



# CODE ENGINEERING

version 17.0.1

user guide

**No Magic, Inc.**  
**2011**

All material contained herein is considered proprietary information owned by No Magic, Inc. and is not to be shared, copied, or reproduced by any means. All information copyright 1998-2011 by No Magic, Inc. All Rights Reserved.

# CONTENTS

---

## INTRODUCTION 7

Overview 7

Code Engineering Sets 8

MagicDraw Code Engineering File (\*.mdr) 10

Generating Code 11

Code Generation for Set 11

Code Generation for Model Element 13

Reverse 13

Rules of the association or attribute creation on reverse 15

Global options for Code Engineering 18

Code engineering options for all sets in your project 18

Java Documentation Properties dialog box 23

Round Trip 23

Type Mapping Table 24

Files of Properties 24

## JAVA CODE ENGINEERING 26

Introduction 26

Abbreviations 26

References 26

Java support in MagicDraw 26

Java Mapping to UML 26

Java Profile 26

Java referenced types 28

Mapping to UML rules 29

Java CE Properties 56

Java Reverse Properties 56

Java Language Options 57

Method Implementation Reverse 60

Sequence Diagram from Java Source Wizard 60

## C++ CODE ENGINEERING 65

Abbreviations 65

References 65

C++ ANSI Profile 65

Data Types 65

Stereotypes 68

Mapping 78

Class 78

Base Class Definition 78

Class Member Variable 79

Class Member Function 79

Class Constructor/Destructor 79

Variable 80

Variable modifiers 82

Variable extern 83

Variable default value 83

Const Volatile qualified type 84

Function 85

# CONTENTS

---

Function variable-length parameter list	86
void parameter	86
Register parameter	87
Function modifiers	87
Function pointer	88
Function operator	89
Exception	89
Visibility	90
Static members	90
Friend declaration	91
Struct	92
Union	92
Enumeration	92
Typedef	93
Namespace	94
Global functions and variables	94
Class definition	95
Class Template Definition	95
Function Template Definition	96
Default template parameter	97
Template instantiation	98
Partial template instantiation	101
Template specialization	102
Forward class declaration	103
Include declaration	104
<b>Conversion from old project version</b>	<b>107</b>
Translation Activity Diagram	107
Language properties	111
Type Modifiers	131
Stereotypes	135
Tag Value	145
Constructor and Destructor name	146
Data type	148
<b>DSL customization</b>	<b>154</b>
Operation and Constructor	154
Attribute	155
Generalization	156
Enumeration literal	156
Namespace	156
Template parameter	157
<b>Profile constraints</b>	<b>157</b>
Operation	157
Constructor	157
Destructor	158
Global	158
Typedef	158
Friend	158
<b>New in MagicDraw 12.1</b>	<b>159</b>
CG Properties Editor	159
Roundtrip on #include statement and forward class declaration	160



# CONTENTS

---

Project Option and Code Generation Options	165
<b>New in MagicDraw 14.0</b>	<b>168</b>
Support C++ dialects	168
CG Properties Editor	169
<b>New in MagicDraw 16.8</b>	<b>170</b>
Doxygen-after-member documentation	170
@see on documentation	171
Symbian macro	171
<b>Tutorial</b>	<b>171</b>
Type Modifier	171
Global Member	172
Typedef	172
Function Pointer	174
Friend	176
How to specify component to generate code to	178
@see support for import-code-only mode	179
Navigable short cut from model link in documentation	180
<b>Project constraint</b>	<b>181</b>
Working with QT	185
<b>Microsoft C++ Profiles</b>	<b>186</b>
Microsoft Visual C++ Profile	186
C++/CLI Profile	194
C++ Managed Profile	203
Modeling with Microsoft Specific Profiles	205
<b>C# CODE ENGINEERING</b>	<b>241</b>
<b>C# 2.0 Description</b>	<b>241</b>
Generics	241
Anonymous Methods	263
Partial Types	264
Nullable Types	267
Accessor Declarations	268
Static Class	269
Extern Alias Directive	270
Pragma Directives	272
Fix Size Buffer	274
<b>C# 3.0 Description</b>	<b>277</b>
Extension Methods	277
Lambda Expression Conversion	278
<b>C# Profile</b>	<b>280</b>
Stereotype	280
Data Type	285
<b>Conversion from old project version</b>	<b>286</b>
Translation Activity Diagram	286
<b>Mapping</b>	<b>291</b>
Language Properties Mapping	291
C# Properties Customization	295
Using Directive Mapping	296
<b>Constraints</b>	<b>300</b>
Mapping Constraints	300

