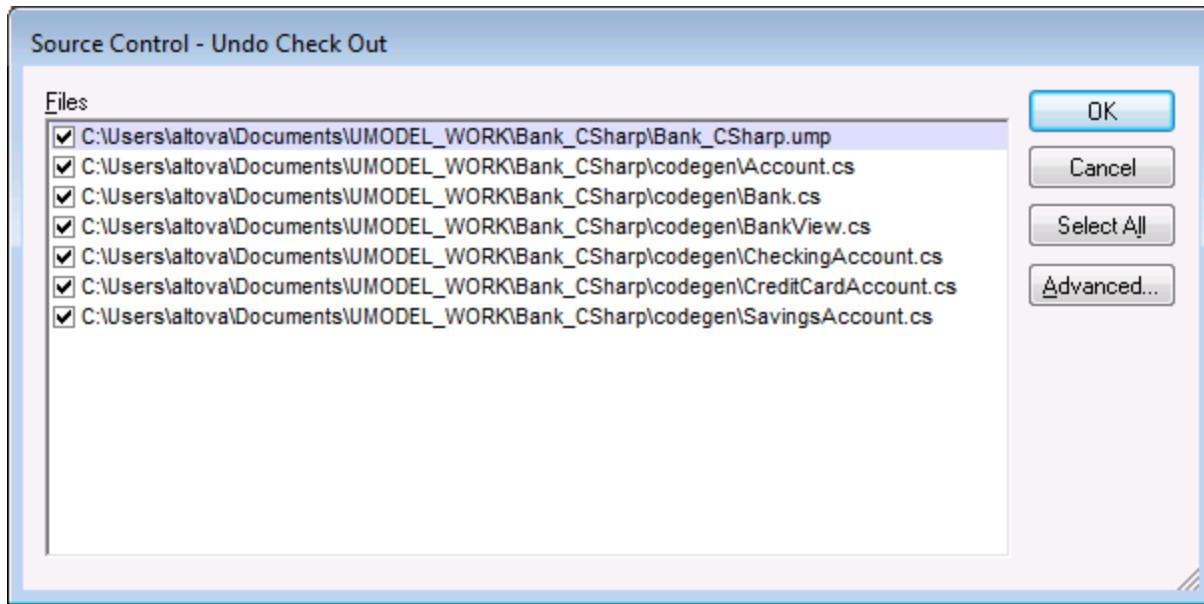
The lock symbol denotes that the file/folder is **under source control**, but is currently not checked out.

### 10.3.8 Undo Check Out...

This command **discards changes** made to previously checked out files, i.e. your locally updated files, and retains the old files from the source control database.

### To Undo Check Out..

- Select the files in the Model Tree
- Select **Project | Source Control | Undo Check Out.**



Note:

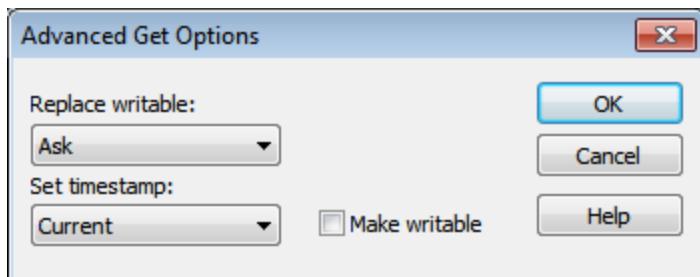
You can change the number of files by activating the individual check boxes in the Files list box.

The Undo check out option can apply to the following items:

- Single files, click on the respective files (CTRL + click, in Model Tree)
- Folders, click on the folders (CTRL + click, in Model Tree)

### Advanced

Allows you to define the **Replace writable** and **Set timestamp** options in the respective combo boxes.



The "Make writable" check box removes the read-only attribute of the retrieved files.

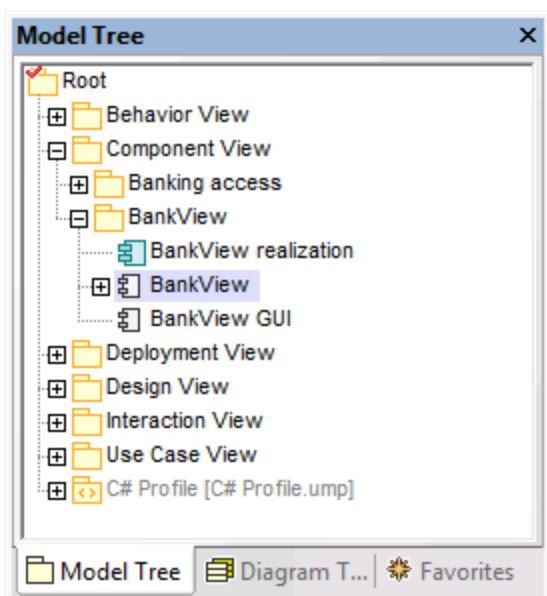
### 10.3.9 Add to Source Control

Adds the selected files or folders to the source control database and places them under source control. If you are adding a new UModel project you will be prompted for the workspace folder and the location at which your project should be stored.

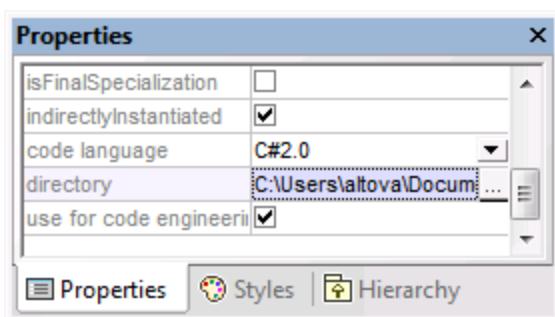
Having placed the UModel project file (\*.ump) under source control, you can then add the code files produced by the code-engineering process, to source control as well. For this to work, the generated code files and the UModel project have to be placed **in**, or **under**, the same SourceSafe **working** directory. The working directory used in this section is **C:\Users\Altova\Documents\UMODEL\_WORK\**.

**To add UModel generated code files to source control:**

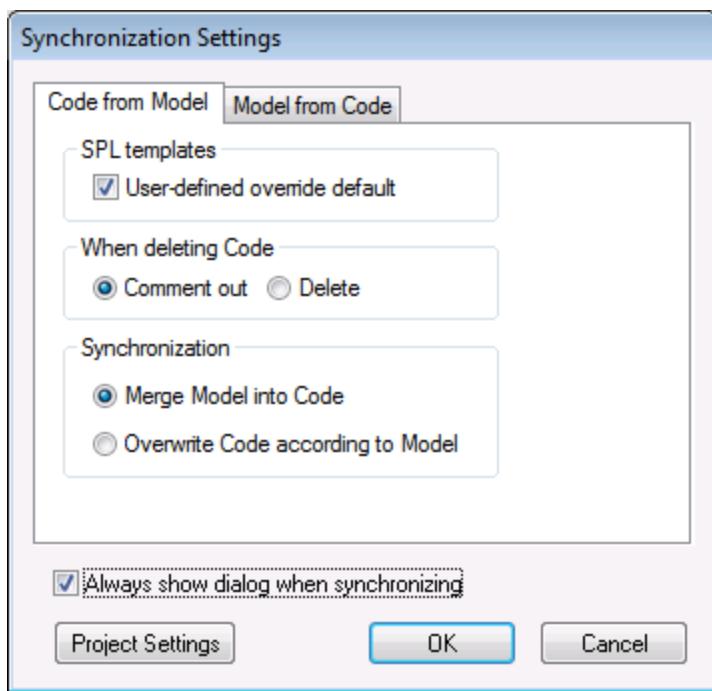
1. Expand the Component View folder in the Model Tree and Navigate to the BankView component.



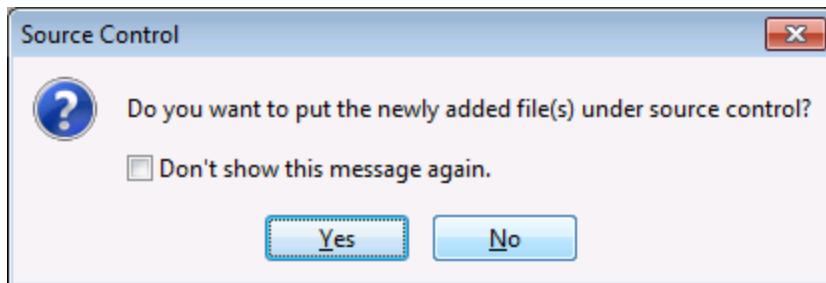
2. Click the BankView component and click the Browse icon next to the "directory" field in the Properties window.



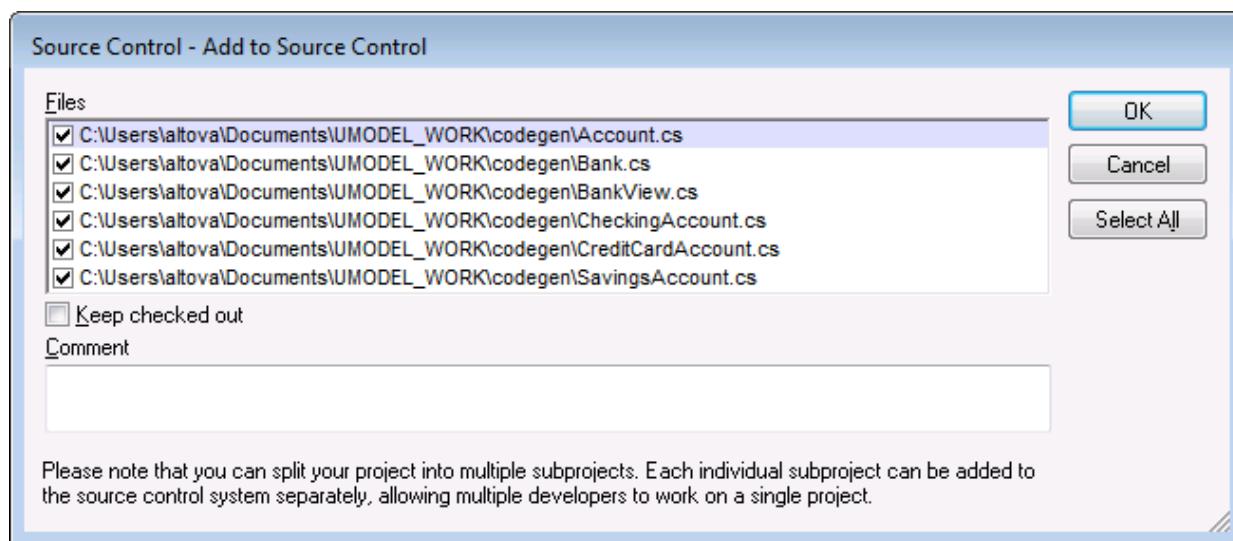
3. Change the code engineering directory to **C:\Users\Altova\Documents\UMODEL\_WORK\codegen**.
4. Select the menu item **Project | Merge Program Code from UModel project**.
5. Change the Synchronization settings if necessary, and click OK to confirm.



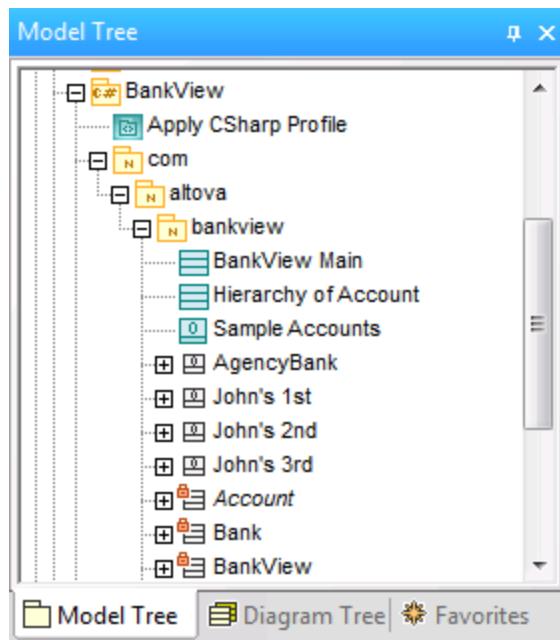
The Messages window displays the code from project process.  
A message box opens asking if you want to place the newly created files under source control.



6. Click Yes to do so.
7. The "Add to Source Control" dialog box is opened, allowing you to select the files you want to place under source control.



- Click OK once you have selected the files you want to place under source control.  
The lock symbol now appears next to each of the classes/file sources placed under source control.

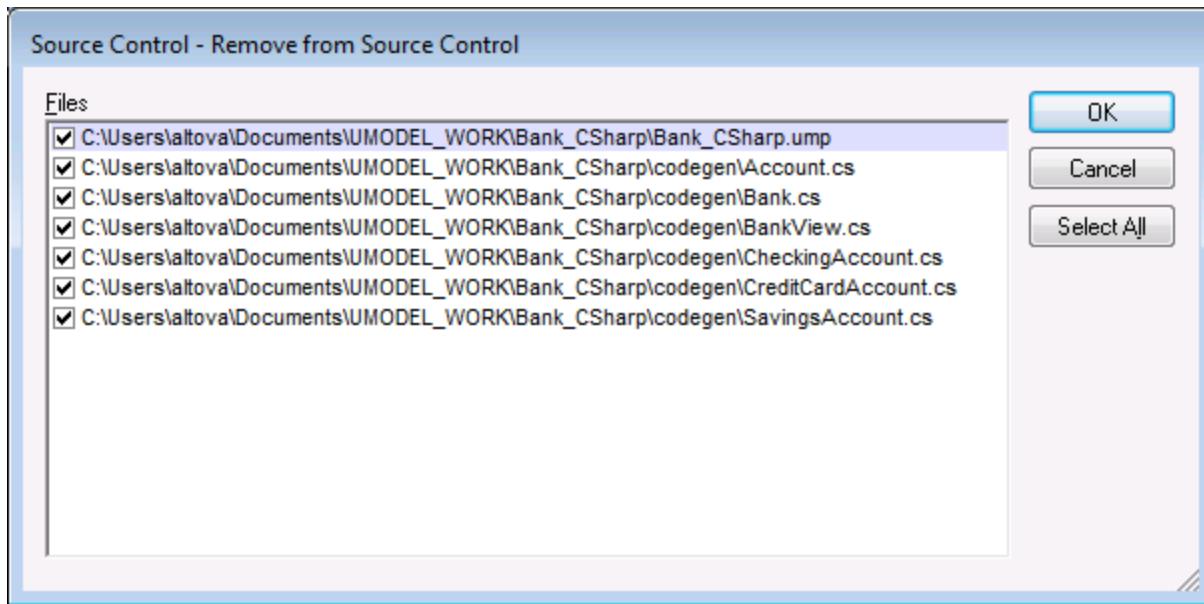


### 10.3.10 Remove from Source Control

This command **removes** previously added files, from the source control database. These type of files remain visible in the Model Tree but cannot be checked in or out. Use the "Add to Source Control" command to place them back under source control.

**To remove files from the source control provider:**

- Select the files you want to remove in the Model Tree.
- Select **Project | Source Control | Remove from Source Control**.

**Note:**

You can change the number of files to remove, by activating the individual check boxes in the Files list box.

The following items can be removed from source control:

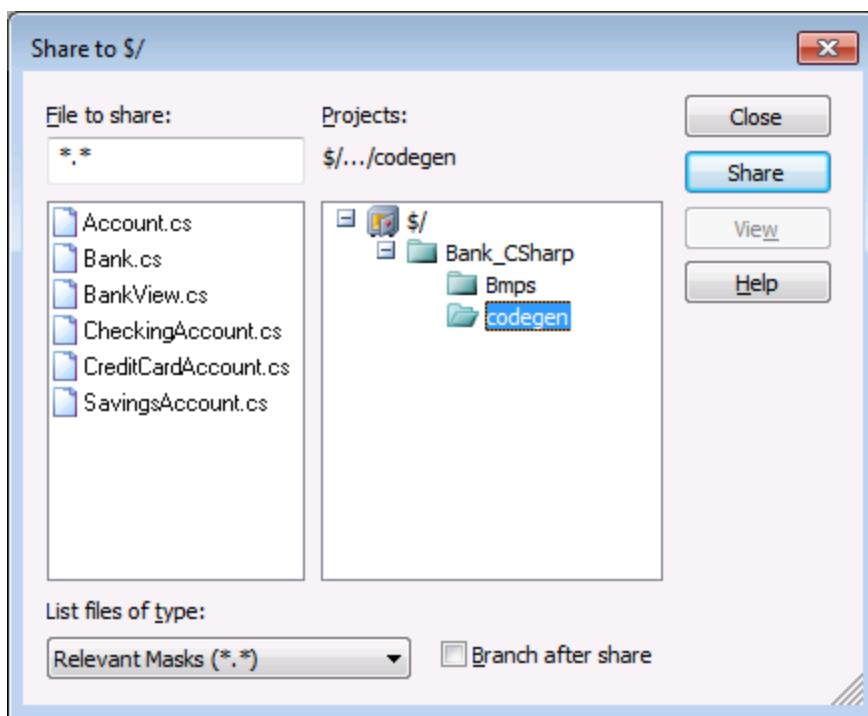
- Single files, click on the respective files (CTRL + click, for several)
- Folders, click on the folder icon.

### 10.3.11 Share from Source Control

This command shares/branches files from other projects/folders within the source control repository, into the selected folder. To use the Share command you must have the Check in/out rights to the project you are sharing from.

**To share a file from source control:**

1. Select the folder you want to share files to, in the Model Tree window, and select **Project | Source Control | Share from Source Control**. e.g. BankView Component in the Component View folder.
2. Select the project folder that contains the file you want to share in the "Projects" list box.



3. Select the file you want to share in the "Files to share" list box and click the Share button.  
The file is now removed from the "File to share" list.
4. Click the Close button to continue.

#### Branch after share

Shares the file and creates a new branch to create a separate version.

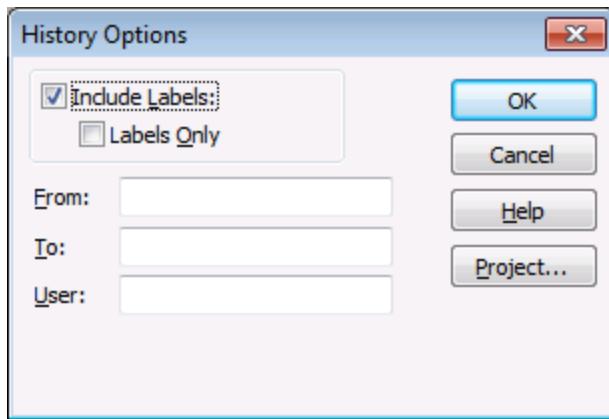
### 10.3.12 Show History

This command **displays** the history of a file under source control, and allows you to view, see detailed history info, difference, or retrieve previous versions of a file.

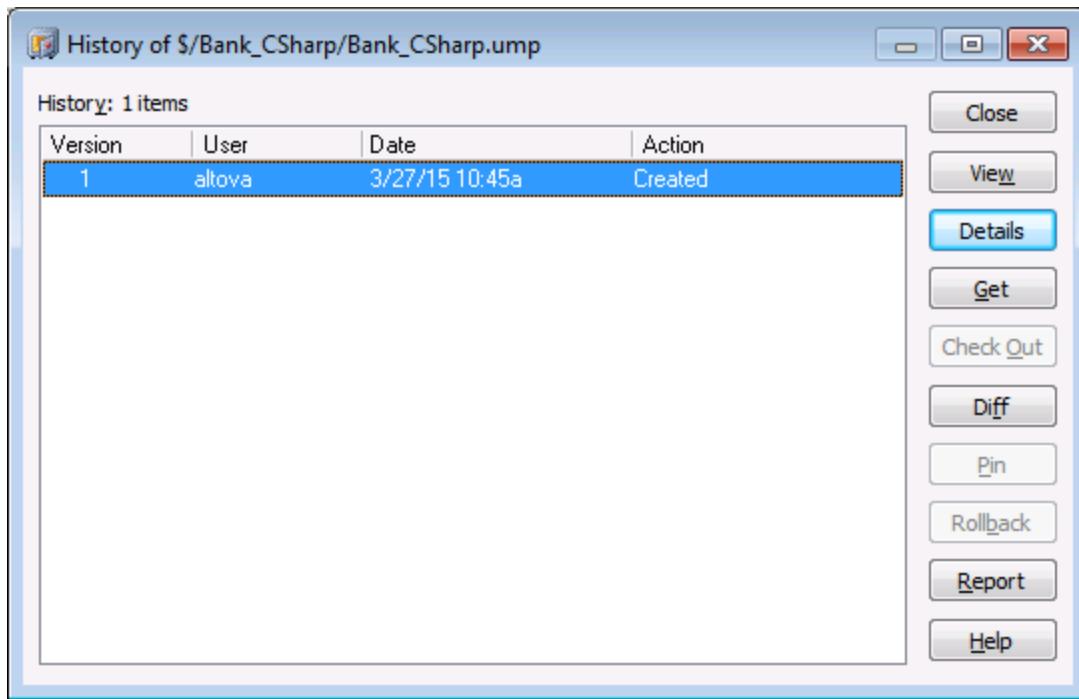
#### To show the history of a file:

1. Click on the file in the Model Tree window.
2. Select the menu options **Project | Source control | Show history**.

A dialog box prompting for more information opens.



3. Select the appropriate entries and confirm with **OK**.



This dialog box provides various ways of comparing and getting specific versions of the file in question. Double clicking an entry in the list opens the History Details dialog box for that file.

#### Close

Closes this dialog box.

#### View

Opens a further dialog box in which you can select the type of viewer you want to see the file with.

#### Details

Opens a dialog box in which you can see the [properties](#) 459 of the currently active file.

#### Get

Allows you to retrieve one of the previous versions of the file in the version list, and place it into the working directory.

**Check Out**

Allows you to check out the **latest** version of the file.

**Diff**

Opens the [Difference options](#) 458 dialog box, which allows you to define the difference options when viewing the differences between two file versions.

Use CTRL+Click to mark two file versions in this window, then click Diff to view the differences between them.

**Pin**

Pins or unpins a version of the file, allowing you to define the specific file version to use when differencing two files.

**Rollback**

Rolls back to the selected version of the file.

**Report**

Generates a history report which you can send to the printer, file, or clipboard.

**Help**

Opens the online help of the source control provider plugin.

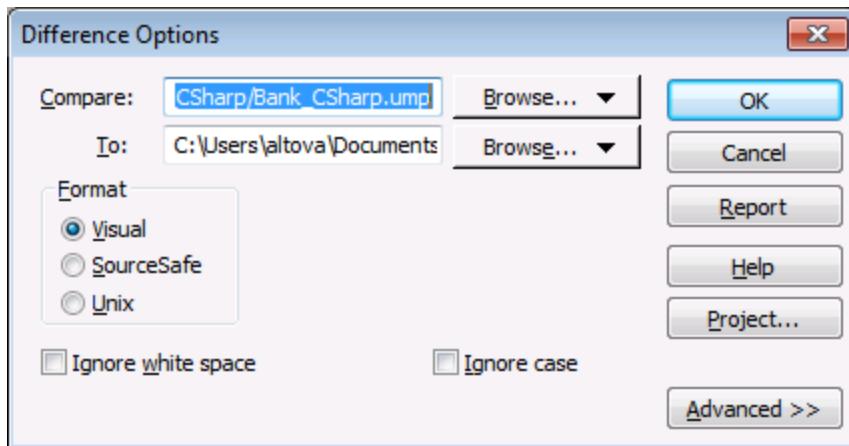
### 10.3.13 Show Differences

This command **displays** the differences between the file currently in the source control repository, and the **checked in/out** file of the same name in the working directory.

If you have "pinned" one of the files in the **history** dialog box, then the pinned file will be used in the "Compare" text box. Any two files can be selected using the Browse buttons.

**To show the differences between two files:**

1. Click on a file in the Model Tree window.
2. Select the menu option **Project | Source control | Show Differences**.  
A dialog box prompting for more information appears.



3. Select the appropriate entries and confirm with **OK**.

The screenshot shows the 'Differences' viewer comparing two XML files. The left pane shows the XML content of 'S/Bank\_CSharp/Bank\_CSharp.ump' and the right pane shows the XML content of 'C:\Users\altova\Documents\UMODEL\_WORK\Bank\_CSharp\Ban...'. The differences are highlighted in green (inserted text) and red (deleted text). The XML code includes various UML model elements like Settings, Model, Package, and Component. At the bottom of the viewer, there are tabs for 'Deleted Text' (red), 'Changed Text' (blue), and 'Inserted Text' (green), with 'Ln 1, Col 1' selected.

The differences between the two files are highlighted in both windows (this example uses MS SourceSafe).

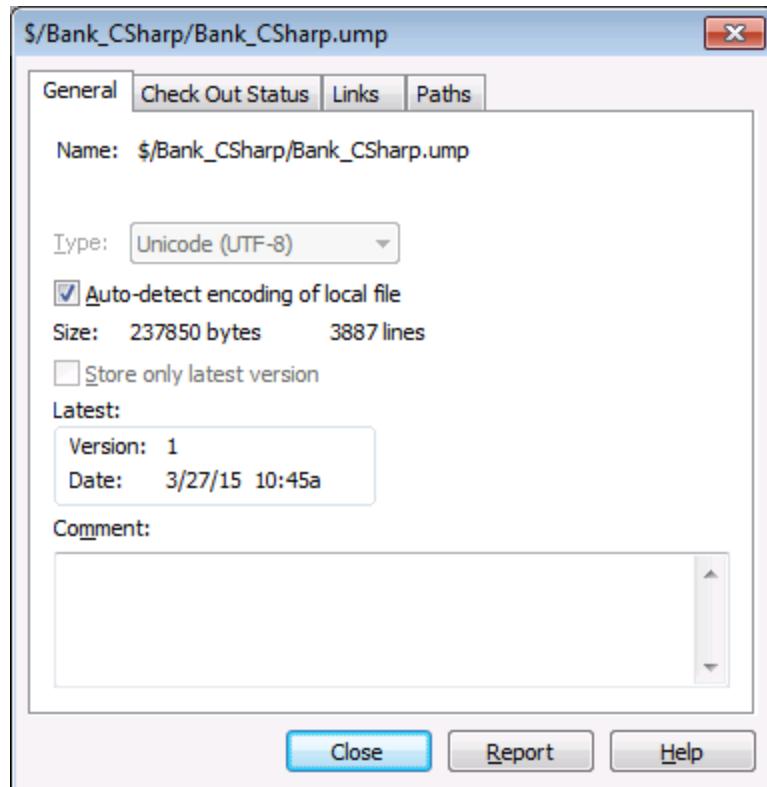
### 10.3.14 Show Properties

This command **displays** the properties of the currently selected file, and is dependent on the source control provider you use.

To display the properties of the currently selected file:

- Select **Project | Source Control | Properties**.

This command can only be used on single files.



### 10.3.15 Refresh Status

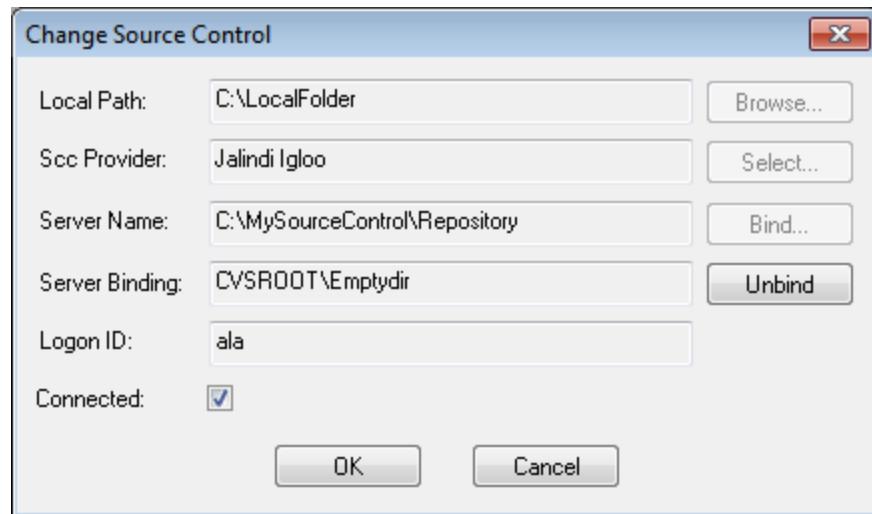
This command **refreshes** the status of all project files, independent of their current status.

### 10.3.16 Source Control Manager

This command **starts** your source control software with its native user interface.

### 10.3.17 Change Source Control

This dialog box allows you to change the source control binding that you are using. Click the Unbind button first, then (optionally) click the Select button to select a new source control provider, and finally click the Bind button to bind to a new location in the repository.

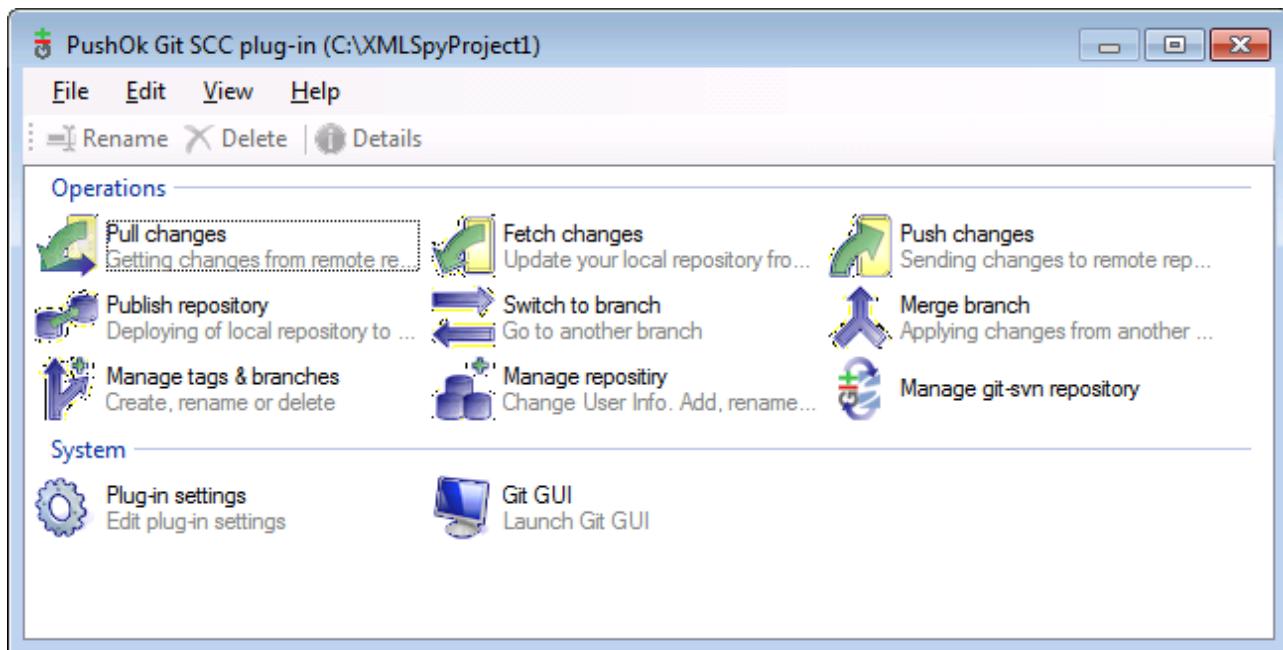


## 10.4 Source Control with Git

Support for Git as a source control system in UModel is available through a third-party plug-in called **GIT SCC plug-in** (<http://www.pushok.com/software/git.html>).

At the time when this documentation is written, the **GIT SCC plug-in** is available for experimental use. Registration with the plug-in publisher is required in order to use the plug-in.

The GIT SCC plug-in enables you to work with a Git repository using the commands available in the **Project | Source Control** menu of UModel. Note that the commands in the **Project | Source Control** menu of UModel are provided by the Microsoft Source Control Plug-in API (MSSCCI API), which uses a design philosophy different from Git. As a result, the plug-in essentially mediates between "Visual Source Safe"-like functionality and Git functionality. On one hand, this means that a command such as **Get latest version** may not be applicable with Git. On the other hand, there are new Git-specific actions, which are available in the "Source Control Manager" dialog box provided by the plug-in (under the **Project | Source Control | Source Control Manager** menu of UModel).



The Source Control Manager dialog box

Other commands that you will likely need to use frequently are available directly under the **Project | Source Control** menu.

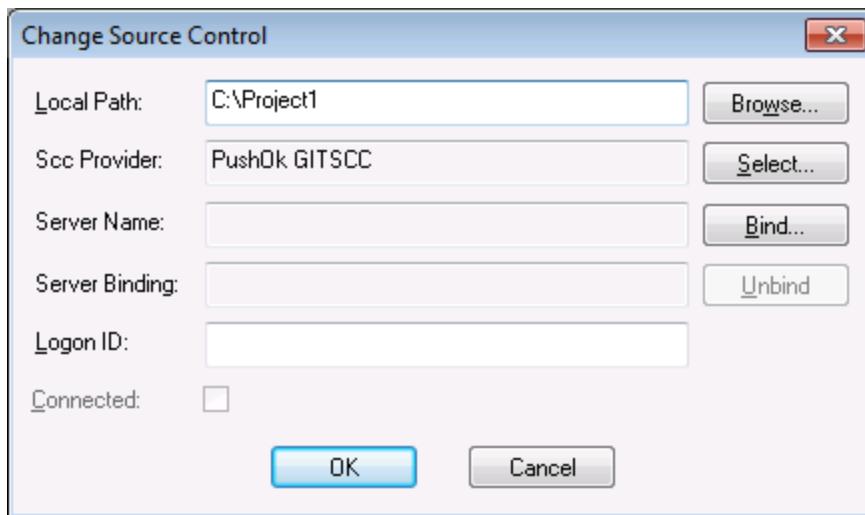
The following sections describe the initial configuration of the plug-in, as well as the basic workflow:

- [Enabling Git Source Control with GIT SCC Plug-in](#) 463
- [Adding a Project to Git Source Control](#) 463
- [Cloning a Project from Git Source Control](#) 465

### 10.4.1 Enabling Git Source Control with GIT SCC Plug-in

To enable Git source control with UModel, the third-party **PushOK GIT SCC plug-in** must be installed, registered, and selected as source control provider, as follows:

1. Download the plug-in installation file from the publisher's website (<http://www.pushok.com>), run it, and follow the installation steps.
2. On the **Project** menu of UModel, click **Change Source Control**, and make sure **PushOk GITSCC** is selected as source control provider. If you do not see **Push Ok GITSCC** in the list of providers, it is likely that the installation of the plug-in was not successful. In this case, check the publisher's documentation for a solution.



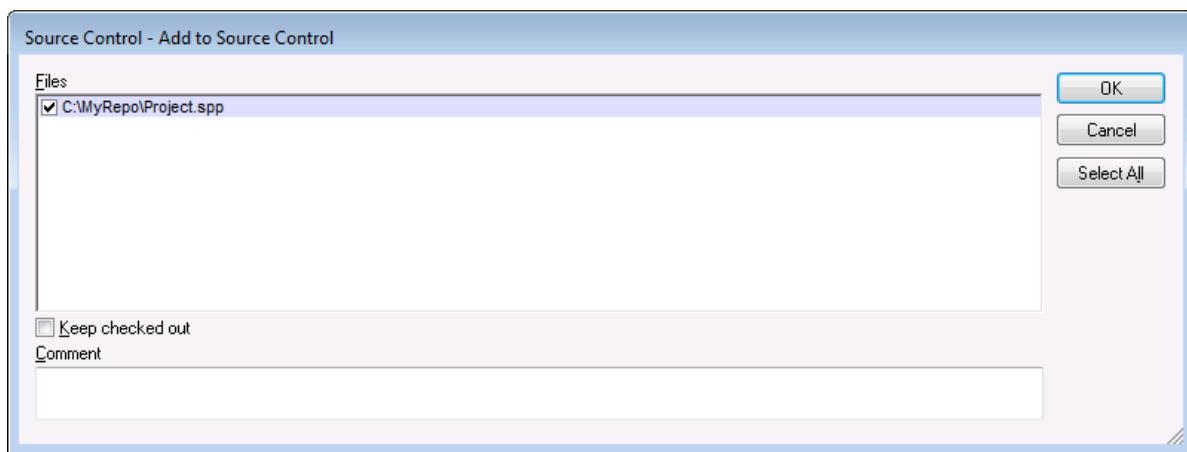
3. When a dialog box prompts you to register the plug-in, click **Registration** and follow the wizard steps to complete the registration process.

### 10.4.2 Adding a Project to Git Source Control

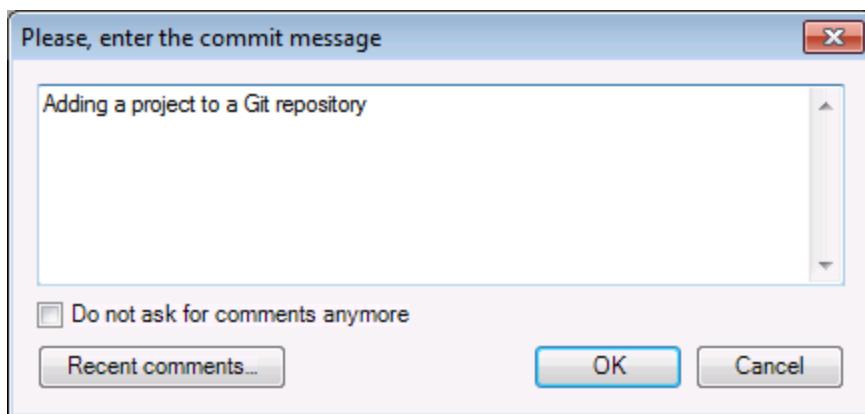
You can save UModel projects as Git repositories. The structure of files or folders that you add to the project would then correspond to the structure of the Git repository.

#### To add a project to Git source control:

1. Make sure that **PushOK GIT SCC Plug-in** is set as source control provider (see [Enabling Git Source Control with GIT SCC Plug-in](#) ).
2. Create a new empty project and make sure that it has no validation errors (that is, the command **Project | Check Project Syntax** does not show any errors or warnings).
3. Save the project to a local folder, for example C:\MyRepo\Project.ump.
4. In the **Model Tree** pane, click the **Root** node.
5. On the **Project** menu, under **Source Control**, click **Add to Source Control**.

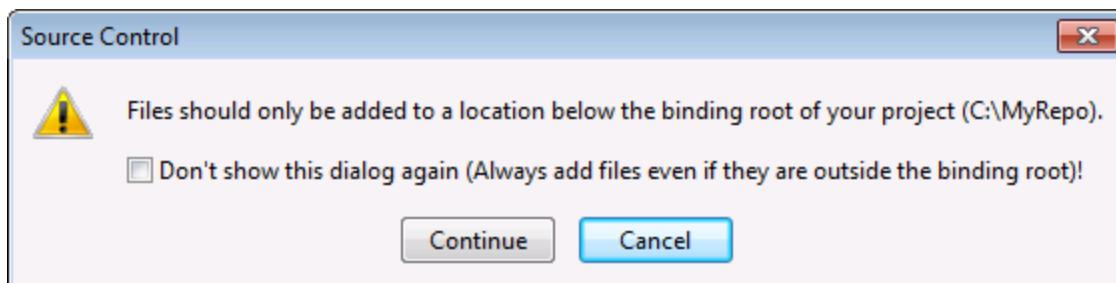


6. Click **OK**.



7. Enter the text of your commit message, and click **OK**.

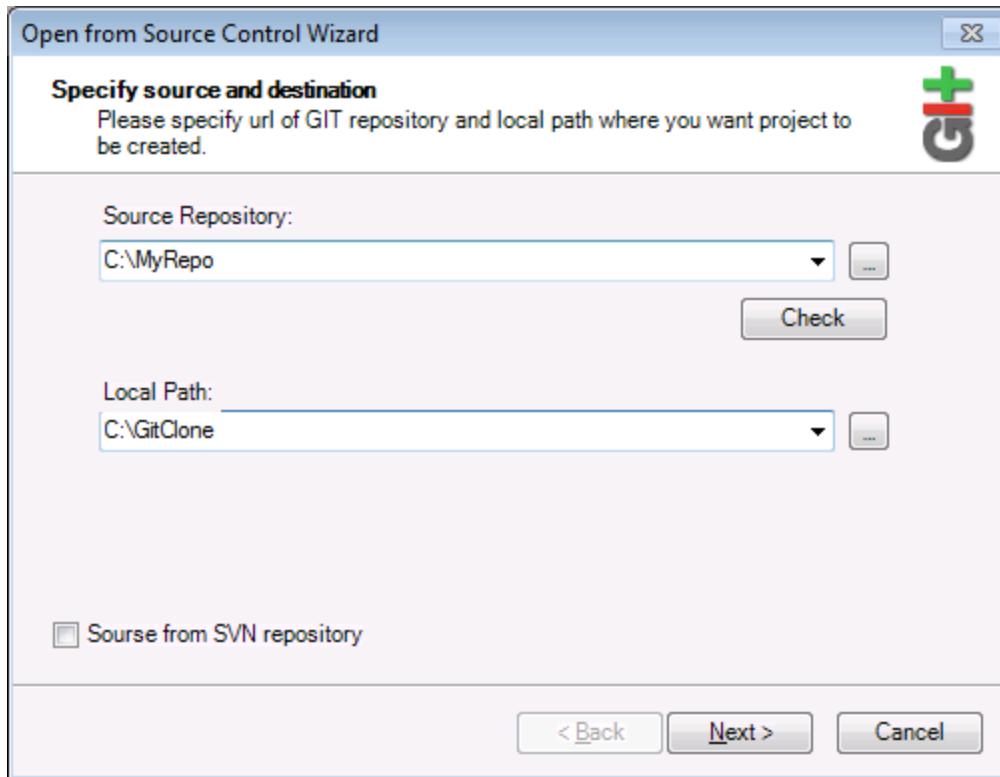
You can now start adding modeling elements (diagrams, classes, packages, and so on) to your project. Note that all project files and folders must be under the root folder of the project. For example, if the project was created in the C:\MyRepo folder, then only files under C:\MyRepo should be added to the project. Otherwise, if you attempt to add to your project files that are outside the project root folder, a warning message is displayed:



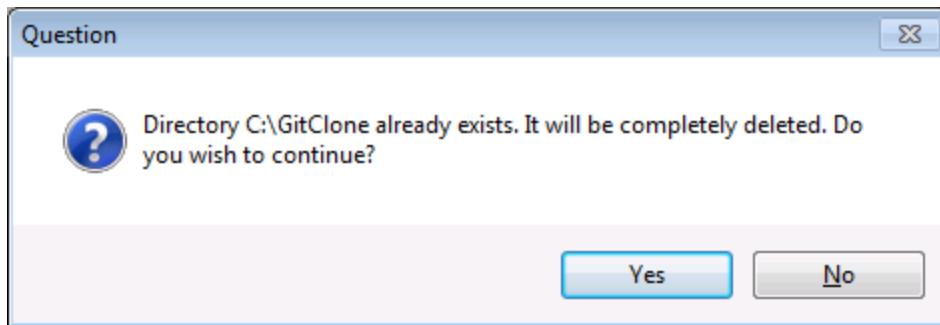
### 10.4.3 Cloning a Project from Git Source Control

Projects that have been previously added to Git source control (see [Adding a Project to Git Source Control](#)<sup>463</sup>) can be opened from the Git repository as follows:

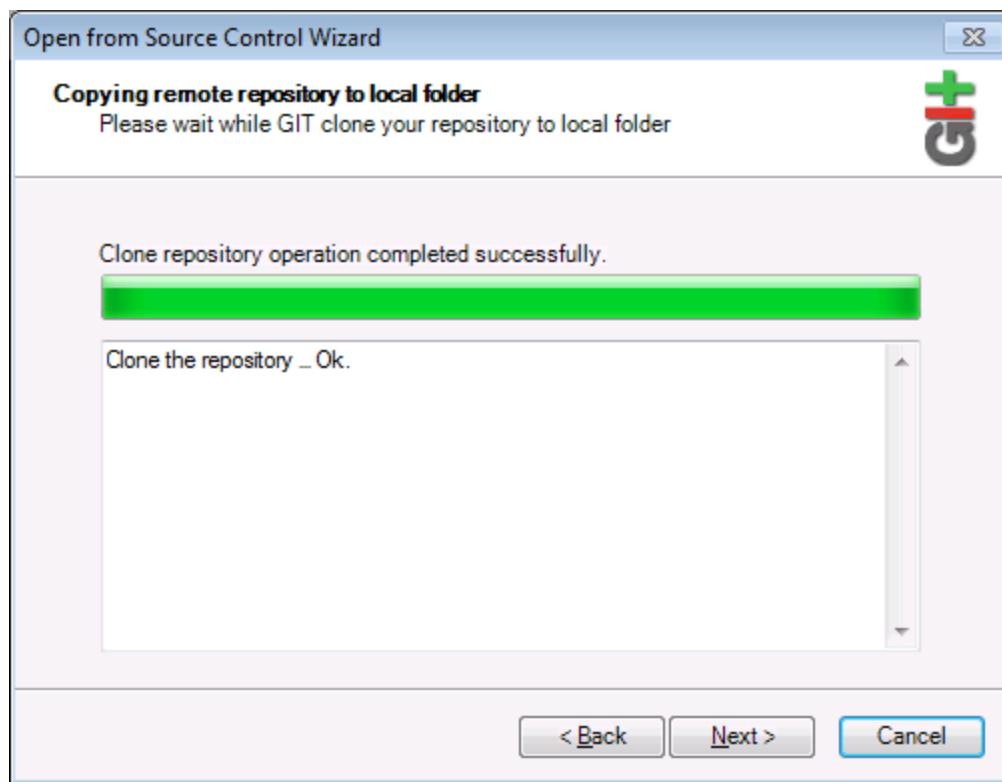
1. Make sure that **PushOK GIT SCC Plug-in** is set as source control provider (see [Enabling Git Source Control with GIT SCC Plug-in](#)<sup>463</sup>).
2. On the **Project** menu, click **Source Control | Open from Source Control**.
3. Enter the path or the URL of the source repository. Click **Check** to verify the validity of the path or URL.



4. Under **Local Path**, enter the path to local folder where you want the project to be created, and click **Next**. If the local folder exists (even if it is empty), the following dialog box opens:



5. Click **Yes** to confirm, and then click **Next**.



6. Follow the remaining wizard steps, as required by your specific case.
7. When the wizard completes, a Browse dialog box appears, asking you to open the UModel Project (\*.ump) file. Select the project file to load the project contents into UModel.

## 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 (CallBehaviorAction)  
Action (CallOperationAction)  
AcceptEventAction  
AcceptEventAction (TimeEvent)  
SendSignalAction

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

InputPin  
OutputPin  
ValuePin

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

ControlFlow  
ObjectFlow  
ExceptionHandler

Activity  
ActivityParameterNode  
StructuredActivityNode  
ExpansionRegion  
ExpansionNode  
InterruptibleActivityRegion

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 Profile Diagram



Add

Profile

Stereotype

Relationship

Generalization

ProfileApplication

PackageImport

ElementImport

Note

NoteLink

## 11.11 Protocol State Machine



### Add

Simple state  
Composite state  
Orthogonal state  
Submachine state

FinalState  
InitialState

EntryPoint  
ExitPoint  
Choice  
Junction  
Terminate  
Fork  
Fork (horizontal)  
Join  
Join (horizontal)  
ConnectionPointReference

### Relationship

Protocol Transition

Note  
Note link

## 11.12 Sequence Diagram



### Add

Lifeline

CombinedFragment

CombinedFragment (Alternatives)

CombinedFragment (Loop)

InteractionUse

Gate

StateInvariant

DurationConstraint

TimeConstraint

Message (Call)

Message (Reply)

Message (Creation)

Message (Destruction)

Asynchronous Message (Call)

Asynchronous Message (Reply)

Asynchronous Message (Destruction)

Note

Note Link

No message numbering

Simple message numbering

Nested message numbering

Toggle dependent message movement

Toggle automatic creation of replies for messages

Toggle automatic creation of operations in target by typing operation names

## 11.13 State Machine Diagram



### Add

Simple state  
Composite state  
Orthogonal state  
Submachine state

FinalState  
InitialState

EntryPoint  
ExitPoint  
Choice  
Junction  
Terminate  
Fork  
Fork (horizontal)  
Join  
Join (horizontal)  
DeepHistory  
ShallowHistory  
ConnectionPointReference

### Relationship

Transition

Note  
Note link

Toggle automatic creation of operations in target by typing operation names

## 11.14 Timing Diagram



### Add

Lifeline (State/Condition)

Lifeline (General value)

TickMark

Event/Stimulus

DurationConstraint

TimeConstraint

Message (Call)

Message (Reply)

Asynchronous Message (Call)

Note

Note Link

## 11.15 Use Case diagram



### Add

Package  
Actor  
UseCase

### Relationship

Association  
Generalization  
Include  
Extend

Note  
Note Link

## 11.16 XML Schema diagram



### Add

XSD TargetNamespace  
XSD Schema  
XSD Element (global)  
XSD Group  
XSD ComplexType  
XSD ComplexType (simpleContent)  
XSD SimpleType  
XSD List  
XSD Union  
XSD Enumeration  
XSD Attribute  
XSD AttributeGroup  
XSD Notation  
XSD Import

### Relationship

XSD Include  
XSD Redefine  
XSD Restriction  
XSD Extension  
XSD Substitution

Note

Note link

## 12 Menu 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. See [Creating, Opening, and Saving Projects](#)<sup>148</sup> and [Opening Projects from a URL](#)<sup>149</sup>.

### Reload

Reloads the current project and saves or discards the changes made since you opened the project file.

### Save

Saves the currently active modeling project using the currently active file name.

### Save as

Saves the currently active modeling project with a different name, or allows you to give the project a new name if this is the first time you save it.

### Save Copy As

Saves a copy of the currently active UModel project with a different file name.

### Save Diagram as Image

Opens the "Save as..." dialog box and allows you to save the currently active diagram as a .png file. Very large .png files, in the gigabyte range, can also be saved.

### Save all Diagrams as Images

Save all diagrams of the currently active project as .png files.

### Import from XMI File

Imports a previously exported XMI file. If the file was produced with UModel, then all extensions etc. will be retained.

### Export to XMI File

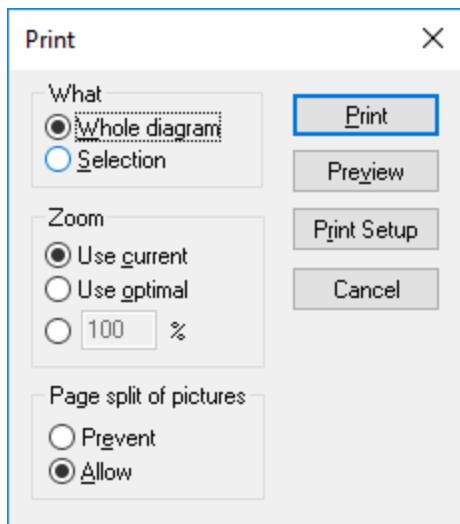
Exports the model as an XMI file. You can select the UML version, as well as the specific IDs that you want to export, see [XMI - XML Metadata Interchange](#)<sup>435</sup>.

### 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 the current diagram (or a selection on the diagram) as hard copy.



**Use current** retains the currently defined zoom factor of the modeling project. Selecting this option enables the "Page split of pictures" group. **Use optimal** scales the modeling project to fit the page size. You can also specify the zoom factor numerically. The **Prevent** option prevents modeling elements from being split over a page, and keeps them as one unit.

## Print all Diagrams

Opens the Print dialog box and prints out all UML diagrams contained in the current project file.

## Print Preview

Opens the same Print dialog box with the same settings as described above.

## Print Setup

Opens the Print Setup dialog box in which you can define the printer you want to use and the paper settings.

## Recent files

This section of the **File** menu lists up to four most recent files you have been working with.

## Exit

The **Exit** command exists in UModel. If any of your current files have unsaved changes, UModel will prompt you to save the changes.

## 12.2 Edit

Undo 

UModel has an unlimited number of "Undo" steps that you can use to retrace your modeling steps.

Redo 

The redo command allows you to redo previously undone commands. You can step backward and forward through the undo history using both these commands.

### Cut/Copy/Paste/Delete

These are the standard Windows text editing commands. You can use them not only for text but also for modeling elements, see [Renaming, Moving, and Copying Elements](#)<sup>107</sup>.

### Paste in Diagram only

Adds a "link" (or "view") of the copied element to the current diagram but not to the Model Tree, see [Renaming, Moving, and Copying Elements](#)<sup>107</sup>.

### Delete from Diagram only

Deletes the selected modeling elements from the currently active diagram. The deleted elements are not deleted from the modeling project and are available in the Model Tree tab. Note that this option is not available to delete properties or operations from a class, they can be selected and deleted there directly.

### Select all

Select all modeling elements of the currently active diagram. Equivalent to the **Ctrl+A** shortcut.

### Find

Allows you to search for specific text in the current window, see [Finding and Replacing Text](#)<sup>109</sup>.

Find Next  F3

Searches for the next occurrence of the same search string in the currently active window.

### Find Previous (Shift+F3)

Searches for the previous occurrence of the same search string in the currently active tab or diagram.

### Replace

Allows you to search and replace any modelling elements in the project, see [Finding and Replacing Text](#)<sup>109</sup>.

## Copy as Bitmap

Copies the currently active diagram to clipboard, from where you can paste it into the application of your choice.

## Copy Selection as Bitmap

Copies the currently selected diagram elements to the clipboard from where you can paste them into the application of your choice.

## 12.3 Project

### Check Project Syntax

Checks the UModel project syntax, see [Checking Project Syntax](#)<sup>167</sup>.

### Source Control

See [Source control systems](#)<sup>437</sup> for detailed information on source control servers and clients and how to use them.

### Import Source Directory

Opens the Import Source Directory wizard. For a specific example, see [Reverse Engineering \(from Code to Model\)](#)<sup>69</sup>.

### Import Source Project

Opens the Import Source Project wizard, see [Importing Source Code](#)<sup>187</sup>.

### Import Binary Types

Opens the Import Binary Types dialog box allowing you to import Java, C#, and VB binary files, see [Importing Java, C#, and VB.NET Binaries](#)<sup>199</sup>.

### Import XML Schema Directory

Opens the Import XML Schema Directory allowing you to import all XML Schemas in that directory and optionally all XML Schemas in any of the subfolders.

### Import XML Schema File

Opens the Import XML Schema File dialog box allowing you to import schema files, see [XML Schema Diagrams](#)<sup>417</sup>.

### Generate Sequence Diagrams from Code...

See [Generate Multiple Sequence Diagrams](#)<sup>363</sup>.

### Generate Code from Sequence Diagrams

UModel can create code from a sequence diagram which is linked to at least one operation. For more information, see [this section](#)<sup>365</sup>.

### Generate State Machine Code

UModel enables you to select one or more state machines in which code should be generated. For details, see [this topic](#)<sup>319</sup>.

## Merge Program Code from UModel Project / Overwrite Program Code from UModel Project

Updates program code from the model (assuming that your project is set up for code engineering, see [Generating Program Code](#)<sup>164</sup>). The name of this command can be either **Merge Program Code from UModel Project** or **Overwrite Program Code from UModel Project**, depending on the settings in the Synchronization Settings dialog box. By default, the Synchronization Settings dialog box opens every time when you run this command. For more information, see [Code Synchronization Settings](#)<sup>216</sup>.

## Merge UModel Project from Program Code / Overwrite UModel Project from Program Code

Updates the model (the UModel Project) from the program code. The name of this command can either be **Merge UModel Project from Program Code** or **Overwrite UModel Project from Program Code**, depending on the settings in the Synchronization Settings dialog box. By default, the Synchronization Settings dialog box opens every time when you run this command. For more information, see [Code Synchronization Settings](#)<sup>216</sup>.

## Project Settings

When generating program code into a UModel project, you may want to set or change [project settings](#)<sup>169</sup>.

## Synchronization Settings

Opens the Synchronization Settings dialog box, see [Code Synchronization Settings](#)<sup>216</sup>.

## Merge Project

Merges two UModel project files into one model. The first file you open is the one the second file will be merged into. Please see [Merging UModel projects](#)<sup>269</sup> for more information.

## Merge Project (3-way)

UModel supports the merging of multiple UModel projects that have been simultaneously edited by different developers, in [a 3-way project merge](#)<sup>269</sup>.

## Include Subproject

See [Including other UModel projects](#)<sup>158</sup>.

## Open Subproject Individually

Opens the selected subproject as a new project.

## Clear Messages

Clears the syntax check and code merging messages, warnings and errors from the [Messages Window](#)<sup>91</sup>.

**Note:** Errors are generally problems that must be fixed before code can be generated, or the model code can be updated during the code engineering process. Warnings can generally be deferred until later. Errors and warnings are generated by the syntax checker, the compiler for the specific language, the UModel parser that reads the newly generated source file, as well as during the import of XMI files.

## Generate Documentation

Generates documentation for the currently open project in HTML, Microsoft Word, and RTF formats, see [Generating UML documentation](#)<sup>278</sup>.

## List Elements not used in any Diagram

Creates a list of all elements not used in any diagram in the project, see [Checking Where and If Elements Are Used](#)<sup>111</sup>.

## List shared Packages

Lists all shared packages of the current project.

## List included Packages

Lists all include packages in the current project.

## 12.4 Layout

The commands of the Layout menu allow you to line up and align the elements of your modeling diagrams, see [Aligning and Resizing Modeling Elements](#)<sup>125</sup>.

### Align

The align command allows you to align modeling elements along their borders, or centers depending on the specific command you select.

### Space Evenly

This set of commands allow you to space selected elements evenly both horizontally and vertically.

### Make Same Size

This set of commands allow you to adjust the width and height of selected elements based on the active element.

### Line Up

This set of commands allow you to line up the selected elements vertically or horizontally.

### Line Style

This set of commands allow you to select the type of line used to connect the various modeling elements. The lines can be any type of dependency, association lines used in the various model diagrams.

### Autosize

This command resizes the selected elements to their respective optimal size(s).

### Autolayout all

This command arranges automatically the modeling elements on the diagram, using one of the options below.

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

### Reposition Text Labels

Repositions modeling element names (of the selected elements) to their default positions.

## 12.5      View

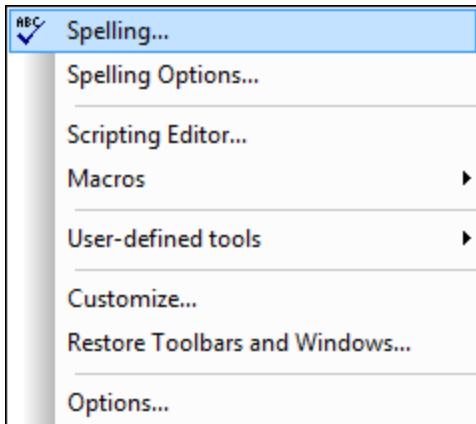
The commands available in this menu allow you to:

- Show or hide any of the UModel helper windows, see [UModel Graphical User Interface](#)<sup>77</sup>
- Define the sort criteria of elements inside the [Model Tree window](#)<sup>79</sup> and [Favorites window](#)<sup>84</sup>
- Define the grouping criteria of diagrams in the [Diagram Tree window](#)<sup>83</sup>
- Show or hide specific UML elements in the Favorites window and Model Tree window
- Define the zoom factor of the current diagram, see [Zooming into/out of Diagrams](#)<sup>129</sup>.

## 12.6 Tools

The commands available in this menu allow you to:

- [Customize](#)<sup>495</sup> the interface: define your own toolbars, keyboard shortcuts, menus, and macros.
- Restore toolbars and windows to their default state.
- Define the global program [settings/options](#)<sup>505</sup>.



### 12.6.1 User-defined Tools

Placing the cursor over the **User-defined Tools** command rolls out a sub-menu containing custom-made commands that use external applications. You can create these commands in the [Tools tab](#)<sup>498</sup> of the Customize dialog. Clicking one of these custom commands executes the action associated with this command.

The **User-Defined Tools | Customize** command opens the [Tools tab](#)<sup>498</sup> of the Customize dialog (in which you can create the custom commands that appear in the menu of the **User-Defined Tools** command.)

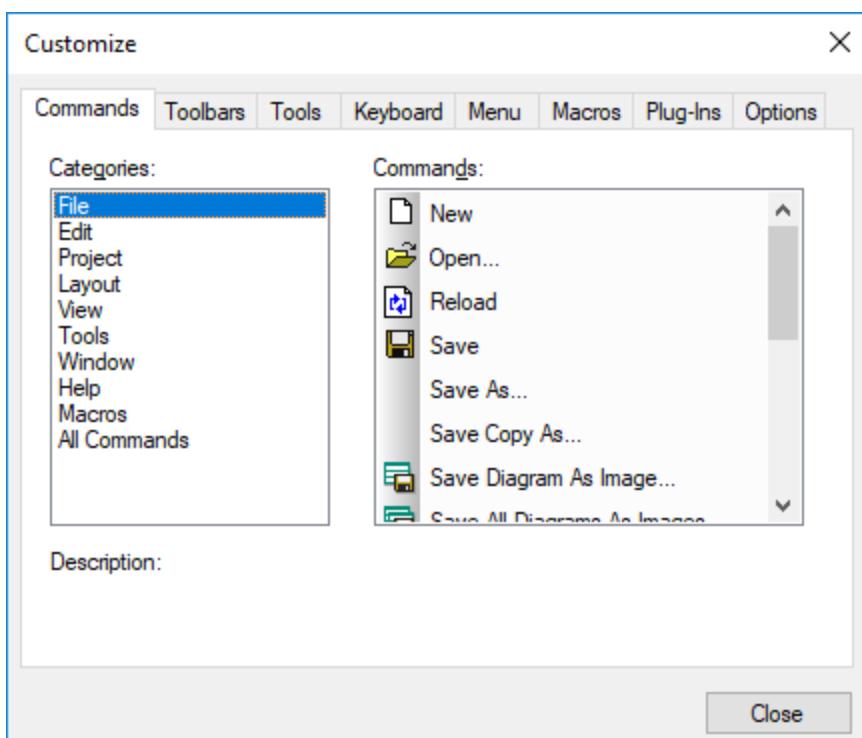
### 12.6.2 Customize

The **Customize** command displays a dialog box from where you can customize UModel to suit your personal needs. You can customize the following entities:

- [Commands](#)<sup>496</sup>
- [Toolbars](#)<sup>497</sup>
- [Tools](#)<sup>498</sup>
- [Keyboard](#)<sup>502</sup>
- [Menu](#)<sup>503</sup>
- [Options](#)<sup>504</sup>

## 12.6.2.1 Commands

The **Commands** tab allows you customize UModel menus or toolbars.



### To add a command to a toolbar or menu:

1. On the **Tools** menu, click **Customize**.
2. Select the command category in the **Categories** list box. The commands available appear in the **Commands** list box.
3. Click a command in the **Commands** list box and drag it to an existing menu or toolbar. An I-beam appears when you place the cursor over a valid position to drop the command.
4. Release the mouse button at the position you want to insert the command. A small button appears at the tip of mouse pointer when you drag a command. The check mark below the pointer means that the command cannot be dropped at the current cursor position. The check mark disappears whenever you can drop the command (over a toolbar or menu).