# CONTENTS

---

UML Constraints **300**  
Translation Constraints **304**

CORBA IDL MAPPING TO UML **305**  
NEW! CORBA Interface Implementation **306**

EJB 2.0 - UML **309**  
Java Profile **309**  
EJB Design Profile **310**  
EJB Deployment Profile **316**  
Using MagicDraw EJB 2.0 **318**  
Reverse engineering **318**  
Code generation **318**

WSDL **320**  
WSDL Mapping to UML elements **321**  
Defined stereotypes **321**  
Definitions **321**  
Import, namespace **322**  
Messages **322**  
Types **324**  
Port types **325**  
Bindings **326**  
Services **328**  
Ports **330**

# INTRODUCTION

## Overview

**View Online Demos** Code Generation  
Code Reverse

**NOTE:** Code Engineering is available in Professional, Architect and Enterprise editions only.

MagicDraw code engineering provides a simple and intuitive graphical interface for merging code and UML models, as well as preparing both code skeletons out of UML models and models from code.

MagicDraw code engineering implements several cases where code engineering may be very useful:

- You already have code that needs to be reversed to a model.
- You wish to have the implementation of the created model.
- You need to merge your models and code.

The tool may generate code from models and create models out of code (reverse). Changes in the existing code can be reflected in the model, and model changes may also be seen in your code. Independent changes to a model and code can be merged without destroying data in the code or model.

MagicDraw UML code engineering supports Java, C++, CORBA IDL, DDL, XML Schema, WSDL, and C# languages also EJB 2.0 UML notation is supported. You may model EJB classes and generate descriptors for them. You may also reverse descriptors and will get a model describing your Enterprise Java Beans. Your models can be converted to any of those languages, or UML models can be created from the source code written in those languages. Also reverse from Java Bytecode and CIL is supported.

The Code Engineering Sets tool is MagicDraw tool managing center for all code engineering matters.

Code engineering is available only in Professional or Enterprise editions. In the following table you'll find what languages are supported in different editions:

Language	Professional Edition	Enterprise Edition
Java	Java	+
Java Bytecode	Java	+
C++	C++	+
CORBA IDL	-	+
DDL/Database engineering	With Cameo Data Modeler plugin (separately purchaseable)	With Cameo Data Modeler plugin (free of charge)
CIL	C#	+
CIL Disassembler	C#	+
XML Schema	With Cameo Data Modeler plugin (separately purchaseable)	With Cameo Data Modeler plugin (free of charge)
WSDL	-	+
C#	C#	+
EJB 2.0	-	+

## Code Engineering Sets

You may manage code engineering through the **Code Engineering Sets** in the Browser tree. The **Code Engineering Sets** tree contains the list of all sets created in the project and instruments for managing those sets.

To add a new set

1. From the **Code Engineering Sets** shortcut menu, choose **New**.
2. Choose the language you want (possible choices include: **Java**, **Java Bytecode**, **C++**, **C#**, **CIL**, **CIL Disassembler**, **CORBA IDL**, **DDL (Cloudscape, DB2, Microsoft Access, Microsoft SQL Server, MySQL, Oracle, Pervasive, Pointbase, PostgreSQL, Sybase)**, **EJB 2.0**, **XML Schema**, and **WSDL**). The new set is created.

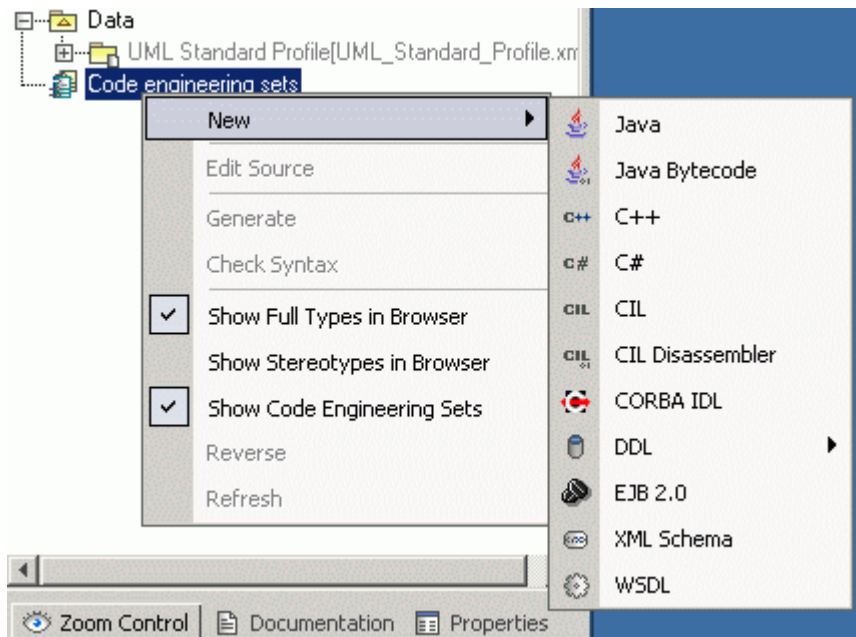


Figure 1 -- Code engineering language options

Edit sets in the **Round Trip Set** dialog box. To open this dialog box

- Choose **Edit** from the set shortcut menu.