#### Notes:

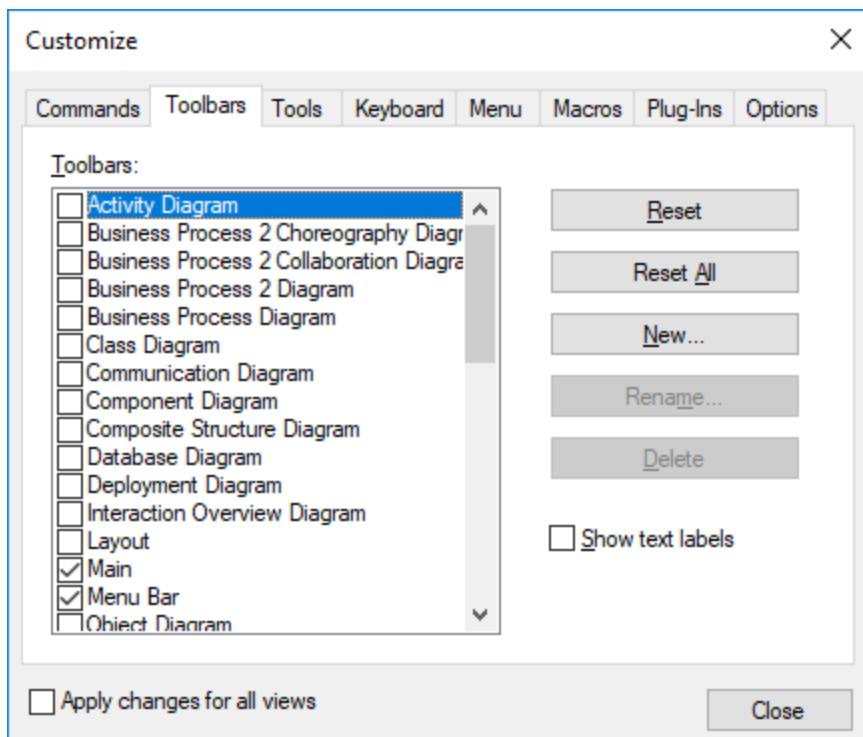
- Placing the cursor over a menu when dragging, opens it, allowing you to insert the command anywhere in the menu.
- Commands can be placed in menus or tool bars. If you created your own toolbar, you can populate it with your own commands/icons.
- You can also edit the commands in the [context menus](#) 503 (right-click anywhere to open the context menu), using the same method. Click the **Menu** tab and then select the specific context menu available in the Context Menus combo box.

**To delete a command or menu:**

1. On the **Tools** menu, click **Customize**.
2. Click the menu entry or icon you want to delete, and drag with the mouse.
3. Release the mouse button whenever the check mark icon appears below the mouse pointer. The command (or menu item) is deleted from the menu or tool bar.

### 12.6.2.2 Toolbars

The **Toolbars** tab allows you to activate or deactivate specific toolbars, as well as create your own specialized ones.



Toolbars contain symbols for the most frequently used menu commands. For each symbol, you get a brief "tool tip" explanation when the mouse cursor is directly over the item and the status bar shows a more detailed description of the command. You can drag the toolbars from their standard position to any location on the screen, where they appear as a floating window. Alternatively, you can also dock them to the left or right edge of the main window.

**To activate or deactivate a toolbar:**

- Click the check box to activate (or deactivate) the specific toolbar.

**To create a new toolbar:**

1. Click the **New...** button, and give the toolbar a name in the Toolbar name dialog box.
2. Add commands to the toolbar using the [Commands](#)<sup>496</sup> tab of the Customize dialog box.

**To reset the Menu Bar:**

1. Click the **Menu Bar** entry, and
2. Click the **Reset** button, to reset the menu commands to the state they were when installed.

**To reset all toolbar and menu commands:**

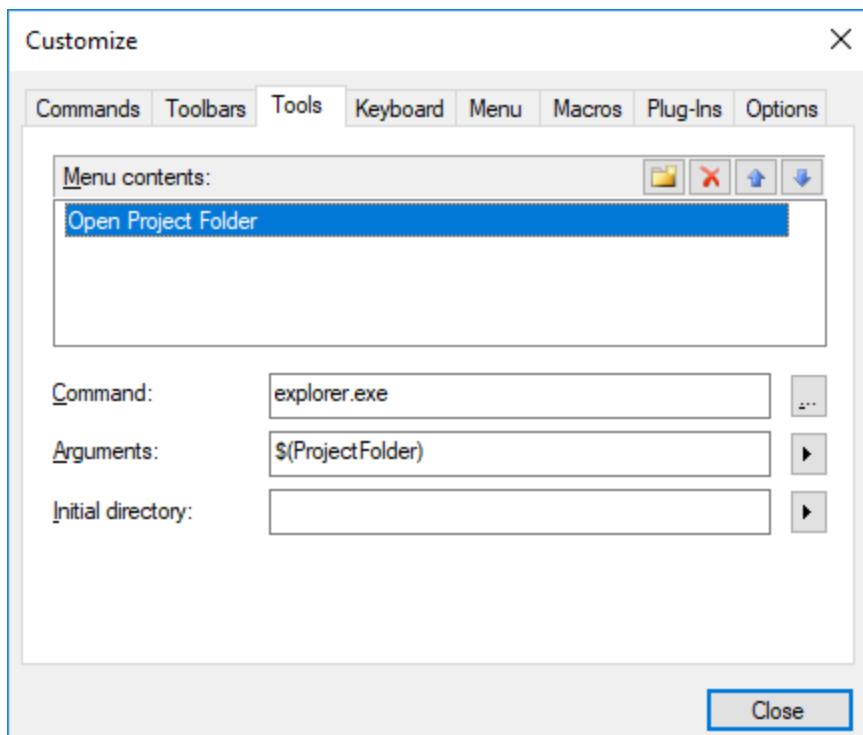
1. Click the **Reset All** button, to reset all the toolbar commands to the state they were when the program was installed. A prompt appears stating that all toolbars and menus will be reset.
2. Click **Yes** to confirm the reset.

The **Show text labels** option places explanatory text below toolbar icons when activated.

### 12.6.2.3 Tools

The **Tools** tab allows you to create custom menu commands that can start external tools directly from UModel. The custom menu commands that you define here appear under the menu **Tools | User-defined tools**. External tools can be programs included with Windows, such as Windows Explorer (**explorer.exe**), Notepad (**notepad.exe**), or other custom executables. You can optionally assign arguments to each user-defined tool and set the directory where the external tool should initialize (in order to look for relative file names).

For example, the configuration illustrated below adds a new menu command called "Open Project Folder". When run, this command will open the directory of the current UModel project in Windows Explorer.

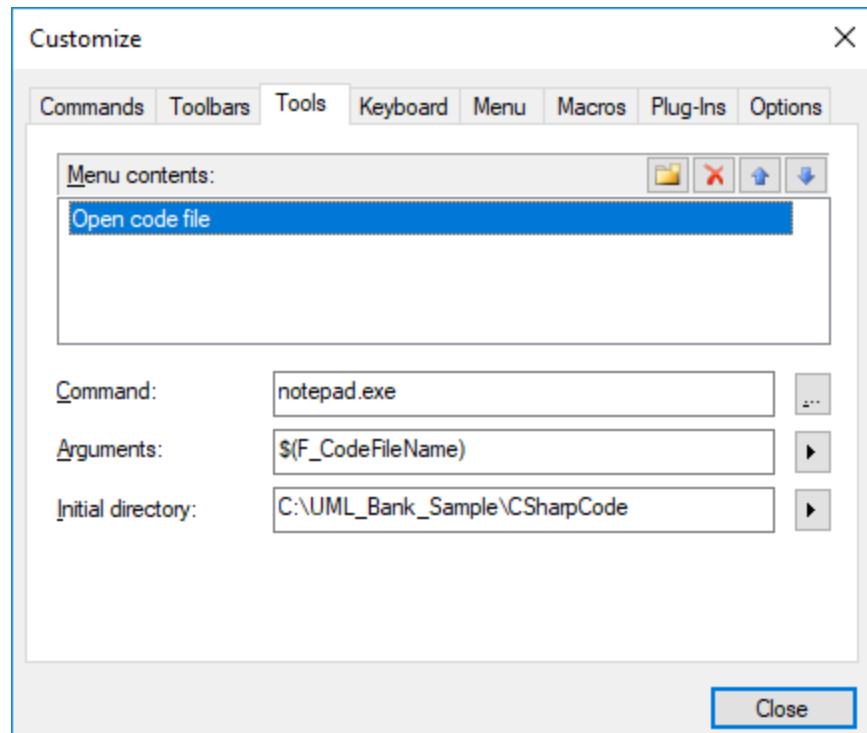


When an external tool takes arguments (like Windows Explorer in the example above), these can be entered in the Arguments input box. To supply multiple arguments, separate them with the space character. The values you can supply as arguments can be plain text (hard-coded values) or be selected with the button from a list of predefined UModel variables. You can use any of the following UModel predefined variables as arguments:

UModel predefined variable	Purpose
<i>Project File Name</i>	The file name of the active UModel project file, for example <b>Test.ump</b> .
<i>Project File Path</i>	The absolute file path of the active UModel project file, for example, <b>C:\MyDirectory\Test.ump</b> .
<i>Focused UML Data – Name</i>	The name of the currently focused UML element, for example, <b>Class1</b> .
<i>Focused UML Data – UML Qualified Name</i>	The qualified name of the currently focused UML element, for example, <b>Package1::Package2::Class1</b> .
<i>Focused UML Data – Code File Name</i>	The code file name of the currently focused UML class, interface or enumeration as shown in the Property window (relative to the realizing component), for example, <b>Class1.cs</b> or <b>MyNamespace\Class1.Java</b> .

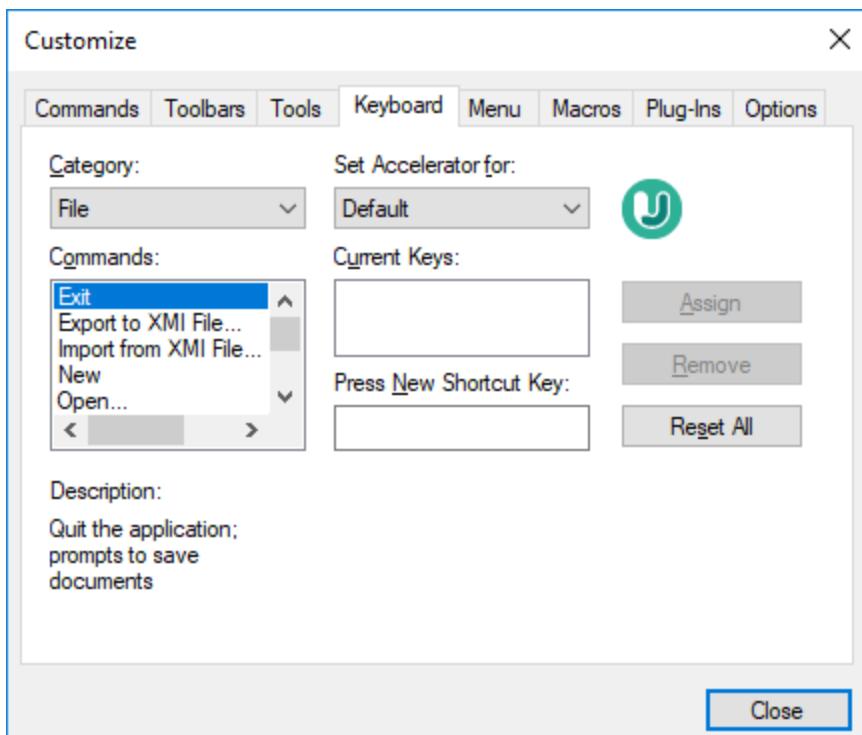
UModel predefined variable	Purpose
<i>Focused UML Data – Code File Path</i>	The code file path of the currently focused UML class, interface or enumeration as shown in the Property window, for example, C:\Temp\MySource\Class1.cs.
<i>Focused UML Data – Code Project File Name</i>	The file name of the code project to which the currently focused UML class, interface or enumeration belongs.  The code project file name can be relative to the UModel project file and is the same as shown in the Properties of the component, for example, C:\Temp\MySource\MyProject.vcproj or MySource\MyProject.vcproj.
<i>Focused UML Data – Code Project File Path</i>	The file path of the code project to which the currently focused UML class, interface or enumeration belongs, for example, C:\Temp\MySource\MyProject.vcproj.
<i>Project Folder</i>	The directory where the current UModel project is saved, for example, C:\Users\<user>\Documents\Altova\UModel2022\UModelExamples\.
<i>Temporary Folder</i>	The directory where the application's temporary files are saved, for example, C:\Users\<user>\AppData\Local\Temp.

In some cases, you may also need to enter a value in the **Initial Directory** input box. For example, the configuration below opens in Notepad the code file of the currently selected element on a diagram. (Note that, for this command to work, the element currently selected on the diagram must have a value (file name) defined in the **code file name** field of the [Properties Window](#)<sup>85</sup>, and that file must exist in C:\UML\_Bank\_Sample\CSharpCode directory).



## 12.6.2.4 Keyboard

The **Keyboard** tab allows you to define (or change) keyboard shortcuts for any command.



### To assign a new Shortcut to a command:

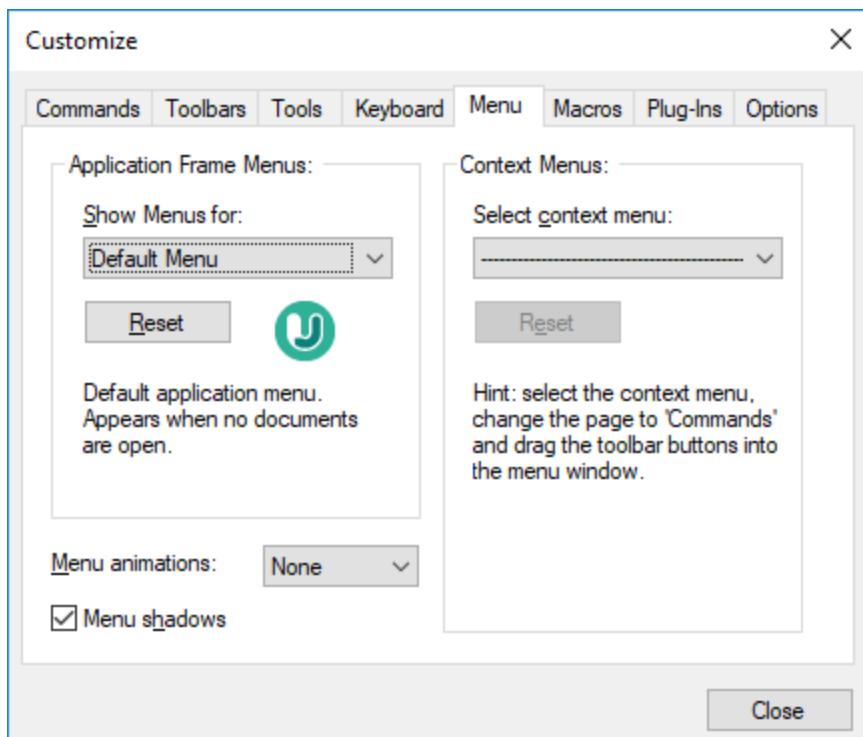
1. Select a value from the **Category** combo box.
2. Select the command you want to assign a new shortcut to, in the **Commands** list box.
3. Click inside the **Press New Shortcut Key** text box, and press the shortcut keys that are to activate the command. The shortcuts appear immediately in the text box. If the shortcut was assigned previously, then that function is displayed below the text box.
4. Click **Assign** to permanently assign the shortcut. The shortcut now appears in the **Current Keys** list box. (To clear this text box, press any of the control keys, **Ctrl**, **Alt** or **Shift**).

### To de-assign (delete) a shortcut:

1. Click the shortcut you want to delete in the **Current Keys** list box, and
2. Click the **Remove** button (which has now become active).
3. Click **Close** to confirm all the changes made in the Customize dialog box.

## 12.6.2.5 Menu

The **Menu** tab allows you to customize the menu bars as well as the context menus.



### Customizing menus

The **Default Menu** bar is the menu bar that is displayed when no project is open. The **UModel project** menu bar is the menu bar that is displayed when a project is open. Each menu bar can be customized separately, and customization changes made to one do not affect the other.

To customize a menu bar, select it from the **Show Menus For** drop-down list. Then click the **Commands** tab and drag commands from the **Commands** list box to the menu bar or into any of the menus.

### Deleting commands from menus and resetting the menu bars

To delete an entire menu or a command inside a menu, do the following:

1. Select from the **Show Menus for** drop-down list the menu bar that is to be customized.
2. With the Customize dialog open, select (i) the menu you want to delete from the application's menu bar, or (ii) the command you want to delete from one of these menus.
3. Either (i) drag the menu from the menu bar or the menu command from the menu, or (ii) right-click the menu or menu command and select **Delete**.

You can reset any menu bar to its original installation state by selecting it from the **Show Menus For** drop-down list and then clicking the **Reset** button.

## Customizing the application's context menus

Context menus are the menus that appear when you right-click certain objects in the application's interface. Each of these context menus can be customized by doing the following:

1. Select the context menu from the **Select context menu** drop-down list. This pops up the context menu.
2. Click the **Commands** tab.
3. Drag a command from the **Commands** list box into the context menu.
4. To delete a command from the context menu, right-click that command in the context menu, and select **Delete**. Alternatively, drag the command out of the context menu.

You can reset any context menu to its original installation state by selecting it in the **Select context menu** drop-down list and then clicking the **Reset** button.

## Menu shadows

Select the **Menu shadows** check box to give all menus shadows.

You can choose from among several menu animations if you prefer animated menus. The **Menu animations** drop-down list provides the following options:

- None (default)
- Unfold
- Slide
- Fade

## 12.6.2.6 Options

The **Options** tab allows you to set general environment settings.

When active, the **Show ScreenTips on toolbars** check box displays a tooltip label when the mouse pointer is placed over a toolbar button. The label contains a short description of the button function. If the **Show shortcut keys in ScreenTips** check box is selected, the tooltip label displays the associated keyboard shortcut, if one has been assigned.

When active, the **Large Icons** check box switches between the standard size icons, and larger versions of the icons.

## 12.6.3 Restore Toolbars and Windows

The **Restore Toolbars and Windows** command closes down UModel and re-starts it with the default settings. Before it closes down a dialog pops up asking for confirmation about whether UModel should be restarted.

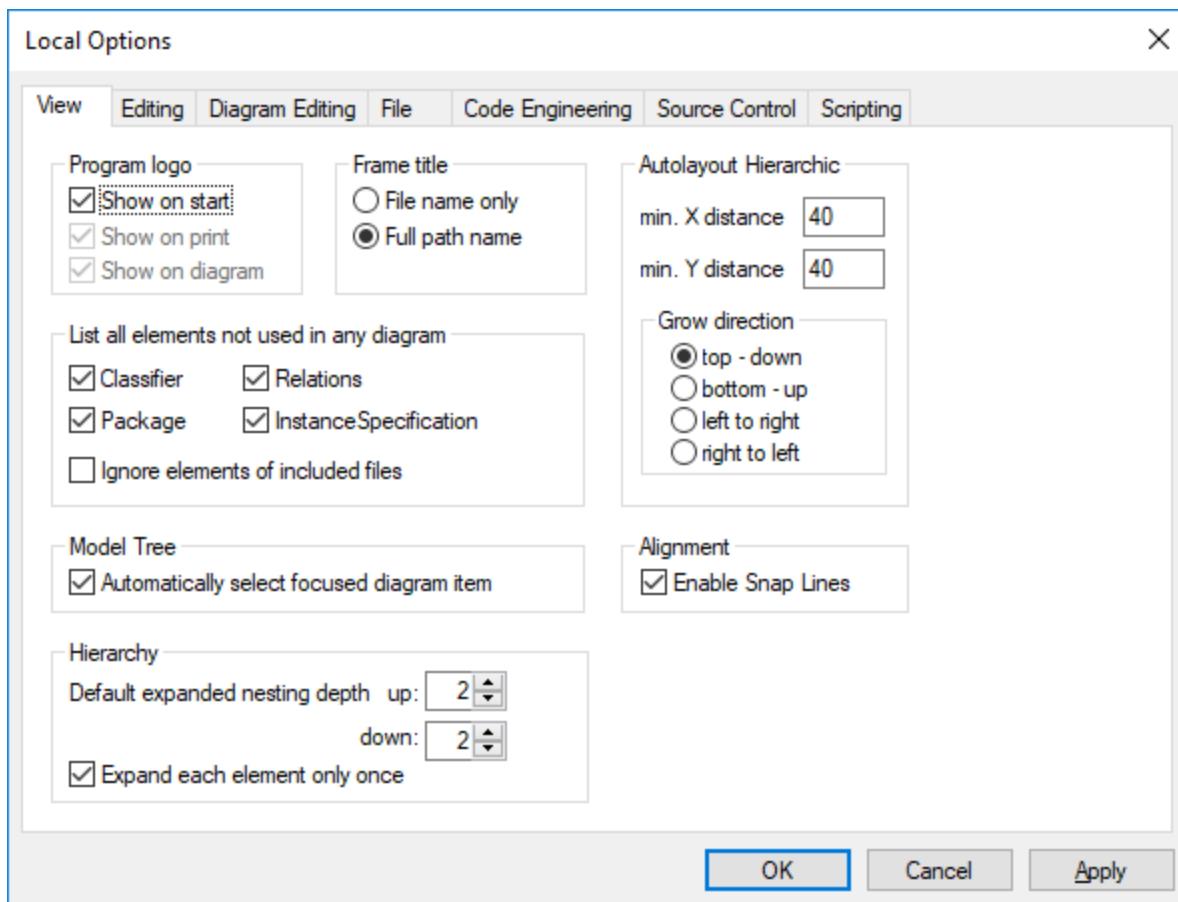
This command is useful if you have been resizing, moving, or hiding toolbars or windows, and would now like to have all the toolbars and windows as they originally were.