If you are performing round trip for the first time, the tip message box appears.

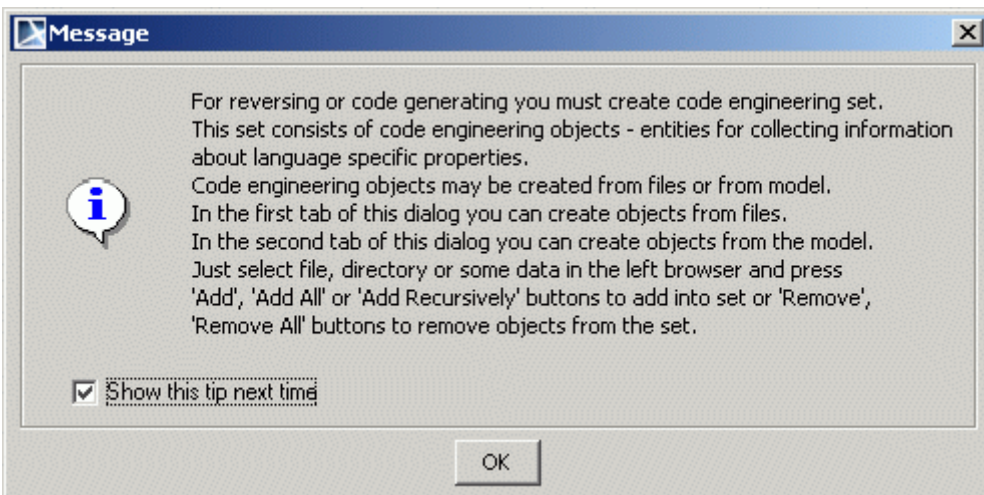


Figure 2 -- Code Engineering Sets tip message box

Disable the tip message box by deselecting the **Show this tip next time** check box.

The **Round Trip Set** dialog box allows you to manage entities to be added/removed to your set.