## 12.6.4 Options

Select the menu item **Tools | Options** to define your project options.

The **View** tab allows you to define:

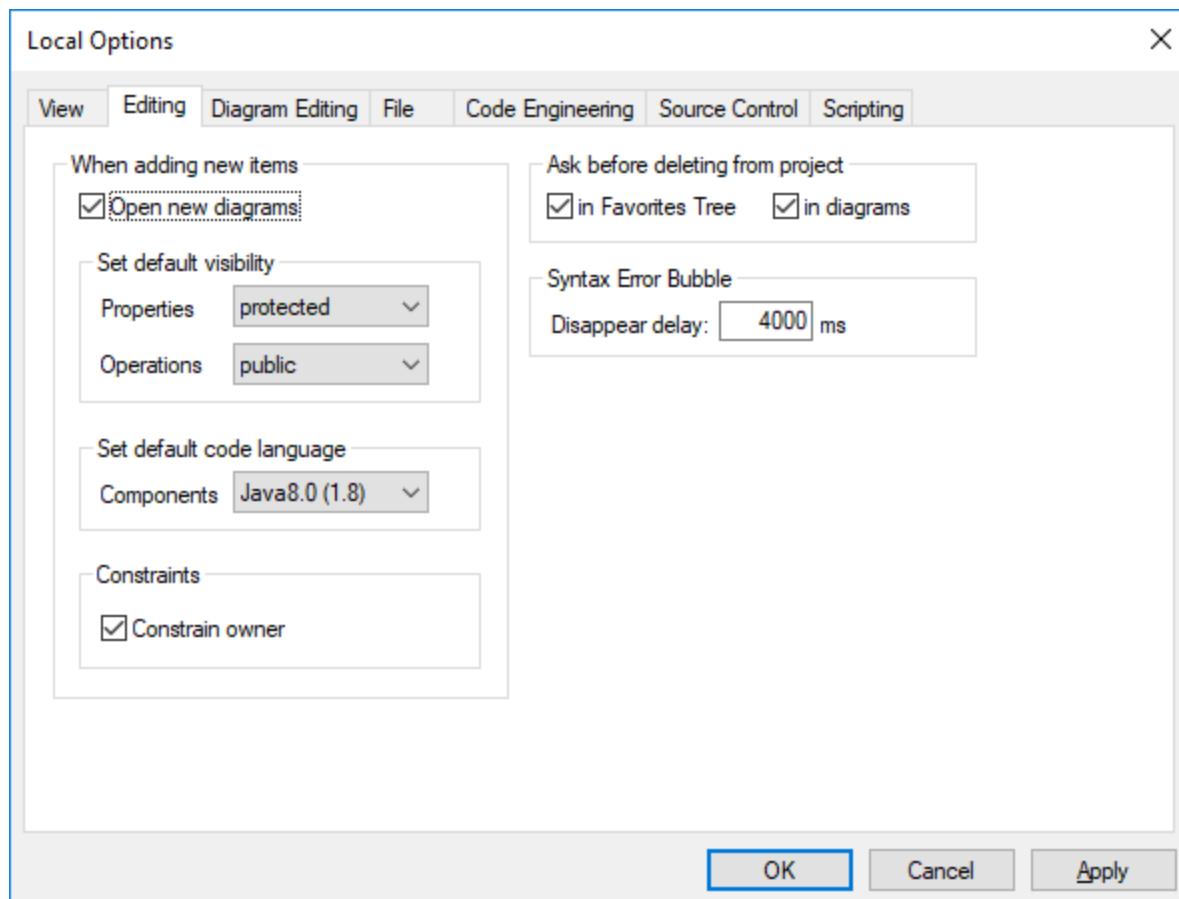
- Where the program logo should appear.
- The application title bar contents.
- The types of elements you want listed when using the "List elements not used in any diagram" context menu option in the Model Tree, or Favorites tab. You also have the option of ignoring elements contained in included files.
- If a selected element in a diagram is automatically selected/synchronized in the Model Tree.
- The default depth of the hierarchy view when using the **Show graph view** in the **Hierarchy** tab.
- The Autolayout Hierarchic settings, which allow you to define the nesting depth up and down in the hierarchy window.
- "Expand each element only once", only allows one of the same classifiers to be expanded in the same image/diagram.
- If you want snap lines to help you align elements when dragging in a diagram.



The **Editing** tab allows you to define:

- If a new Diagram created in the Model Tree tab, is also automatically opened in the main area.
- Default visibility settings when adding new elements - Properties or Operations.

- The default code language when a new component is added.
- If a newly added constraint, is to automatically constrain its owner as well.
- If a prompt should appear when deleting elements from a project, from the Favorites tab or in any of the diagrams. This prompt can be deactivated when deleting items there; this option allows you to reset the "prompt on delete" dialog box.
- The delay with which the syntax error pop-up message should be closed.

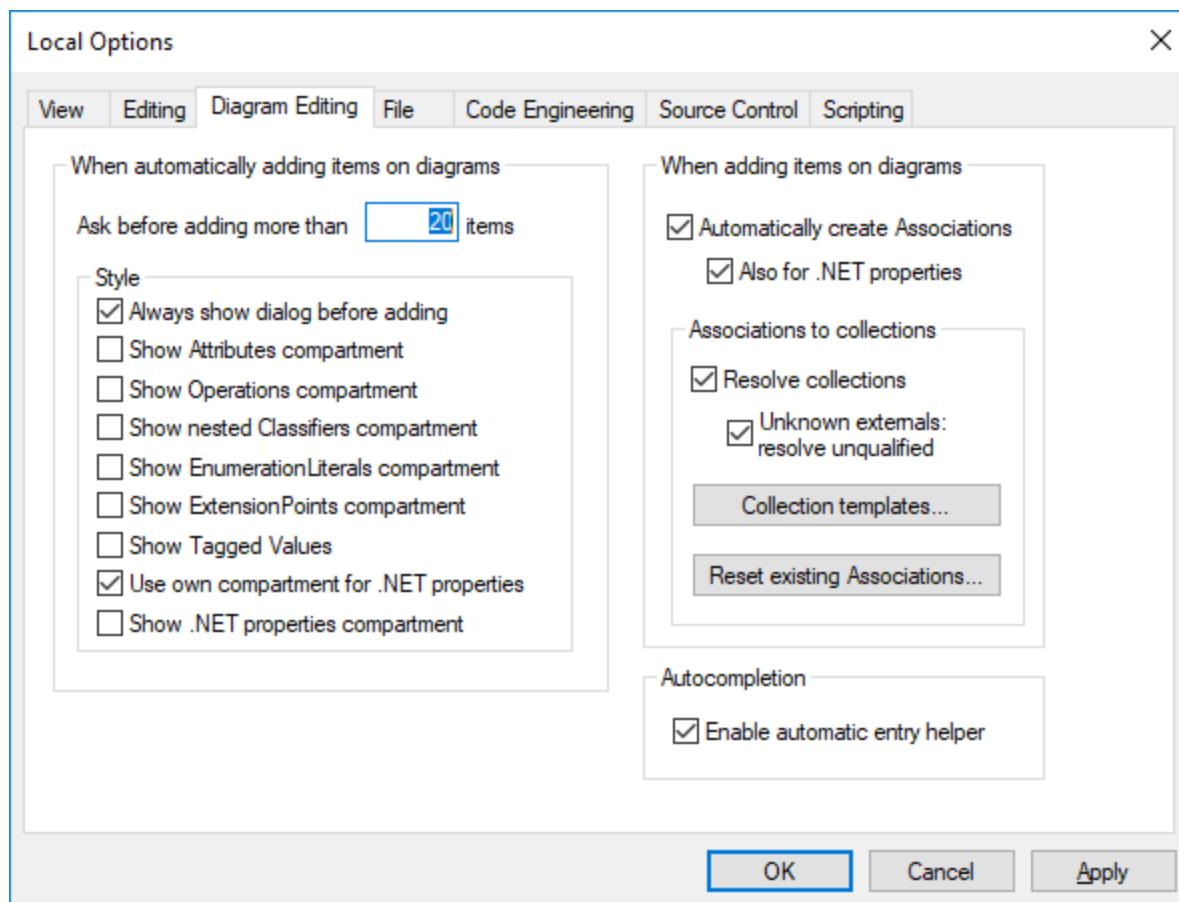


The **Diagram Editing** tab allows you to define:

- The number of items that can be automatically added to a diagram, before a prompt appears.
- The display of Styles when they are automatically added to a diagram.
- If Associations between modeling elements, are to be created automatically when items are added to a diagram.
- If the associations to collections are to be resolved.
- If templates from unknown externals are to be resolved as not fully qualified.
- or use preexisting Collection Templates, or define new ones.

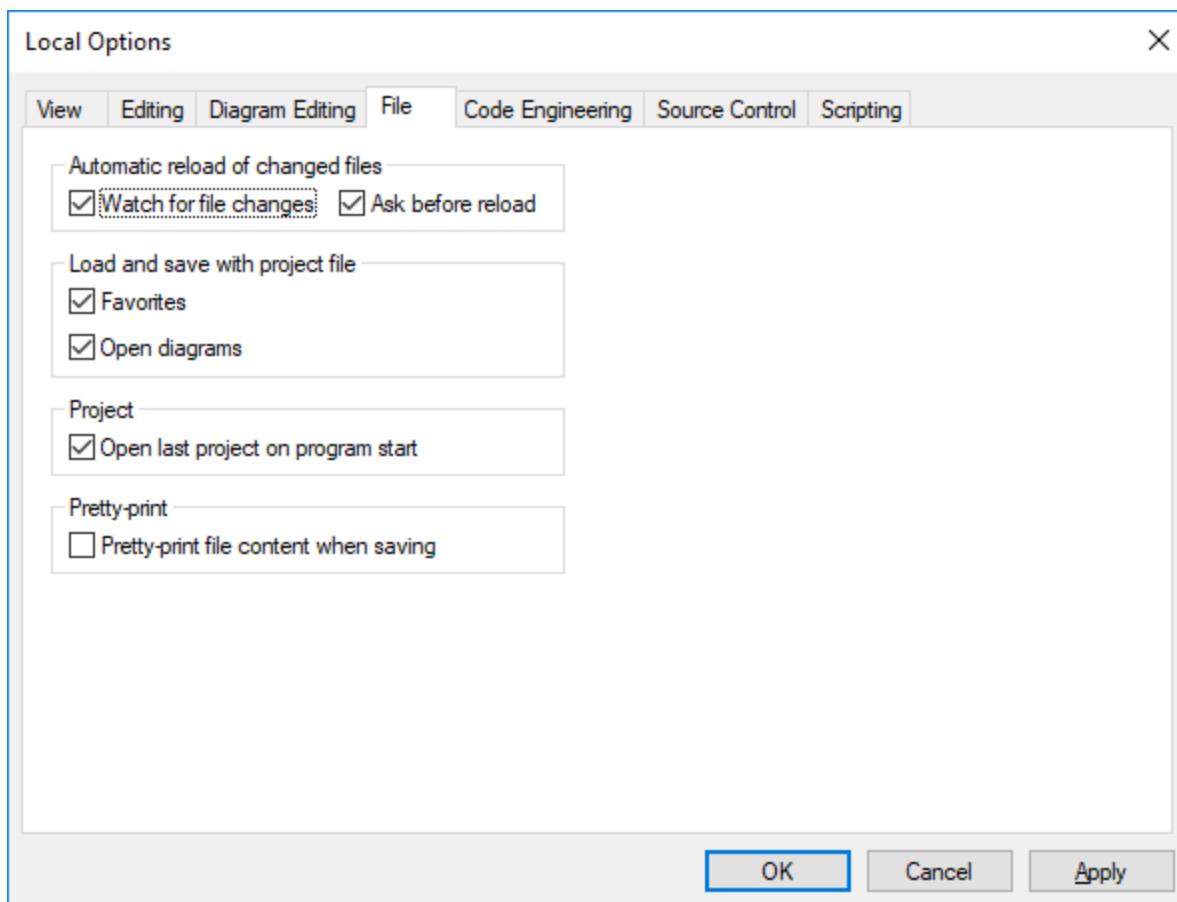
Collection Templates should be defined as fully qualified i.e. a.b.c.List. If the template has this namespace then UModel automatically creates a Collection Association. Exception: If the template belongs to the Unknown Externals package, and the option "Unknown externals: resolve unqualified", is enabled, then only the template name is considered (i.e. List instead of a.b.c.List).

- If the autocompletion window is to be available when editing attributes or operations in the class diagram.



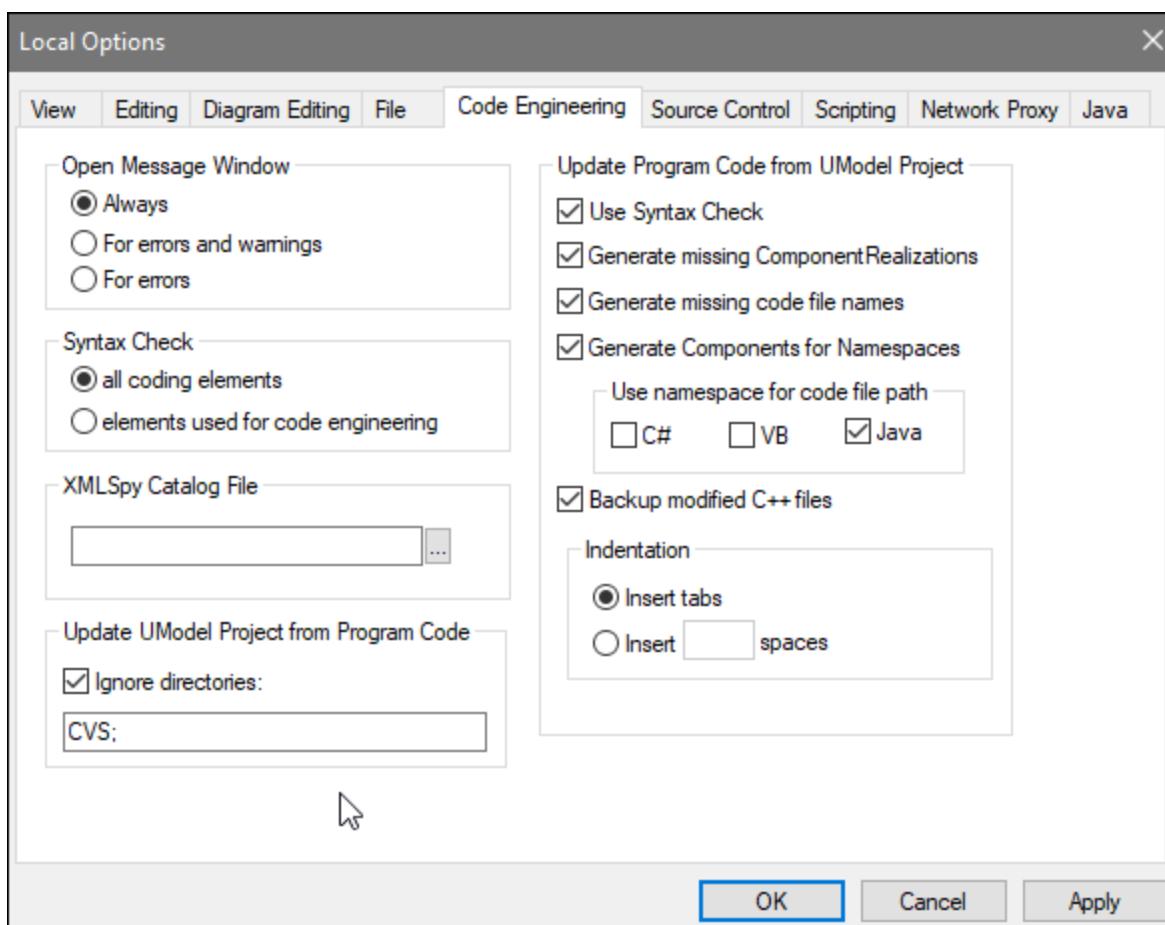
The **File** tab allows you to define:

- The actions performed when files are changed.
- If the contents of the Favorites tab are to be loaded and saved with the current project, as well as the any currently open diagrams.
- If the previously opened project is to automatically be opened when starting the application.
- If you want to structure the project file with CR/LF and tab indents in a pretty-print format.



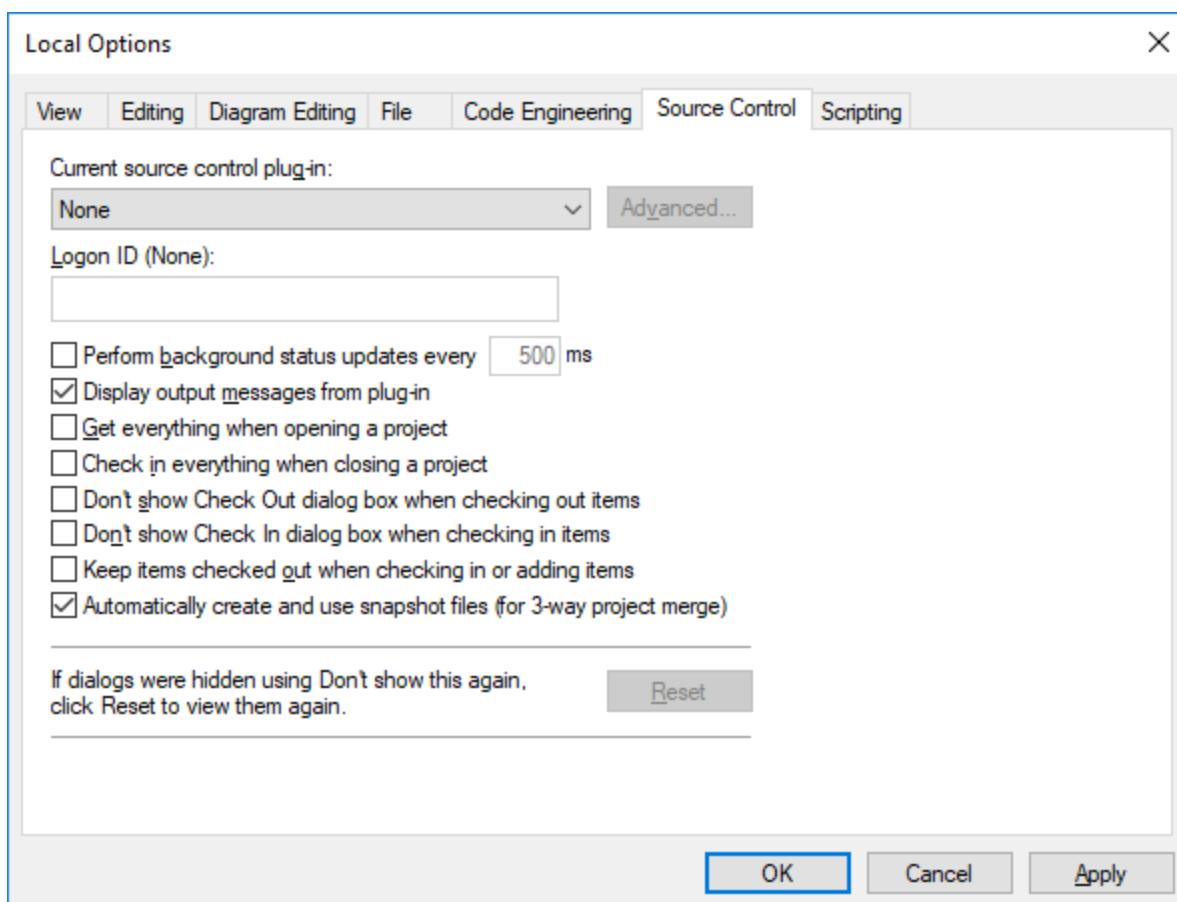
The **Code Engineering** tab allows you to define the following parameters:

- The circumstances under which the Message window will open.
- If **all coding elements** i.e. those contained in a Java / C# / VB namespace root, as well as those assigned to a Java / C# / VB component, are to be checked, or only **elements used for code engineering**, i.e. where "use for code engineering" check box is active, are to be checked.
- When updating program code if:
  - If a syntax check is to be performed.
  - If missing ComponentRealizations are to be automatically generated.
  - If missing code file names in the merged code are to be generated.
  - If namespaces are to be used in the code file path.
- The Indentation method used in the code, i.e. tabs or any number of spaces.
- The directories to be ignored when updating a UModel project from code, or directory. Separate the respective directories with a semicolon ;. Child directories of the same name are also ignored.
- The location of the XMLSpy Catalog File, **RootCatalog.xml**, which enables UModel as well as XMLSpy to retrieve commonly used schemas (as well as stylesheets and other files) from local user folders. This increases the overall processing speed, and enables users to work offline.
- You can also specify whether you want to back up modified C++ files.



The **Source Control** tab allows you to define:

- The current source control plug-in using the combo box. The **Advanced** button allows you to define the specific settings of the source control plug-in that you selected. These settings change depending on the source control plug-in that you use.
- The login ID for the source control provider.
- Specific settings check in/out settings.
- The **Reset** button is made available if you have checked/activated the "Don't show this again" option in one of the dialog boxes. The **Don't show this again** prompt is then reenabled.

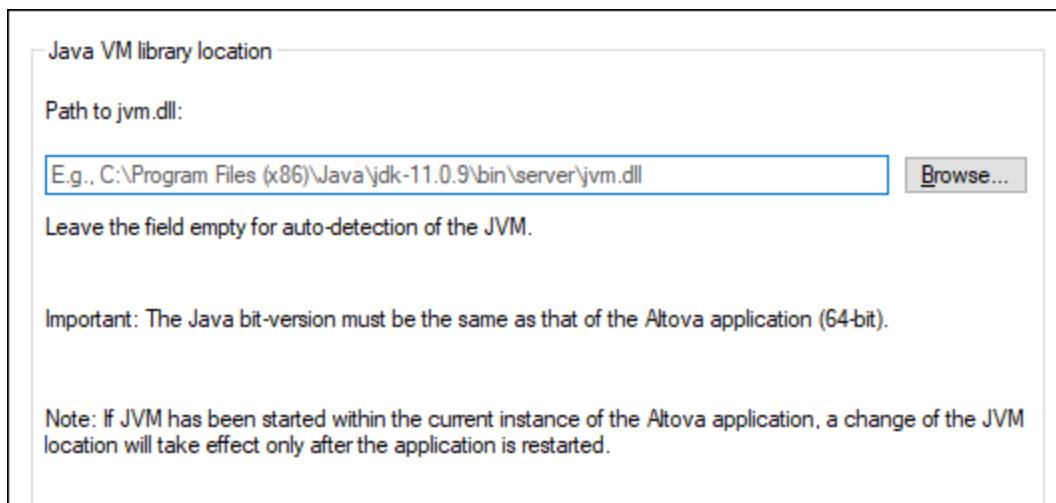


For information about the settings available in the **Network Proxy** tab, see [Network Proxy Settings](#)<sup>511</sup>. To find out more about Java VM settings, see [Java Virtual Machine Settings](#)<sup>510</sup>.

#### 12.6.4.1 Java Virtual Machine Settings

On the **Java** tab, you can optionally enter the path to a Java VM (Virtual Machine) on your file system. Note that adding a custom Java VM path is not always necessary. By default, UModel attempts to detect the Java VM path automatically by reading (in this order) the Windows registry and the JAVA\_HOME environment variable. The custom path added on this dialog box will take priority over any other Java VM path detected automatically.

You may need to add a custom Java VM path, for example, if you are using a Java virtual machine which does not have an installer and does not create registry entries (for example, Oracle's OpenJDK). You might also want to set this path if you need to override, for whatever reason, any Java VM path detected automatically by UModel.



Note the following:

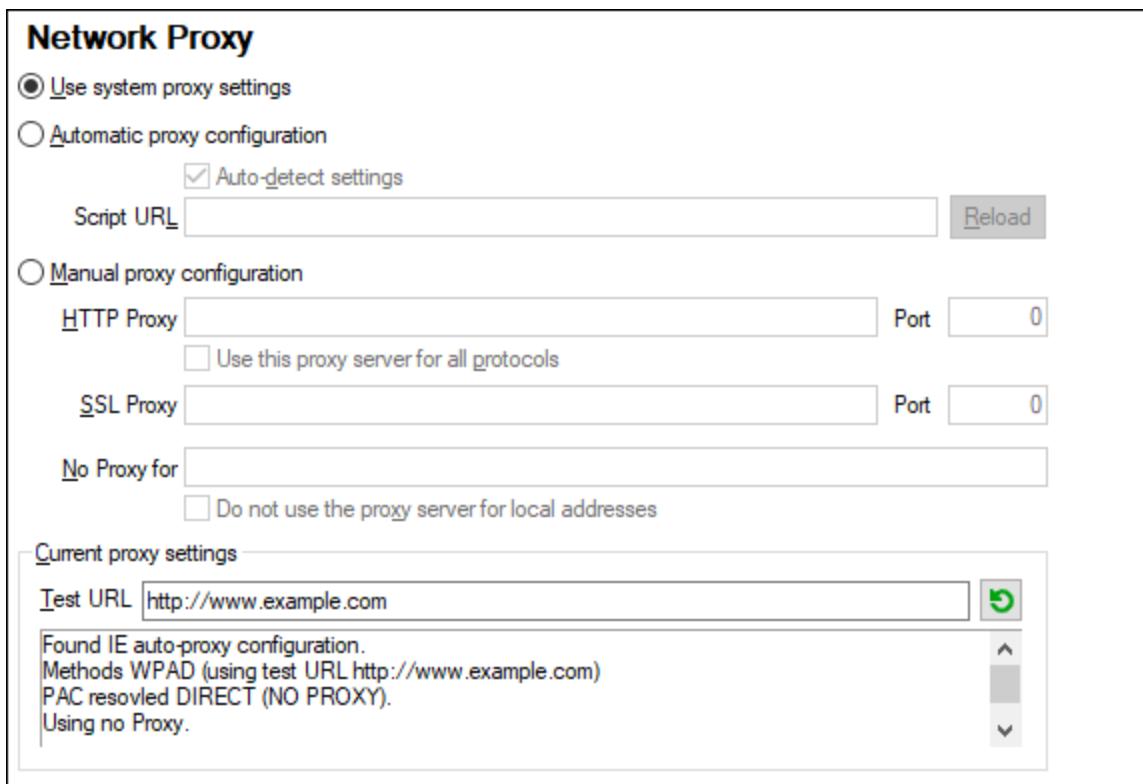
- The Java VM path is shared between Altova desktop (not server) applications. Consequently, if you change it in one application, it will automatically apply to all other Altova applications.
- The path must point to the **jvm.dll** file from the **\bin\server** or **\bin\client** directory, relative to the directory where the JDK was installed.
- The UModel platform (32-bit, 64-bit) must be the same as that of the JDK.
- After changing the Java VM path, you may need to restart UModel for the new settings to take effect.

This setting does not affect Java code generation and import. Note that the Java runtimes used for importing Java binaries into UModel can be configured separately, see [Adding Custom Java Runtimes](#)<sup>200</sup>.

## 12.6.4.2 Network Proxy Settings

The **Network Proxy** section enables you to configure custom proxy settings. These settings affect how the application connects to the Internet (for XML validation purposes, for example). By default, the application uses the system's proxy settings, so you should not need to change the proxy settings in most cases. If necessary, however, you can set an alternative network proxy using the options below.

**Note:** The network proxy settings are shared between all Altova MissionKit applications. Consequently, if you change the settings in one application, they will automatically affect all other applications.



## Use system proxy settings

Uses the Internet Explorer (IE) settings configurable via the system proxy settings. It also queries the settings configured with `netsh.exe winhttp`.

## Automatic proxy configuration

The following options are provided:

- *Auto-detect settings*: Looks up a WPAD script (`http://wpad.LOCALDOMAIN/wpad.dat`) via DHCP or DNS, and uses this script for proxy setup.
- *Script URL*: Specify an HTTP URL to a proxy-auto-configuration (.pac) script that is to be used for proxy setup.
- *Reload*: Resets and reloads the current auto-proxy-configuration. This action requires Windows 8 or newer, and may need up to 30s to take effect.

## Manual proxy configuration

Manually specify the fully qualified host name and port for the proxies of the respective protocols. A supported scheme may be included in the host name (for example: `http://hostname`). It is not required that the scheme is the same as the respective protocol if the proxy supports the scheme.

The following options are provided:

- *Use this proxy for all protocols*: Uses the host name and port of the HTTP Proxy for all protocols.

- *No Proxy for:* A semi-colon (;) separated list of fully qualified host names, domain names, or IP addresses for hosts that should be used without a proxy. IP addresses may not be truncated and IPv6 addresses have to be enclosed by square brackets (for example: [2606:2800:220:1:248:1893:25c8:1946]). Domain names must start with a leading dot (for example: .example.com).
- *Do not use the proxy server for local addresses:* If checked, adds <local> to the *No Proxy for* list. If this option is selected, then the following will not use the proxy: (i) 127.0.0.1, (ii) ::1, (iii) all host names not containing a dot character (.).

## Current proxy settings

Provides a verbose log of the proxy detection. It can be refreshed with the **Refresh** button to the right of the *Test URL* field (for example, when changing the test URL, or when the proxy settings have been changed).

- *Test URL:* A test URL can be used to see which proxy is used for that specific URL. No I/O is done with this URL. This field must not be empty if proxy-auto-configuration is used (either through *Use system proxy settings* or *Automatic proxy configuration*).

## 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.

### Forward

Whenever you change focus from a diagram window to another one, or navigate a hyperlink, UModel "remembers" this as an event. This command takes you "forward" in the history of such events. It is only meaningful and available if you already used the **Back** menu command (see below).

### Back

This command takes you back to the window that was previously in focus. This can be useful when you work with many diagram windows simultaneously, or when you navigate with hyperlinks, see [Hyperlinking Elements](#)<sup>113</sup>.

### Window list (1, 2)

This list shows all currently open diagram windows, and lets you quickly switch between them. You can also use the **Ctrl+Tab** or **Ctrl F6** keyboard shortcuts to cycle through the open windows.

## Windows

Displays a dialog box where you can layout or close multiple diagram windows simultaneously, see also [Diagram Pane](#)<sup>94</sup>.

## 12.8 Help

### ▼ Table of Contents

#### □ Description

Opens the onscreen help manual of UModel with the Table of Contents displayed in the left-hand-side pane of the Help window. The Table of Contents provides an overview of the entire Help document. Clicking an entry in the Table of Contents takes you to that topic.

### ▼ Index

#### □ Description

Opens the onscreen help manual of UModel with the Keyword Index displayed in the left-hand-side pane of the Help window. The index lists keywords and lets you navigate to a topic by double-clicking the keyword. If a keyword is linked to more than one topic, a list of these topics is displayed.

### ▼ Search

#### □ Description

Opens the onscreen help manual of UModel with the Search dialog displayed in the left-hand-side pane of the Help window. To search for a term, enter the term in the input field, and (i) press **Enter** or (ii) click **List Topics**. The Help system performs a full-text search on the entire Help documentation and returns a list of hits. Double-click any item to display that item.

---

### ▼ Software Activation

#### □ Description

After you download your Altova product software, you can license—or activate—it using either a free evaluation key or a purchased permanent license key.

- **Free evaluation license.** When you first start the software after downloading and installing it, the Software Activation dialog will pop up. In it is a button to request a free evaluation license. Enter your name, company, and e-mail address in the dialog that appears, and click **Request**. A license file is sent to the e-mail address you entered and should reach you in a few minutes. Save the license file to a suitable location. When you clicked **Request**, an entry field appeared at the bottom of the Request dialog. This field takes the path to the license file. Browse for or enter the path to the license file, and click **OK**. (In the Software Activation dialog, you can also click **Upload a New License** to access a dialog in which the path to the license file is entered.) The software will be unlocked for a period of 30 days.
- **Permanent license key.** The Software Activation dialog contains a button to purchase a permanent license key. Clicking this button takes you to Altova's online shop, where you can purchase a permanent license key for your product. Your license will be sent to you by e-mail in the form of a license file, which contains your license-data. There are three types of permanent license: installed, concurrent user, and named user. An *installed license* unlocks the software on a single computer. If you buy an *installed license* for  $n$  computers, then the license allows use of the software on up to  $n$  computers. A *concurrent-user license* for  $n$

concurrent users allows **n** users to run the software concurrently. (The software may be installed on **10N** computers.) A *named-user license* authorizes a *specific user* to use the software on up to 5 different computers. To activate your software, click **Upload a New License**, and, in the dialog that appears, browse for or enter the path to the license file, and click **OK**.

**Note:** For multi-user licenses, each user will be prompted to enter his or her own name.

***Your license email and the different ways to license (activate) your Altova product***

The license email that you receive from Altova will contain your license file as an attachment. The license file has a **.altova\_licenses** file extension.

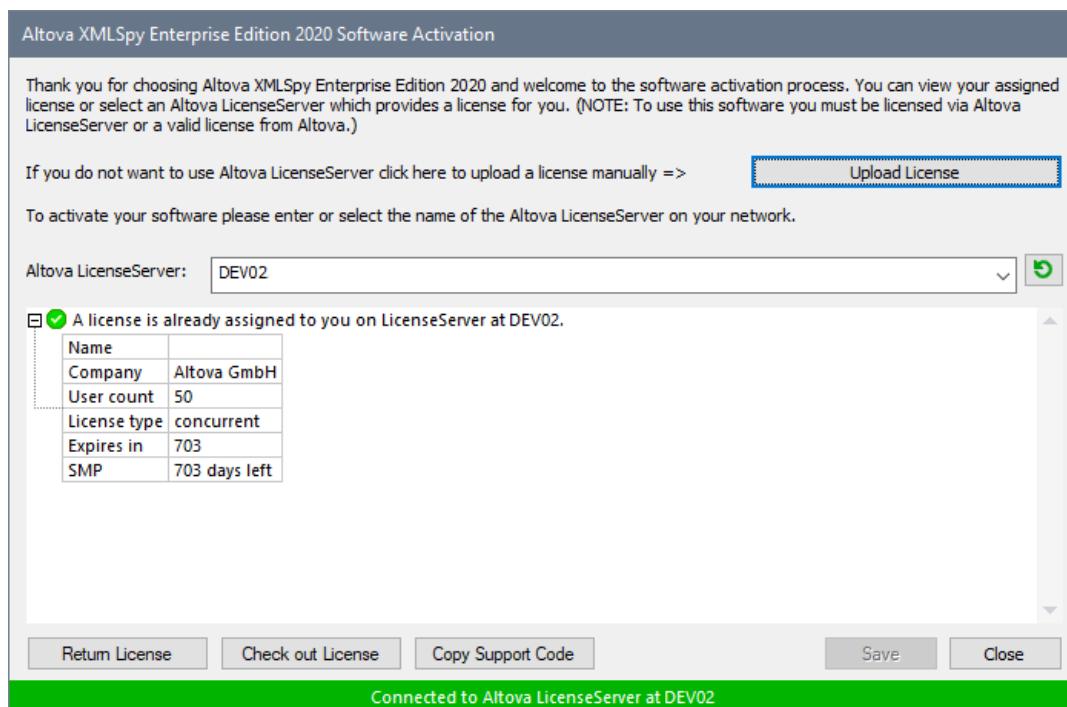
To activate your Altova product, you can do one of the following:

- Save the license file (**.altova\_licenses**) to a suitable location, double-click the license file, enter any requested details in the dialog that appears, and finish by clicking **Apply Keys**.
- Save the license file (**.altova\_licenses**) to a suitable location. In your Altova product, select the menu command **Help | Software Activation**, and then **Upload a New License**. Browse for or enter the path to the license file, and click **OK**.
- Save the license file (**.altova\_licenses**) to any suitable location, and upload it from this location to the license pool of your Altova LicenseServer. You can then either: (i) acquire the license from your Altova product via the product's Software Activation dialog (see below), or (ii) assign the license to the product from Altova LicenseServer. *For more information about licensing via LicenseServer, read the rest of this topic.*

The Software Activation dialog (screenshot below) can be accessed at any time by clicking the **Help | Software Activation** command.

You can activate the software by either:

- *Registering the license in the Software Activation dialog.* In the dialog, click **Upload a New License**, and browse for and select the license file. Click **OK** to confirm the path to the license file and to confirm any data you entered (your name in the case of multi-user licenses). Finish by clicking **Save**.
- *Licensing via an Altova LicenseServer on your network:* To acquire a license via an Altova LicenseServer on your network, click **Use Altova LicenseServer**, located at the bottom of the Software Activation dialog. Select the machine on which the LicenseServer you want to use has been installed. Note that the auto-discovery of License Servers works by means of a broadcast sent out on the LAN. As these broadcasts are limited to a subnet, License Server must be on the same subnet as the client machine for auto-discovery to work. If auto-discovery does not work, then type in the name of the server. The Altova LicenseServer must have a license for your Altova product in its license pool. If a license is available in the LicenseServer pool, this is indicated in the Software Activation dialog (see screenshot below showing the dialog in Altova XMLSpy). Click **Save** to acquire the license.



After a machine-specific (aka installed) license has been acquired from a LicenseServer, it cannot be returned to the LicenseServer for a period of seven days. After that time, you can return the machine license to LicenseServer (click **Return License**) so that this license can be acquired from LicenseServer by another client. (A LicenseServer administrator, however, can unassign an acquired license at any time via the administrator's Web UI of LicenseServer.) Note that the returning of licenses applies only to machine-specific licenses, not to concurrent licenses.

#### Check out license

You can check out a license from the license pool for a period of up to 30 days so that the license is stored on the product machine. This enables you to work offline, which is useful, for example, if you wish to work in an environment where there is no access to your Altova LicenseServer (such as when your Altova product is installed on a laptop and you are traveling). While the license is checked out, LicenseServer displays the license as being in use, and the license cannot be used by any other machine. The license automatically reverts to the checked-in state when the check-out period ends. Alternatively, a checked-out license can be checked in at any time via the **Check in** button of the Software Activation dialog.

To check out a license, do the following: (i) In the Software Activation dialog, click **Check out License** (see screenshot above); (ii) In the License Check-out dialog that appears, select the check-out period you want and click **Check out**. The license will be checked out. After checking out a license, two things happen: (i) The Software Activation dialog will display the check-out information, including the time when the check-out period ends; (ii) The **Check out License** button in the dialog changes to a **Check In** button. You can check the license in again at any time by clicking **Check In**. Because the license automatically reverts to the checked-in status after the check-out period elapses, make sure that the check-out period you select adequately covers the period during which you will be working offline.

**Note:** For license check-outs to be possible, the check-out functionality must be enabled on LicenseServer. If this functionality has not been enabled, you will get an error message to this effect when you try to check out. In this event, contact your LicenseServer administrator.

[Copy Support Code](#)

Click **Copy Support Code** to copy license details to the clipboard. This is the data that you will need to provide when requesting support via the [online support form](#).

Altova LicenseServer provides IT administrators with a real-time overview of all Altova licenses on a network, together with the details of each license, as well as client assignments and client usage of licenses. The advantage of using LicenseServer therefore lies in administrative features it offers for large-volume Altova license management. Altova LicenseServer is available free of cost from the [Altova website](#). For more information about Altova LicenseServer and licensing via Altova LicenseServer, see the [Altova LicenseServer documentation](#).

▼ Order Form

[Description](#)

When you are ready to order a licensed version of the software product, you can use either the **Purchase a Permanent License Key** button in the Software Activation dialog (*see previous section*) or the **Order Form** command to proceed to the secure Altova Online Shop.

▼ Registration

[Description](#)

Opens the Altova Product Registration page in a tab of your browser. Registering your Altova software will help ensure that you are always kept up to date with the latest product information.

▼ Check for Updates

[Description](#)

Checks with the Altova server whether a newer version than yours is currently available and displays a message accordingly.

---

▼ Support Center

[Description](#)

A link to the Altova Support Center on the Internet. The Support Center provides FAQs, discussion forums where problems are discussed, and access to Altova's technical support staff.

▼ FAQ on the Web

[Description](#)

A link to Altova's FAQ database on the Internet. The FAQ database is constantly updated as Altova support staff encounter new issues raised by customers.

▼ Download Components and Free Tools

  □ Description

A link to Altova's Component Download Center on the Internet. From here you can download a variety of companion software to use with Altova products. Such software ranges from XSLT and XSL-FO processors to Application Server Platforms. The software available at the Component Download Center is typically free of charge.

▼ UModel on the Internet

  □ Description

A link to the [Altova website](#) on the Internet. You can learn more about UModel and related technologies and products at the [Altova website](#).

▼ About UModel

  □ Description

Displays the splash window and version number of your product. If you are using the 64-bit version of UModel, this is indicated with the suffix (x64) after the application name. There is no suffix for the 32-bit version.

## 13 SPL Reference

This section gives an overview of SPL (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 ...\\UModel\\spl 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 Basic SPL structure

An SPL file contains literal text to output, interspersed with code generator instructions.

Code generator instructions are enclosed in square brackets '[' and ']'. Multiple statements can be included in a bracket pair. Additional statements have to be separated by a new line or a colon ':'.

Valid examples are:

```
[$x = 42  
$x = $x + 1]
```

or

```
[$x = 42: $x = $x + 1]
```

### Adding text to files

Text not enclosed by [ and ], is written directly to the current output file.

To output literal square brackets, escape them with a backslash: \\[ and \\]; to output a backslash use \\\.

### Comments

Comments inside an instruction block always begin with a ' character, and terminate on the next line, or at a block close character ].

## 13.2 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](#)<sup>534</sup> statement

Variable types are declared by first assignment:

```
[$x = 0]
```

x is now an integer.

```
[$x = "teststring"]
```

x is now treated as a string.

### Strings

String constants are always enclosed in double quotes, like in the example above. \n and \t inside double quotes are interpreted as newline and tab, \" is a literal double quote, and \\ is a backslash. String constants can also span multiple lines.

String concatenation uses the & character:

```
[$BasePath = $outputpath & "/" & $JavaPackageDir]
```

### Objects

Objects represent the information contained in the UModel project. Objects have **properties**, which can be accessed using the . operator. It is not possible to create new objects in SPL (they are predefined by the code generator, derived from the input), but it is possible to assign objects to variables.

Example:

```
class [= $class.Name]
```

This example outputs the word "class", followed by a space and the value of the **Name** property of the **\$class** object.

The following table shows the relationship between UML elements their SPL equivalents along with a short description.