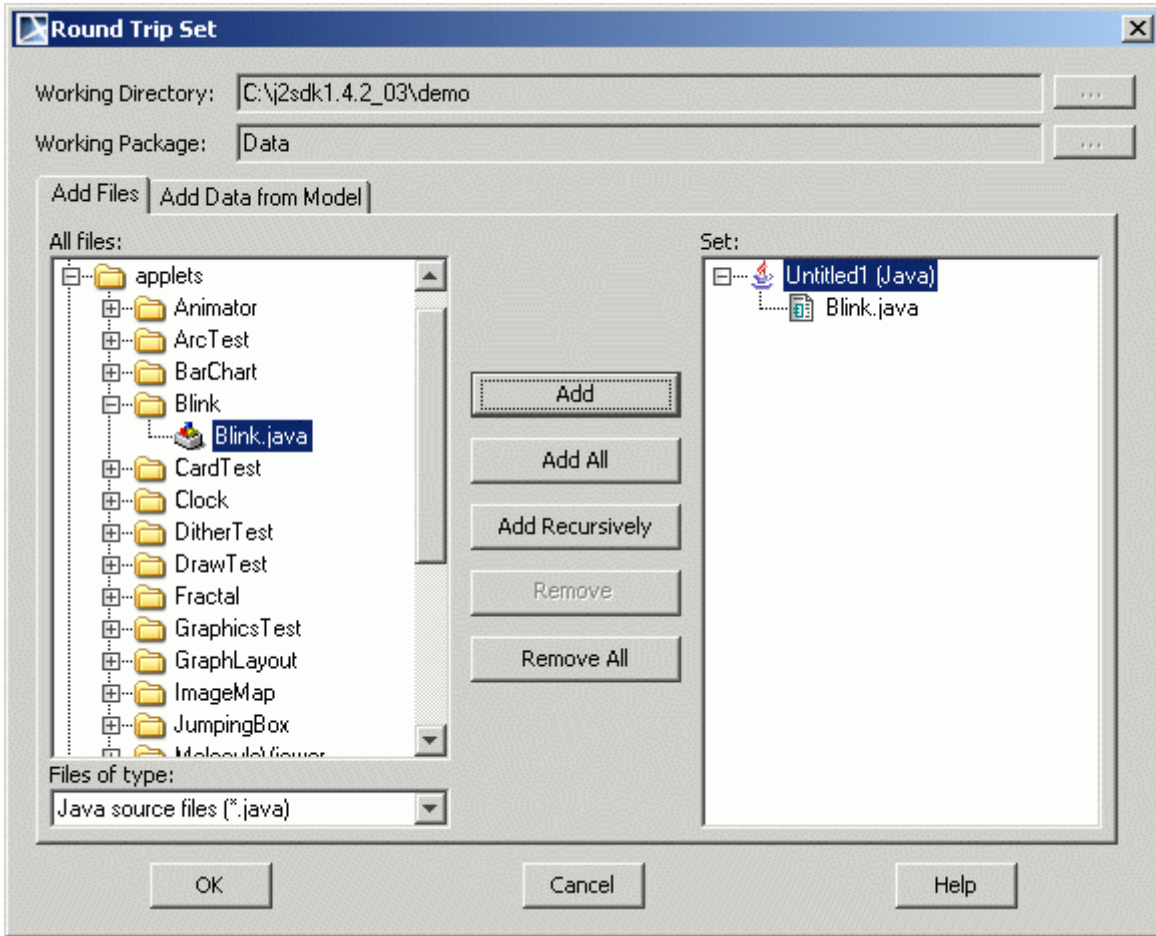


Figure 3 -- Round Trip Set dialog box. Add files tab

Specify **Working Directory** for displaying source files. This option indicates files and required sub-directories, where a code generation output goes. Type a path manually or by browsing in the directory tree, by clicking the '...' button.

The **Working Package** option allows to define any package for reverse output or code generation. Model will be reversed or code generated from this specified package.

**NOTE** The working package may be selected or changed only prior to the addition of files from working directory to code engineering set.

The **Round Trip Set** dialog box has two tabs: **Add Files** and **Add Data from Model**.

The **Add Files** tab helps you manage the files of source code involved in your code engineering set.

Element name	Function
<b>All files</b>	Helps you find directories with the source files for the set.
<b>Files of type</b>	Contains possible file name extensions for the chosen language.

The **Add Data from Model** tab helps you manage elements located in the UML model.

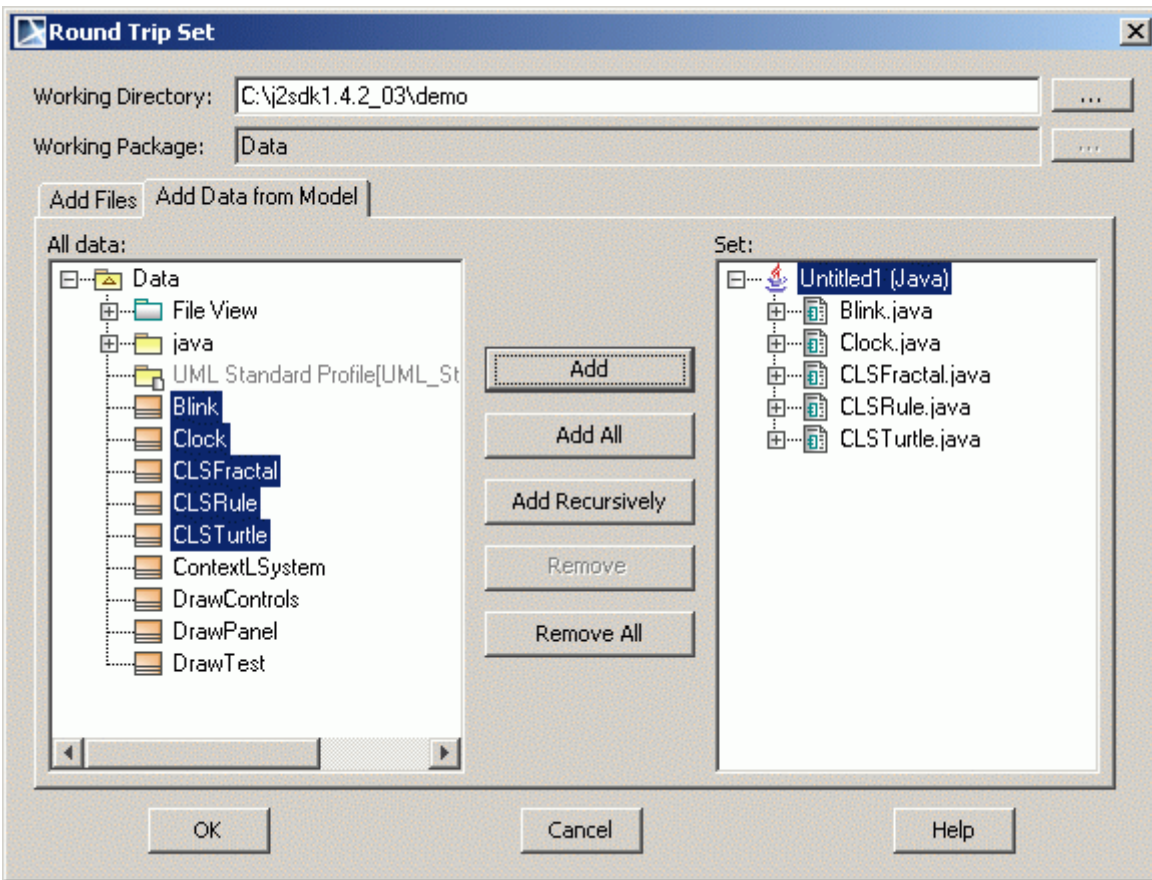


Figure 4 -- Round Trip Set dialog box. Add data from model tab

The **All Data** list contains the hierarchy of UML model packages with model elements (or other packages) inside of them. Your code engineering set can be combined out of model and code elements.

The following buttons are available in the **Round Trip Set** dialog box:

<b>Add</b>	The selected file in the <b>All Files</b> or <b>All Data</b> list is added to the set.
<b>Add All</b>	All files in the opened or selected directory are added to the set.
<b>Add Recursively</b>	All files in the selected directory and its subdirectories are added to the set.
<b>Remove</b>	Removes the selected entity from the set.
<b>Remove All</b>	Removes all entities from the set.

## MagicDraw Code Engineering File (\*.mdr)

Information about code engineering sets with inner objects of these sets and code engineering options are stored in the \*.mdr file. This file is automatically created together with a new MagicDraw project creation.

If you are working with code engineering in MagicDraw, then together with MagicDraw project file \*.mdzip you have to keep \*.mdr file. When you want to move, copy or do other actions with project file you have to do these actions with both files - \*.mdzip and \*.mdr file.

Name of the \*.mdr file is the same as name of the created MagicDraw project. The \*.mdr file is located at the same place as the project itself. See the sample in Figure 5 on page 11.

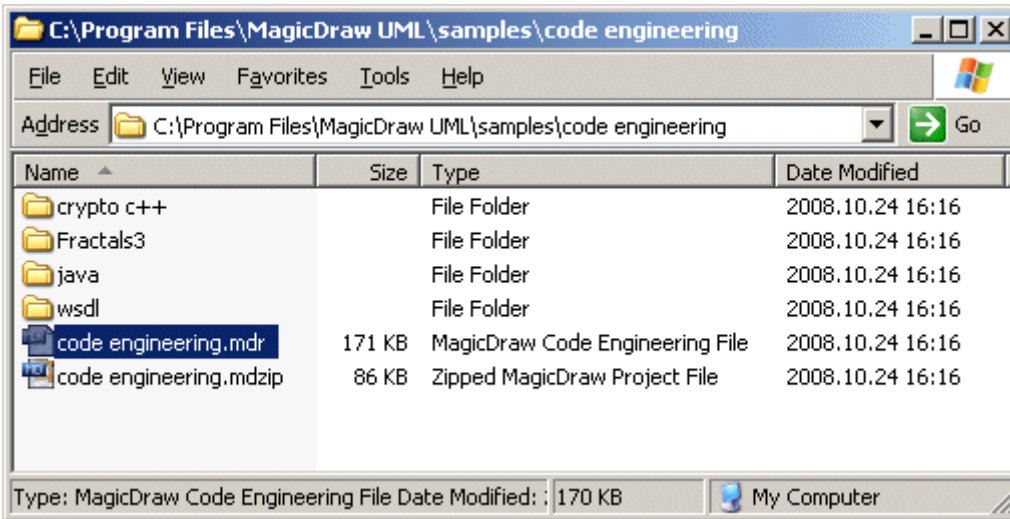


Figure 5 -- Sample of the \*.mdr file name and location

If \*.mdr files were lost, \*.mdzip the rest of project would be loaded without problems, only information with sets would be lost and you will have to create sets with objects again.

## Generating Code

[View Online Demo](#) Code Generation

You may generate code for the selected and prepared set and directly for model elements.

### Code Generation for Set

Start code generation once the set or sets are prepared. For more details about creating and editing sets, see “Code Engineering Sets” on page 8.

- Choose **Generate** from the **Code Engineering Sets** item shortcut menu. It allows code generating for all created sets.
- Choose **Generate** from the selected set shortcut menu. It allows code generating only for the selected set.



The **Code Generation Options** dialog box appears.