## Predefined variables

UML element	SPL property	Multiplicity	UML Attribute / Association	UModel Attribute / Association	Description
BehavioralFeature	isAbstract		isAbstract:Boolean		
BehavioralFeature	raisedException	*	raisedException:Type		
BehavioralFeature	ownnedParameter	*	ownnedParameter:Parameter		
BehavioredClassifier	interfaceRealization	*	interfaceRealization:InterfaceRealization		
Class	ownnedOperation	*	ownnedOperation:Operation		
Class	nestedClassifier	*	nestedClassifier:Classifier		
Classifier	namespace	*		namespace:Package	packages with code language <<namespace>> set
Classifier	rootNamespace	*		project root namespace:String	VB only - root namespace
Classifier	generalization	*	generalization:Generalization		
Classifier	isAbstract		isAbstract:Boolean		
ClassifierTemplateParameter	constrainingClassifier	*	constrainingClassifier		
Comment	body		body:String		
DataType	ownnedAttribute	*	ownnedAttribute:Property		
DataType	ownnedOperation	*	ownnedOperation:Operation		
Element	kind			kind:String	
Element	owner	0..1	owner:Element		
Element	appliedStereotype	*		appliedStereotype:StereotypeApplication	applied stereotypes

UML element	SPL property	Multiplicity	UML Attribute / Association	UModel Attribute / Association	Description
Element	ow nedComment	*	ow nedComment:Comment		
ElementImport	importedElement	1	importedElement:PackagableElement		
Enumeration	ow nedLiteral	*	ow nedLiteral:EnumerationLiteral		
Enumeration	nestedClassifier	*		nestedClassifier::Classifier	
Enumeration	interfaceRealization	*		interfaceRealization:Interface	
EnumerationLiteral	ow nedAttribute	*		ow nedAttribute:Property	
EnumerationLiteral	ow nedOperation	*		ow nedOperation:Operation	
EnumerationLiteral	nestedClassifier	*		nestedClassifier::Classifier	
Feature	isStatic		isStatic:Boolean		
Generalization	general	1	general:Classifier		
Interface	ow nedAttribute	*	ow nedAttribute:Property		
Interface	ow nedOperation	*	ow nedOperation:Operation		
Interface	nestedClassifier	*	nestedClassifier::Classifier		
InterfaceRealization	contract	1	contract:Interface		
MultiplicityElement	lowerValue	0..1	lowerValue:ValueSpecification		
MultiplicityElement	upperValue	0..1	upperValue:ValueSpecification		
NamedElement	name		name:String		
NamedElement	visibility		visibility:VisibilityKind		
NamedElement	isPublic			isPublic:Boolean	visibility <public>
NamedElement	isProtected			isProtected:Boolean	visibility <protected>
NamedElement	isPrivate			isPrivate:Boolean	visibility <private>

UML element	SPL property	Multiplicity	UML Attribute / Association	UModel Attribute / Association	Description
NamedElement	isPackage	*		isPackage:Boolean	visibility <package>
NamedElement	namespacePrefix	*		namespacePrefix:String	XSD only - namespace prefix when exists
NamedElement	parseableName	*		parseableName:String	CSharp, VB only - name with escaped keywords (@)
Namespace	elementImport	*	elementImport:ElementImport		
Operation	ownedReturnParameter	0..1		ownedReturnParameter:Parameter	parameter with direction return set
Operation	type	0..1		type	type of parameter with direction return set
Operation	ownedOperationParameter	*		ownedOperationParameter:Parameter	all parameters excluding parameter with direction return set
Operation	implementedInterface	1		implementedInterface:Interface	CSharp only - the implemented interface
Operation	ownedOperationImplementations	*		implementedOperation:OperationImplementation	VB only - the implemented interfaces/operations
OperationImplementation	implementedOperationOwner	1		implementedOperationOwner:Interface	interface implemented by the operation
OperationImplementation	implementedOperationName			name:String	name of the implemented operation
OperationImplementation	implementedOperationParseableName			parseableName:String	name of the implemented operation with escaped keywords
Package	namespace	*		namespace:Package	packages with code language <<namespace>> set
PackageableElement	owningPackage	0..1		owningPackage	set if owner is a package

UML element	SPL property	Multiplicity	UML Attribute / Association	UModel Attribute / Association	Description
PackageableElement	owningNamespace	0..1		owningNamespace	owning package with code language <<namespace>> set
Parameter	direction		direction: ParameterDirectionKind		
Parameter	isIn			isIn:Boolean	direction <in>
Parameter	isInOut			isInOut:Boolean	direction <inout>
Parameter	isOut			isOut:Boolean	direction <out>
Parameter	isReturn			isReturn:Boolean	direction <return>
Parameter	isVarArgList			isVarArgList:Boolean	true if parameter is a variable argument list
Parameter	defaultValue	0..1	defaultValue:Value Specification		
Property	defaultValue	0..1	defaultValue:Value Specification		
RedefinableElement	isLeaf		isLeaf:Boolean		
Slot	name			name:String	name of the defining feature
Slot	values	*	value:ValueSpecification		
Slot	value			value:String	value of the first value specification
StereotypeApplication	name			name:String	name of applied stereotype
StereotypeApplication	taggedValue	*		taggedValue:Slot	first slot of the instance specification
StructuralFeature	isReadOnly		isReadOnly		
StructuredClassifier	ownedAttribute	*	ownedAttribute:Property		
TemplateBinding	signature	1	signature:TemplateSignature		
TemplateBinding	parameterSubstitution	*	parameterSubstitution:TemplateParameterSubstitution		

UML element	SPL property	Multiplicity	UML Attribute / Association	UModel Attribute / Association	Description
TemplateParameter	paramDefault			paramDefault:String	template parameter default value
TemplateParameter	ownnedParameteredElement	1	ownnedParameteredElement:ParameterableElement		
TemplateParameter Substitution	parameterSubstitution			parameterSubstitution:String	Java only - code wildcard handling
TemplateParameter Substitution	parameterDimensionCount			parameterDimensionCount:Integer	code dimension count of the actual parameter
TemplateParameter Substitution	actual	1	ownnedActual:ParameterableElement		
TemplateParameter Substitution	formal	1	formal:TemplateParameter		
TemplateSignature	template	1	template:TemplateableElement		
TemplateSignature	ownnedParameter	*	ownnedParameter:TemplateParameter		
TemplateableElement	isTemplate			isTemplate:Boolean	true if template signature set
TemplateableElement	ownnedTemplateName	0..1	ownnedTemplateName:TemplateName		
TemplateableElement	templateBinding	*	templateBinding:TemplateBinding		
Type	typeName	*		typeName:PackagableElement	qualified code type names
TypedElement	type	0..1	type>Type		
TypedElement	postTypeModifier			postTypeModifier:String	postfix code modifiers
ValueSpecification	value			value:String	string value of the value specification

### Adding a prefix to attributes of a class during code generation

You might need to prefix all new attributes with the "m\_" characters in your project.

All new coding elements are written using the SPL templates. For example, if you open **UModel\SPLOC#\Java\Default\Attribute.spl**, you can change the way the name is written. Namely, you can replace

```
write $Property.name
```

with

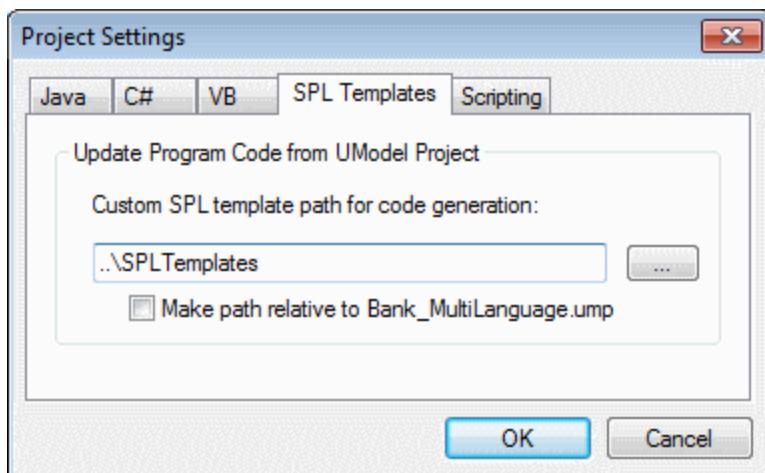
```
write "m_" & $Property.name
```

It is highly recommended that you immediately update your model from code after code generation, to ensure that code and model are synchronized.

**Note:** As previously mentioned, copy the SPL templates one directory higher (i.e. above the **default** directory to **UModelSPL\CS**) before modifying them. This ensures that they are not overwritten when you install a new version of UModel. Please make sure that the "user-defined override default" check box is activated in the **Code from Model** tab of the "Synchronization Settings" dialog box.

## SPL Templates

SPL templates can be specified per UModel project using the menu option **Project | Project Settings** (as shown in the screenshot below). Relative paths are also supported. Templates which are not found in the specified directory, are searched for in the local default directory.



## Global objects

\$Options	an object holding global options:
	generateComments:bool generate doc comments (true/false)
\$Indent	a string used to indent generated code and represent the current nesting level
\$IndentStep	a string, used to indent generated code and represent one nesting level
\$NamespacePrefix	XSD only – the target namespace prefix if present

## String manipulation routines

```
integer Compare(s)
```

The return value indicates the lexicographic relation of the string to s (case sensitive):

<0:	the string is less than s
0:	the string is identical to s
>0:	the string is greater than s

```
integer CompareNoCase(s)
```

The return value indicates the lexicographic relation of the string to s (case insensitive):

<0:	the string is less than s
0:	the string is identical to s
>0:	the string is greater than s

```
integer Find(s)
```

Searches the string for the first match of a substring s. Returns the zero-based index of the first character of s or -1 if s is not found.

```
string Left(n)
```

Returns the first n characters of the string.

```
integer Length()
```

Returns the length of the string.

```
string MakeUpper()
```

Returns a string converted to upper case.

```
string MakeUpper(n)
```

Returns a string, with the first n characters converted to upper case.

```
string MakeLower()
```

Returns a string converted to lower case.

```
string MakeLower(n)
```

Returns a string, with the first n characters converted to lower case.

```
string Mid(n)
```

Returns a string starting with the zero-based index position n

```
string Mid(n,m)
```

Returns a string starting with the zero-based index position n and the length m

```
string RemoveLeft(s)
```

Returns a string excluding the substring s if Left( s.Length() ) is equal to substring s.

```
string RemoveLeftNoCase(s)
```

Returns a string excluding the substring s if Left( s.Length() ) is equal to substring s (case insensitive).

```
string RemoveRight(s)
```

Returns a string excluding the substring s if Right( s.Length() ) is equal to substring s.

```
string RemoveRightNoCase(s)
```

Returns a string excluding the substring s if Right( s.Length() ) is equal to substring s (case insensitive).

```
string Repeat(s,n)
```

Returns a string containing substring s repeated n times.

```
string Right(n)
```

Returns the last n characters of the string.

## 13.3 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

## 13.4 Conditions

SPL allows you to use standard "if" statements. The syntax is as follows:

```
if condition
    statements
else
    statements
endif
```

or, without else:

```
if condition
    statements
endif
```

**Note:** There are no round brackets enclosing the condition.

As in any other programming language, conditions are constructed with logical and comparison [operators](#)<sup>532</sup>.

Example:

```
[if $namespace.ContainsPublicClasses and $namespace.Prefix <> ""]
    whatever you want ['inserts whatever you want, in the resulting file]
[endif]
```

## Switch

SPL also contains a multiple choice statement.

Syntax:

```
switch $variable
    case X:
        statements
    case Y:
    case Z:
        statements
    default:
        statements
endswitch
```

The case labels must be constants or variables.

The switch statement in SPL does not fall through the cases (as in C), so there is no need for a "break" statement.

## 13.5 Collections and foreach

### Collections and iterators

A collection contains multiple objects - like a ordinary array. Iterators solve the problem of storing and incrementing array indexes when accessing objects.

Syntax:

```
foreach iterator in collection
    statements
next
```

Example:

```
[foreach $class in $classes
    if not $class.Internal
        ]      class [= $class.Name];
[     endif
next]
```

Example 2:

```
[foreach $i in 1 To 3
    Write "// Step " & $i & "\n"
    ' Do some work
next]
```

**Foreach** steps through all the items in \$classes, and executes the code following the instruction, up to the **next** statement, for each of them.

In each iteration, **\$class** is assigned to the next class object. You simply work with the class object instead of using, **classes[i]->Name()**, as you would in C++.

All collection iterators have the following additional properties:

Index	The current index, starting with 0
IsFirst	true if the current object is the first of the collection (index is 0)
IsLast	true if the current object is the last of the collection

Example:

```
[foreach $enum in $facet.Enumeration
    if not $enum.IsFirst
        ], [
```

```
endif  
] "[$enum.Value]" [  
next]
```

## Collection manipulation routines:

collection **SortByName**( bAscending )

returns a collection whose elements are sorted by name (case sensitive) in ascending or descending order.

collection **SortByNameNoCase**( bAscending )

returns a collection whose elements are sorted by name (case insensitive) in ascending or descending order

### Example:

```
$SortedNestedClassifier = $Class.nestedClassifier.SortByNameNoCase( true )
```

collection **SortByKind**( bAscending )

returns a collection whose elements are sorted by kind names (e.g. “Class”, “Interface”,...) in ascending or descending order.

collection **SortByKindAndName**( bAscendingKind, bAscendingName )

returns a collection whose elements are sorted by kind (e.g. “Class”, “Interface”,...) in ascending or descending order and if the kinds are equal by name (case sensitive in ascending or descending order)

collection **SortByKindAndNameNoCase**( bAscending )

returns a collection whose elements are sorted by kind (e.g. “Class”, “Interface”,...) in ascending or descending order and if the kinds are equal by name (case insensitive in ascending or descending order)

## 13.6 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)

### 13.6.1 Subroutine declaration

#### Subroutines

Syntax example:

```
Sub SimpleSub()  
    ... lines of code  
EndSub
```

- **Sub** is the keyword that denotes the procedure.
- **SimpleSub** is the name assigned to the subroutine.
- Round **parenthesis** can contain a parameter list.
- The code block of a subroutine starts immediately after the closing parameter parenthesis.
- **EndSub** denotes the end of the code block.

**Note:** Recursive or cascaded subroutine **declaration** is not permitted, i.e. a subroutine may not contain another subroutine.

#### Parameters

Parameters can also be passed by procedures using the following syntax:

- All parameters must be variables
- Variables must be prefixed by the \$ character
- Local variables are defined in a subroutine
- Global variables are declared explicitly, outside of subroutines
- Multiple parameters are separated by the comma character "," within round parentheses
- Parameters can pass values

#### Parameters - passing values

Parameters can be passed in two ways, by value and by reference, using the keywords **ByVal** and **ByRef** respectively.

Syntax:

```
' define sub CompleteSub()
[Sub CompleteSub( $param, ByVal $paramByValue, ByRef $paramByRef )
] ...
```

- **ByVal** specifies that the parameter is passed by value. Note that most objects can only be passed by reference.
- **ByRef** specifies that the parameter is passed by reference. This is the default if neither **ByVal** nor **ByRef** is specified.

## Function return values

To return a value from a subroutine, use the **return** statement. Such a function can be called from within an expression.

Example:

```
' define a function
[Sub MakeQualifiedName( ByVal $namespacePrefix, ByVal $localName )
if $namespacePrefix = ""
    return $localName
else
    return $namespacePrefix & ":" & $localName
endif
EndSub
]
```

## 13.6.2 Subroutine invocation

Use **call** to invoke a subroutine, followed by the procedure name and parameters, if any.

```
Call SimpleSub()
```

or

```
Call CompleteSub( "FirstParameter", $ParamByValue, $ParamByRef )
```

## Function invocation

To invoke a function (any subroutine that contains a **return** statement), simply use its name inside an expression. Do not use the **call** statement to call functions. Example:

```
$QName = MakeQualifiedName($namespace, "entry")
```

## 14 License Information

This section contains information about:

- the distribution of this software product
- software activation and license metering
- the license agreement governing the use of this product

Please read this information carefully. It is binding upon you since you agreed to these terms when you installed this software product.

To view the terms of any Altova license, go to the [Altova Legal Information page](#) at the [Altova website](#).

## 14.1 Electronic Software Distribution

This product is available through electronic software distribution, a distribution method that provides the following unique benefits:

- You can evaluate the software free-of-charge for 30 days before making a purchasing decision. (*Note: Altova MobileTogether Designer is licensed free of charge.*)
- Once you decide to buy the software, you can place your order online at the [Altova website](#) and get a fully licensed product within minutes.
- When you place an online order, you always get the latest version of our software.
- The product package includes an onscreen help system that can be accessed from within the application interface. The latest version of the user manual is available at [www.altova.com](#) in (i) HTML format for online browsing, and (ii) PDF format for download (and to print if you prefer to have the documentation on paper).

### 30-day evaluation period

After downloading this product, you can evaluate it for a period of up to 30 days free of charge. About 20 days into the evaluation period, the software will start to remind you that it has not yet been licensed. The reminder message will be displayed once each time you start the application. If you would like to continue using the program after the 30-day evaluation period, you must purchase a product license, which is delivered in the form of a license file containing a key code. Unlock the product by uploading the license file in the Software Activation dialog of your product.

You can purchase product licenses at <https://shop.altova.com/>.

### Helping Others within Your Organization to Evaluate the Software

If you wish to distribute the evaluation version within your company network, or if you plan to use it on a PC that is not connected to the Internet, you may distribute only the installer file, provided that this file is not modified in any way. Any person who accesses the software installer that you have provided must request their own 30-day evaluation license key code and after expiration of their evaluation period, must also purchase a license in order to be able to continue using the product.

## 14.2 Software Activation and License Metering

As part of Altova's Software Activation, the software may use your internal network and Internet connection for the purpose of transmitting license-related data at the time of installation, registration, use, or update to an Altova-operated license server and validating the authenticity of the license-related data in order to protect Altova against unlicensed or illegal use of the software and to improve customer service. Activation is based on the exchange of license related data such as operating system, IP address, date/time, software version, and computer name, along with other information between your computer and an Altova license server.

Your Altova product has a built-in license metering module that further helps you avoid any unintentional violation of the End User License Agreement. Your product is licensed either as a single-user or multi-user installation, and the license-metering module makes sure that no more than the licensed number of users use the application concurrently.

This license-metering technology uses your local area network (LAN) to communicate between instances of the application running on different computers.

### Single license

When the application starts up, as part of the license metering process, the software sends a short broadcast datagram to find any other instance of the product running on another computer in the same network segment. If it doesn't get any response, it will open a port for listening to other instances of the application.

### Multi-user license

If more than one instance of the application is used within the same LAN, these instances will briefly communicate with each other on startup. These instances exchange key-codes in order to help you to better determine that the number of concurrent licenses purchased is not accidentally violated. This is the same kind of license metering technology that is common in the Unix world and with a number of database development tools. It allows Altova customers to purchase reasonably-priced concurrent-use multi-user licenses.

We have also designed the applications so that they send few and small network packets so as to not put a burden on your network. The TCP/IP ports (2799) used by your Altova product are officially registered with the IANA ([see the IANA Service Name Registry for details](#)) and our license-metering module is tested and proven technology.

If you are using a firewall, you may notice communications on port 2799 between the computers that are running Altova products. You are, of course, free to block such traffic between different groups in your organization, as long as you can ensure by other means, that your license agreement is not violated.

### Note about certificates

Your Altova application contacts the Altova licensing server ([link.altova.com](http://link.altova.com)) via HTTPS. For this communication, Altova uses a registered SSL certificate. If this certificate is replaced (for example, by your IT department or an external agency), then your Altova application will warn you about the connection being insecure. You could use the replacement certificate to start your Altova application, but you would be doing this at your own risk. If you see a *Non-secure connection* warning message, check the origin of the certificate and consult your IT team (who would be able to decide whether the interception and replacement of the Altova certificate should continue or not).

If your organization needs to use its own certificate (for example, to monitor communication to and from client machines), then we recommend that you install Altova's free license management software, [Altova LicenseServer](#), on your network. Under this setup, client machines can continue to use your organization's certificates, while Altova LicenseServer can be allowed to use the Altova certificate for communication with Altova.