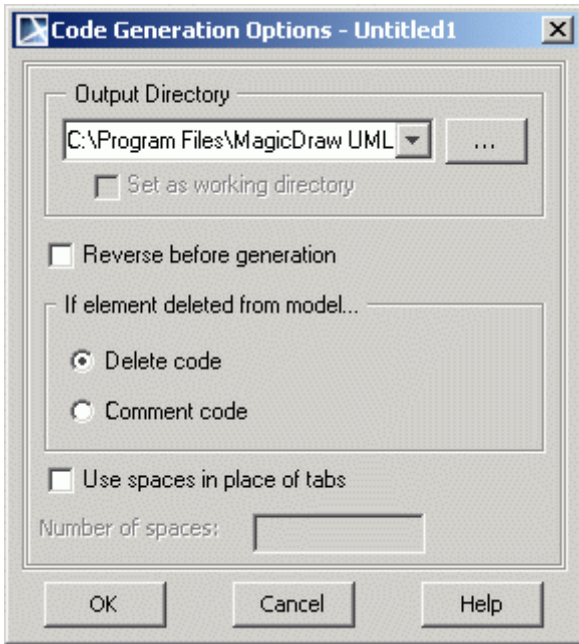


Figure 6 -- Code Generation Options dialog box.

The **Code Generation Options** dialog box allows you to specify the way your code will be generated.

Once generating options are specified for the set, code can be generated.

Box name	Function
<b>Output Directory</b>	Type the directory where the generated files will be saved.
'...'	The <b>Set Output Directory</b> dialog box appears. Select the directory for the generated files.
<b>Set as Working Directory</b>	The output directory is set as a working directory and files are saved to the working directory.
<b>Reverse before generation</b>	Changes your model according to changes in the existing code. <b>WARNING:</b> Exercise caution when selecting the <b>Reverse before generation</b> check box. If the model differs from the code, all differences in the model will be discarded. In such cases, you will lose some your work.
<b>If element deleted from model</b>	To influence the structure of generated code, click one of the following option buttons: <ul style="list-style-type: none"> <li>• <b>Delete code.</b> The representation of deleted entities will be deleted from the code file.</li> <li>• <b>Comment code.</b> Deleted entities will be commented in the code file.</li> </ul>
<b>Use spaces in place of tabs</b>	When selected, spaces (instead of tabs) will be written to the code file.
<b>Number of spaces</b>	Specify the number of spaces to be written.



Box name	Function
OK	The Messages Window appears, displaying how code files are being generated. The Messages Window informs you of problems and errors (mainly file access and syntax errors) found in the code generation process and generation summary. You are also prompted to confirm that you wish to overwrite the file if the output directory already contains one with the same name.
Cancel	Closes the dialog box without saving changes.
Help	Displays MagicDraw Help

## Code Generation for Model Element

All the classes contained in the component will be written to one file. However, code for the class can be generated in a different way. Select the class you wish to generate in the browser Data package and click Generate in the class shortcut menu. For packages and components, you may also select Generate, but you will not be able to specify the generation options. All the options related to that task will be set according to the default values.

If you have chosen framework generation for a single class or for packages, the **Code Generation Options** dialog box does not appear. The code is generated according to the default values.

If no errors occurred, you may view the results with your favorite file viewer or programming environment. Look for the files in the directory that you specified as your Working directory in the Round trip set dialog box or in the **Project Options** dialog box. Additional sub-directories could be created.

## Reverse

[View Online Demo](#)
[Code Reverse](#)

A reverse is an opposite operation to the code generation. The existing code can be converted to UML models with the help of MagicDraw reverse mechanism.

Prepare the sets in the exact same way that you did for code generation (see “Code Engineering Sets” on page 8)

- Choose **Reverse** from the **Code engineering sets** item shortcut menu. It allows code reversing for all already created sets.
- Choose **Reverse** from the selected set shortcut menu.

The UML model for the component can be reversed in the same way. Just select the component you are interested in from the browser and click **Reverse** on it shortcut menu.

Models can be reversed without creating a set.

To reverse a model without creating a set

- NOTE:**
1. From the **Tools** menu, choose **Quick Reverse**. The **Round Trip Set** dialog box appears. Quick Reverse is available only in Professional and Enterprise editions.
  2. Select the files from the **Round Trip Set** dialog box, **Add Files** tab.

3. Click **OK**. The **Reverse Options** dialog box appears.

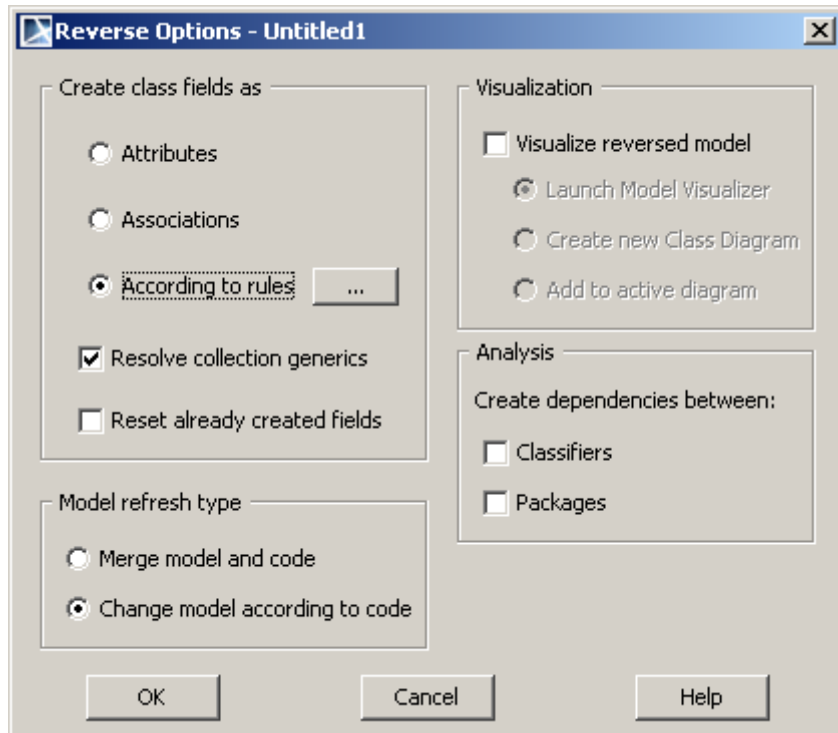


Figure 7 -- Reverse options dialog box

Element name	Function
<b>CREATE CLASS FIELDS AS</b>	
<b>Attributes</b>	Class fields are represented in model as attributes.
<b>Associations</b>	Class fields are represented in model as association ends.
<b>According to rules</b>	Association or Attribute creation on reverse is ability to enter rules that help to decide if an association or attribute must be created on reverse. For more information, see "Rules of the association or attribute creation on reverse" on page 15.
<b>Resolve collection generics</b>	Reverse engineering is able to create associations when one class has collection of other classes and uses Java generics (e.g. List<String>). If selected, types of collection will be resolved (property type will be not collection, but real type). Predefined container types in Java language properties will be appended by all the same containers in form: :java.util.List<\$\$type\$\$> where \$\$type\$\$ replaced to value of "type" property when code is generated.
<b>Reset already created fields</b>	Select this option if you want to keep already created UML representation (attribute or association) for class fields.
<b>MODEL REFRESH TYPE</b>	
<b>Merge model and code</b>	The model elements are updated by code. Elements that do not exist in the code will not be removed from the model.
<b>Change model according to code</b>	Model will be created strictly by code. Entities in the model that do not match entities in the code will be discarded.
<b>VIZUALIZATION</b>	

Element name	Function
<b>Visualize reversed model</b>	Classes that are created while reversing can be added to a diagrams.
<b>Launch Model Visualizer</b>	After reversing, the <b>Model Visualizer</b> dialog box appears. It will assist you in creating a class diagram or sequence diagram (Java only) for newly created entities.
<b>Create new class diagram</b>	After reversing, the <b>Create Diagram</b> dialog box appears. Create a new diagram where the created entities will be added.
<b>Add to active diagram</b>	After reversing, all created entities will be added to the current opened diagram.
<b>ANALYSIS - create dependencies between</b>	
<b>Classifiers</b>	Dependencies between classes will be analyzed and created.
<b>Packages</b>	Dependencies only between packages will be created.
	<b>NOTE:</b> Method bodies are not parsed on dependency search. Only static information is used.

If you have a code set combined from several files, you may see changes you wish to model without reversing all the code. Only changed files should be reversed. This type of reversing can be done by clicking the Refresh button on the set shortcut menu, or by performing model refresh from the Code Engineering Sets dialog box.

## Rules of the association or attribute creation on reverse

Association or Attribute creation on reverse is ability to enter rules that help to decide if an association or attribute must be created on reverse.

Creating association or attribute creation rules on reverse

1. In the MagicDraw Browser, select the code engineering set for reverse.
2. From the set shortcut menu, choose the **Reverse** command. The **Reverse Options** dialog box is opened.
3. In the **Create class fields as** group, select the **According to rules** radio button. Then press the "..." button. The **Class Field Creation Rules** dialog box opens (see Figure 8 on page 16).