## 14.3 Altova End-User License Agreement

- The Altova End-User License Agreement is available here: <https://www.altova.com/legal/eula>
- Altova's Privacy Policy is available here: <https://www.altova.com/privacy>

# Index

- - .NET 5,**
    - as UModel profile, 158
    - importing types from binaries, 96
    - support, 11
  - .NET Core, 11**
    - importing assemblies, 204
  - .NET Framework, 158**
    - importing assemblies, 204

## 3

- 3-way project,**
  - merge, 269

## A

- Abstract,**
  - class, 27
- Activation box,**
  - Execution Specification, 347
- Activity,**
  - Add diagram to transition, 309
  - Add operation, 309
  - Add to state, 309
  - create branch / merge, 294
  - diagram elements, 296
  - icons, 468
- Activity diagram, 290**
  - inserting elements, 291
- Actor,**
  - customize, 18
  - user-defined, 18
- Add, 452**
  - diagram to package, 18
  - new project, 147
  - package to project, 18
  - project to source control, 452
- Align,**
  - elements when dragging, 18
  - snap lines when dragging, 505
- All,**
  - expand / collapse, 380
- Artifact,**
  - add to node, 55
  - manifest, 55
- Association,**
  - aggregate/composite, 27
  - as relationship, 130
  - between classes, 27
  - changing the properties of, 133
  - creating, 130, 133
  - object links, 42
  - reflexive associations, 133
  - show typed property, 276
  - use case, 18
  - viewing, 133
- Association qualifier,**
  - creating, 133
- Associations,**
  - viewing, 87
- Attribute,**
  - autocomplete window, 505
  - coloring, 385
  - show / hide, 380
- Autocomplete,**
  - function, 27
- Autocompletion,**
  - window on class editing, 505
- Autocompletion of data types,**
  - disabling, 127
  - triggering, 127
- Autogenerate,**
  - reply message, 353
- Automatically add operation, 309**

## B

- Ball and socket,**
  - interface notation, 380
- Base,**
  - class, 36
- Base class,**

- Base class,**  
 expand, collapse compartments, 380  
 multiple instances on diagram, 380  
 overriding, 380
- Batch mode,**  
 creating projects, 101  
 loading projects, 101  
 saving projects, 101
- Behavioral,**  
 diagrams, 290
- Binary files,**  
 import into model, 199
- Binding,**  
 template, 276
- Branch,**  
 create in Activity, 294
- C**
- C#,**  
 code generation options, 169  
 code import options, 189  
 generating code, 164, 171  
 import attributes, 200  
 import binary files, 199, 204  
 importing source code, 187
- Call,**  
 message, 353
- Call message,**  
 go to operation, 353
- CallBehavior,**  
 insert, 291
- CallOperation,**  
 insert, 291
- Catalog,**  
 file - XMLSpy Catalog file, 505
- Change provider,**  
 source control, 460
- Check In, 450**
- Check Out, 448**
- Class,**  
 abstract and concrete, 27  
 add new, 27  
 add operations, 27  
 add properties, 27  
 associations, 27  
 base, 36  
 derived, 36  
 diagrams, 27  
 enable autocompletion window, 505  
 icons, 470  
 in component diagram, 49  
 name changes - synchronization, 215  
 synchronization, 212  
 syntax coloring, 385
- Class diagram, 380**
- Class name changing,**  
 effect on code file name, 215
- Classifier,**  
 constraining, 274  
 new, 213  
 renaming, 213
- Code, 215**  
 adding code to sequence diagram, 368  
 default, 505  
 generate from sequence diagram, 365  
 generating sequence diagrams from, 359  
 Java code and class file names, 215  
 refactoring, 215  
 SPL, 521  
 synchronization, 212
- Code engineering,**  
 errors, 91  
 from code to model, 69  
 from model to code, 60  
 generate ComponentRealizations, 213  
 information messages, 91  
 move project file to new location, 147  
 resolving associations, 136  
 tutorial samples, 14  
 warnings, 91
- Collaboration,**  
 Composite Structre diagram, 395
- Collapse,**  
 class compartments, 380
- Collection Association,**  
 creating, 136  
 prerequisites, 136  
 resolving to collection templates, 136
- Color,**  
 syntax coloring - enable/disable, 385
- Combined fragment, 349**
- Command,**  
 add to toolbar/menu, 496

**Command line,**

- creating projects, 101
- Generating program code, 96
- Importing binary types, 96
- Importing source code, 96
- loading projects, 101
- Reference, 96
- saving projects, 101
- Synchronizing code and model, 96

**Communication,**

- icons, 471

**Communication diagram, 335**

- generate from Sequence diagram, 336

**Compare source files, 458****Compartment,**

- expand single / multiple, 380

**Compatibility,**

- updating projects, 212

**Component,**

- diagram, 49
- icons, 473
- insert class, 49
- realization, 49

**Component diagram, 397****Component view,**

- as package, 107

**ComponentRealizations,**

- autogeneration, 213

**Composite state, 316**

- add region, 316

**Composite Structure,**

- icons, 472
- insert elements, 395

**Composite Structure diagram, 394****Composition,**

- association - create, 27

**Concrete,**

- class, 27

**Constraining,**

- classifiers, 274

**Containment,**

- drawing in a diagram, 139

**Copyright information, 538****CR/LF,**

- for ump file on save, 147

**Create,**

- getter / setter methods, 380

**Customize,**

- actor, 18
- toolbar/menu commands, 496

# D

**Default,**

- project code, 505
- SPL templates, 212

**Delete,**

- command from toolbar, 496
- icon from toolbar, 496
- toolbar, 497

**Dependencies,**

- viewing, 87

**Dependency,**

- include, 18
- usage, 49

**Deployment,**

- diagram, 55
- icons, 474

**Deployment diagram, 397****Derived,**

- class, 36

**Diagram, 398**

- Activity, 290
- Communication, 335
- Component, 397
- Composite structure, 394
- Deployment, 397
- Interaction Overview, 339
- Object, 398
- Package, 398
- Sequence, 344
- State machine, 307
- Timing, 371
- Use Case, 335

Add activity to transition, 309

add to Favorites, 84

adding code to sequence diagram, 368

Additional - XML schema, 417

Class, 380

finding unused elements, 111

generate code from sequence diagram, 365

generate Package dependency diagram, 398

icon reference, 79

icons, 467

**Diagram, 398**

- ignore elem. from included files, 505
- inserting elements into, 105
- multiple instances of class, 380
- quick scroll, 89
- save as png, 486
- save open diagrams with project, 505
- styles, 86
- viewing an outline of, 89
- XML Schema, 417

**Diagram Tree window, 83****Diagram type,**

- identifying, 93

**Diagrams, 289**

- behavioral, 290
- changing the appearance of, 123
- changing the size of, 123
- creating, 93, 119
- deleting from project, 123
- fit into window, 129
- generating, 120
- generating from Hierarchy window, 87
- opening, 122
- structural, 380
- viewing inside a project, 83
- zoom in/out, 129

**Directory,**

- change project location, 147
- ignoring on merge, 505

**Disable source control, 445****Distribution,**

- of Altova's software products, 538, 539

**Documentation,**

- adding to elements, 116
- generate from UML project, 278
- generating source code with, 116
- importing from source code, 116

**Documentation window, 90****Download source control project, 442****Drid,**

- snap lines while dragging, 18

**DurationConstraint,**

- Timing diagram, 377

**E****Edit menu,**

- commands, 488

**Element,**

- add to Favorites, 84
- styles, 86

**ElementImport,**

- viewing, 87

**Elements,**

- adding to a diagram, 105
- adding to the model, 79, 104
- aligning within a diagram, 125
- applying custom images to, 117
- autolayout, 125
- changing properties of, 85
- changing the appearance of, 117
- constraining, 112
- copying, 107
- deleting from diagram, 108
- deleting from project, 108
- documenting, 90, 116
- finding, 109
- finding in a diagram, 111
- hyperlinking, 113
- ignore from include files, 505
- insert State Machine, 308
- moving, 107
- renaming, 107
- replacing, 109
- resizing, 125

**Enable source control, 445****End User License Agreement, 538, 542****Enhance,**

- performance, 163

**Entry point,**

- add to submachine, 316

**Errors,**

- during code engineering, 91

**Evaluation period,**

- of Altova's software products, 538, 539

**Event/Stimulus,**

- Timing diagram, 376

**Exception,**

- Adding raised exception, 380

**Execution specification,**

lifeline, 347

**Exit point,**

add to submachine, 316

**Expand,**

all class compartments, 380

**Export,**

UModel projects to XMI, 435

**External applications,**

opening from UModel, 498

# F

**Favorites window,**

adding to, 84

removing from, 84

**Fetch file,**

source control, 446

**File,**

merging project files, 269

open from URL, 486

ump, 147

**File menu,**

commands, 486

**Find,**

diagrams, 109

elements, 109

text, 109

**Folders,**

get in source control, 447

**Forward engineering, 60**

# G

**Gate,**

sequence diagram, 352

**General Value lifeline,**

Timing diagram, 372

**Generalization,**

as relationship, 105, 130

creating, 130

**Generalizations,**

viewing, 87

**Generalize,**

specialize, 36

**Generate,**

ComponentRealizations automatically, 213

reply message automatically, 353

Sequence dia from Communication, 336

UML project documentation, 278

**Generated documentation,**

options, 282

**Get,**

getter / setter methods, 380

**Get file,**

source control, 446

**Get folders,**

source control, 447

**Get latest version, 446****Goto,**

lifeline, 347

**Grid,**

snap lines, 505

# H

**Help menu,**

commands, 516

**Hide,**

show - slot, 380

**Hierarchy diagram,**

levels shown in documentation, 278

**Hierarchy window, 87****History,**

show, 456

**Hotkeys,**

assigning, 502

deleting, 502

**Hyperlinks,**

in documentation text, 116

# I

**Icon,**

Activity, 468

add to toolbar/menu, 496

class, 470

Communication, 471

**Icon,**  
 component, 473  
 Composite Structure, 472  
 deployment, 474  
 Interaction Overview, 475  
 object, 476  
 Package, 477  
 Sequence, 480  
 show large, 504  
 State machine, 481  
 Timing, 482  
 use case, 483  
 XML Schema, 484

**Icons,**  
 visibility, 380

**Ignore,**  
 directories, 505  
 elements in list, 505

**Images,**  
 using as element background, 117

**Import,**  
 XMI to UModel, 435

**Include,**  
 .NET Framework, 158  
 dependency, 18  
 UModel project, 158

**Insert, 291**  
 action (CallBehavior), 291  
 action (CallOperation), 291  
 Composite Structure elements, 395  
 Interaction Overview elements, 340  
 Package diagram elements, 400  
 simple state, 309  
 Timing diagram elements, 372

**Instance,**  
 diagram, 42  
 multiple class, and display of, 380  
 object, 42

**Intelligent,**  
 autocomplete, 27

**Interaction operand, 349**  
 multi-line, 349

**Interaction operator,**  
 defining, 349

**Interaction Overview,**  
 icons, 475  
 inserting elements, 340

**Interaction Overview diagram, 339**

**Interaction use, 352**

## J

**Java,**  
 code and class file names, 215  
 code generation options, 169  
 code import options, 189  
 generating code, 164, 176  
 import annotations, 200  
 import binary files, 206  
 importing source code, 187

## L

**Layout menu,**  
 commands, 493

**Legal information, 538**

**License, 542**  
 information about, 538

**License metering,**  
 in Altova products, 540

**Lifeline, 347**  
 attributes, 347  
 General Value, 372  
 typed property as, 347

**Lifeline,**  
 goto, 347

**Line,**  
 orthogonal, 49

**Line break,**  
 in actor text, 18

**Lines,**  
 changing the style of, 131  
 custom, 131  
 direct, 131  
 formatting, 42  
 moving, 131  
 orthogonal, 131  
 snap lines, 505

**Links,**  
 in generated documentation, 282

**Local project, 442**

**Location,**

**Location,**  
move project, 147

## M

**Mail,**  
send project, 486

**Manifest,**  
artifact, 55

**Menu,**  
add/delete command, 496

**Merge,**  
3-way manual project merge, 271  
3-way project merge, 269  
create in Activity, 294  
ignore directory, 505  
projects, 269

**Message, 353**  
arrows, 353  
call, 353  
create object, 353  
go to operation, 353  
inserting, 353  
moving, 353  
numbering, 353  
Timing diagram, 378

**Messages window,**  
reference, 91

**Method,**  
Add raised exception, 380

**Methods,**  
getter / setter, 380

**Model,**  
adding elements to, 79, 104  
changing class name - effect in Java, 215

**Model Tree window,**  
expanding or collapsing items, 79  
exploring the project from, 79  
icon reference, 79  
showing or hiding items, 79  
sorting items, 79

**Modeling,**  
enhance performance, 163

**Move,**  
project, 147

**Moving message arrows, 353**

**Multiline, 18**  
**Multi-line,**  
actor text, 18  
interactionOperand, 349  
use case, 18

## N

**Name,**  
region names - hide / show, 316

**New,**  
classifier, 213

**New line,**  
in Lifeline, 336  
interactionOperand, 349

**Node,**  
add, 55  
add artifact, 55  
styles, 86

**Numbering,**  
messages, 353

## O

**Object,**  
create message, 353  
diagram, 42  
icons, 476  
links - associations, 42

**Object diagram, 398**

**Open Project,**  
source control, 442

**OpenJDK,**  
importing binaries, 200

**Operand,**  
interaction, 349

**Operation,**  
autocompletion window, 505  
Automatically add on Activity, 309  
coloring, 385  
goto from call message, 353  
overriding, 380  
reusing, 36  
show / hide, 380

- Operation,**  
    template, 276
- Operations,**  
    adding, 27
- Operator,**  
    interaction, 349
- Options,**  
    source control, 505  
    tools, 505  
    when generating documentation, 282
- Orthogonal,**  
    line, 49  
    state, 316
- Override,**  
    class operations, 380  
    default SPL templates, 212
- Overview window,**  
    scrolling, 89
- P**
- Package,**  
    default packages, 79  
    icon reference, 79  
    icons, 477
- Package diagram, 398**  
    generating dependency diagram, 398  
    insert elements, 400
- PackageImport, 400**  
    viewing, 87
- PackageMerge, 400**  
    viewing, 87
- Parameter,**  
    template, 276
- Path,**  
    change project location, 147  
    SPL template path, 523
- Performance,**  
    enhancement, 163
- Pretty print,**  
    in exported XMI files, 435  
    project on save, 147
- Print preview,**  
    options, 486
- Profiles,**  
    applying to a package, 154, 405
- built-in, 405  
    creating, 405  
    definition, 404
- Project, 147**  
    3-way manual merge, 271  
    3-way merge, 269  
    add or remove items, 79  
    add to source control, 452  
    create, 147  
    default code, 505  
    exploring, 79  
    file - updating, 212  
    generating documentation, 278  
    include UModel project, 158  
    insert package, 147  
    Merge, 269  
    modularize, 155  
    move, 147  
    open last on start, 505  
    remove from source control, 454  
    save - pretty print, 147  
    save open diagrams, 505  
    send by mail, 486  
    split into subprojects, 155  
    styles, 86  
    workflow, 147
- Project menu,**  
    commands, 490
- Project open,**  
    source control, 442
- Project syntax,**  
    checking, 91
- Properties,**  
    adding, 27  
    source control, 459
- Properties window,**  
    adding custom properties, 85
- Property,**  
    coloring, 385  
    reusing, 36  
    typed - show, 276  
    typed as lifeline, 347
- Provider,**  
    select, 442

# R

**Raised exception,**  
    Adding, 380

**Realization,**  
    component, 49  
    generate ComponentRealizations, 213

**Refactoring code,**  
    class names - synchronization, 215

**Reference, 485**

**Refresh status,**  
    source control, 460

**Region,**  
    add to composite state, 316

**Region name,**  
    show / hide, 316

**Reject source edits, 450**

**Relationships,**  
    aggregation, 130  
    association, 105, 130  
    changing the style of, 131  
    composition, 130  
    dependency, 130  
    generalization, 105, 130  
    realization, 130  
    viewing, 133

**Remove,**  
    from source control, 454

**Rename,**  
    classifier, 213

**Reply,**  
    message - autogenerate, 353

**Reset,**  
    toolbar & menu commands, 497

**Restore,**  
    toolbars and windows, 495

**Reverse engineering, 69**

**Root,**  
    as package, 107  
    catalog - XMLSpy, 505  
    package/class synchronization, 212

**Run native interface, 460**

# S

**Save,**  
    diagram as image, 486

**SC,**  
    syntax coloring, 385

**Search,**  
    diagrams, 109  
    elements, 109  
    text, 109

**Send by mail,**  
    project, 486

**Sequence,**  
    icons, 480

**Sequence diagram, 344**  
    adding code to, 368  
    combined fragment, 349  
    gate, 352  
    generate code from, 365  
    generate from Communication diag, 336  
    inserting elements, 345  
    interaction use, 352  
    lifeline, 347  
    messages, 353  
    state invariant, 353

**Sequence diagrams,**  
    generating from getters/setters, 363  
    generating from source code, 359  
    generating multiple, 363

**Set,**  
    getter / setter methods, 380

**Setting,**  
    synchronization, 212

**Settings,**  
    source control, 505

**Share,**  
    from source control, 455

**Shortcut,**  
    show in tooltip, 504

**Shortcuts,**  
    assigning, 502  
    deleting, 502

**Show,**  
    hide - slot, 380  
    hide- region name, 316

- Show,**  
property as association, 276
- Show differences, 458**
- Show history, 456**
- Show/hide,**  
attributes, operations, 380
- Signature,**  
template, 274, 275
- Slot,**  
show / hide, 380
- Snap,**  
line - when dragging, 505
- Snap lines, 18**
- Socket,**  
Ball and socket, 380
- Software product license, 542**
- Source control,**  
add to source control, 452  
change provider, 460  
Check In, 450  
Check Out, 448  
commands, 442  
enable / disable, 445  
get file, 446  
get latest version, 446  
installing a source-control plug-in, 437  
open project, 442  
options / settings, 505  
properties, 459  
refresh status, 460  
remove from, 454  
run native interface, 460  
show differences, 458  
show history, 456  
Undo Check out, 450
- Specialize,**  
generalize, 36
- Speed,**  
enhancement, 163
- Spelling,**  
checking, 90
- SPL, 521**  
code blocks, 522  
conditions, 533  
foreach, 534  
subroutines, 536  
templates user-defined, 212
- SPL templates,**
- template path, 523
- Start,**  
with previous project, 505
- State, 316**  
add activity, 309  
composite, 316  
define transition between, 309  
insert simple, 309  
orthogonal, 316  
submachine state, 316
- State changes,**  
defining on a timeline, 372
- State invariant, 353**
- State machine,**  
composite states, regions, 316  
diagram elements, 328  
icons, 481  
insert elements, 308  
states, activities, transitions, 309
- State Machine Diagram, 307**
- Stereotypes,**  
adding custom icons to, 414  
adding custom styles to, 414  
adding to the Properties window, 85  
applying to elements, 142, 409  
creating, 406, 409  
definition, 140  
example, 409  
examples, 140, 404
- Structural,**  
diagrams, 380
- Styles,**  
applying to diagrams, 123  
applying to elements, 117  
applying to lines, 131  
cascading, 117, 123, 131  
precedence, 117, 123, 131
- Styles window, 86**
- StyleVision,**  
customize generated documentation with, 287  
customizing generated documentation with, 278
- Submachine state,**  
add entry/exit point, 316
- Subproject,**  
create from main project, 155  
reintegrate into main project, 155
- Symbols,**  
visibility icons, 380

**Synchronization, 215**

- class and code file name, 215
- class name changes, 215
- settings, 212

**Synchronize,**

- root/package/class, 212
- to new location, 147

**Syntax coloring, 385**

# T

**Tagged values,**

- as enumerations, 406, 409
- creating, 142, 406
- definition, 141
- example, 409
- examples, 141
- showing or hiding, 144

**Template,**

- binding, 276
- operation/parameter, 276
- signature, 274, 275

**Templates,**

- SPL templates, 523
- user-defined SPL, 212

**Tick mark,**

- Timing diagram, 375

**TimeConstraint,**

- Timing diagram, 378

**Timeline,**

- defining state changes, 372

**Timing,**

- icons, 482

**Timing diagram, 371, 372**

- DurationConstraint, 377
- Event/Stimulus, 376
- General Value lifeline, 372
- inserting elements, 372
- Lifeline, 372
- Message, 378
- switch between types, 372
- Tick mark, 375
- TimeConstraint, 378
- Timeline, 372

**Toolbar,**

- activate/deactivate, 497

- add command to, 496

- create new, 497

- reset toolbar & menu commands, 497

- show large icons, 504

**Toolbars,**

- restore to default, 495

**Tools,**

- options, 505

**Tools menu,**

- adding custom commands to, 498

**Tooltip,**

- show, 504

- show shortcuts in, 504

**Transition,**

- Add Activity diagram to, 309
- define between states, 309
- define trigger, 309

**Trigger,**

- define transition trigger, 309

**Tutorial,**

- sample files, 14

**Type,**

- property - show, 276

**Typed,**

- property - as lifeline, 347

# U

**UML,**

- Diagrams, 289
- templates, 274
- variables, 523
- visibility icons, 380

**UModel,**

- Introduction, 11
- Main features, 11

**UModel diagram icons, 467****UModel projects,**

- opening, saving, creating, 15

**UMP, 147**

- change project location, 147
- file extension, 147

**Undo Check out, 450****Update,**

- project file, 212

**URL,**

**URL,**  
open file from, 486

**Usage,**  
dependency, 49

**Use case,**  
adding, 18  
association, 18  
compartments, 18  
icons, 483  
multi-line, 18

**Use Case diagram, 335**

**User defined,**  
actor, 18

**User-defined,**  
SPL templates, 212

source control, 442

## X

**XMI,**  
import and export, 435

**XML Schema,**  
creating diagrams, 423  
declare namespace, 423  
diagrams, 417  
generating from model, 425  
icons, 484  
importing into a model, 418  
modeling, 423, 425

## V

**Variables,**  
UML, 523

**VB.NET,**  
code generation options, 169  
code import options, 189  
generating code, 164  
import binary files, 199  
importing source code, 187

**Version control,**  
commands, 442

**View,**  
to multiple instances of element, 380

**View menu,**  
commands, 494

**Visibility,**  
icons - selecting, 380

## W

**Warnings,**  
during code engineering, 91

**Windows,**  
restore to default, 495

**Workflow,**  
project, 147

**Working directory,**