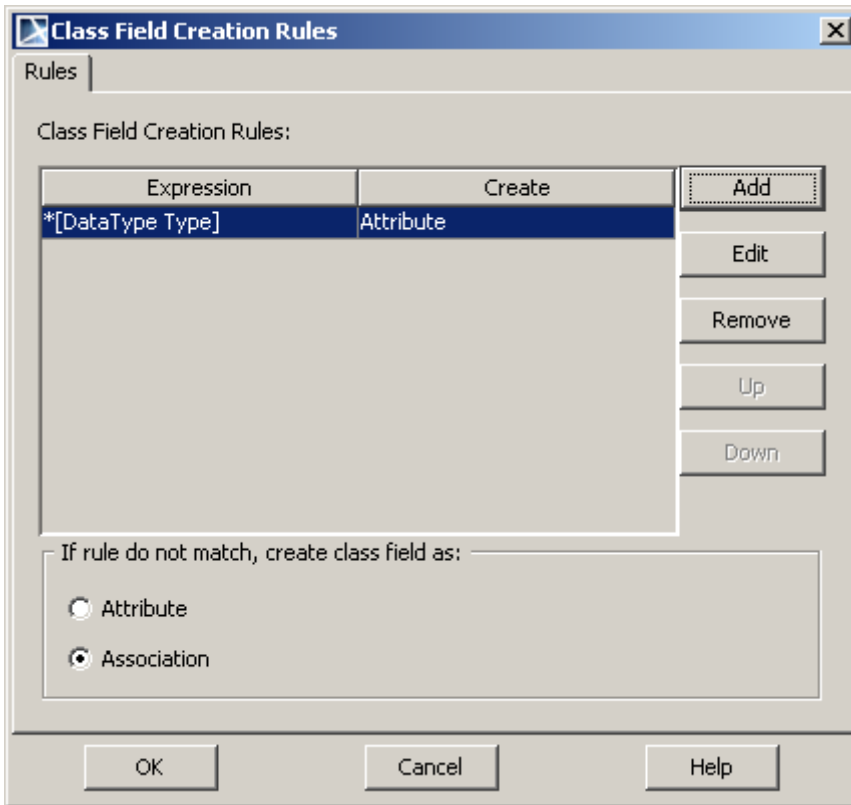


Figure 8 -- The Class Field Creation Rules dialog box

#### Managing association or attribute creation on reverse rules

The **Class Field Creation Rules** dialog box lists the described rules. If type qualified name matches any of the rule, then specified type is created. Rules described in this dialog are executed in order from the top to bottom. If one rule matches, no others are executed. Dialog allows to add a new rule, edit or remove an old one, order rules.

If no rules are matched, attribute or association according the radio button choice at the bottom of dialog is created.

Press the **Add** button for entering expression. On editing success new rule will be created. The **Rule** dialog box opens.

Press the **Edit** button for editing selected rule. Button is disabled, if no rule is selected. The **Rule** dialog box opens.

#### Defining rules

You can define rules in the **Rules** dialog box (see Figure 9 on page 17). To invoke this dialog box, in the **Class Field Creation** dialog box, press the **Add** or **Edit** button.

The **Rules** dialog box description:

- In the **Create** drop down list select to create **Attribute** or **Association** on reverse if property matches defined critters.
- The **Qualified Name** expression field supports simple search patterns - supports '\*' and '?'

- The Qualified Name expression pattern can contain qualified name of the target element, which contains separator of UML separator style (::) and qualified name is counted to the first package with <<modelLibrary>> stereotype
- The **Element Type** drop down list contains subset of UML Type names: Any, Class, Interface, DataType, Primitive, Enumeration, Stereotype values are displayed.
- If in the **Element Type** drop down list **Any** value is selected, the element type will be matched by qualified name expression.

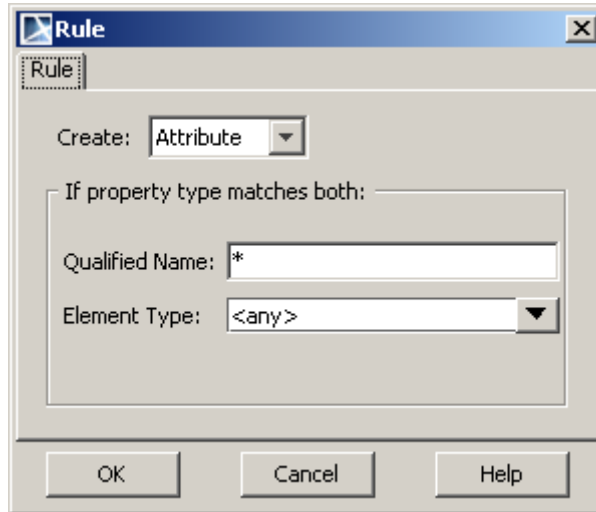


Figure 9 -- The Rule dialog box

### Sample of the association creation on reverse

In these two samples it will be presented how to create associations among my reversed classes, but avoid creating associations to the used libraries (java).

Suppose you have your classes placed inside the package "com::myProduct". Creation rules must be following:

- Create "Association" if property type qualified name match "com::myProduct::\*" with "all" element type
- If rules do not match, create class fields as attributes.

Suppose that your classes are placed inside two packages "myClient" and "myServer". Creation rules should be:

- Create "Association", if type name "match" path "myClient::\*" with "all" meta type
- Create "Association", if type name "match" path "myServer::\*" with "all" meta type
- If rules do not match, create class fields as attributes.

## Global options for Code Engineering

### Code engineering options for all sets in your project

From the **Options** menu, choose **Project**. The **Project Options** dialog box appears.

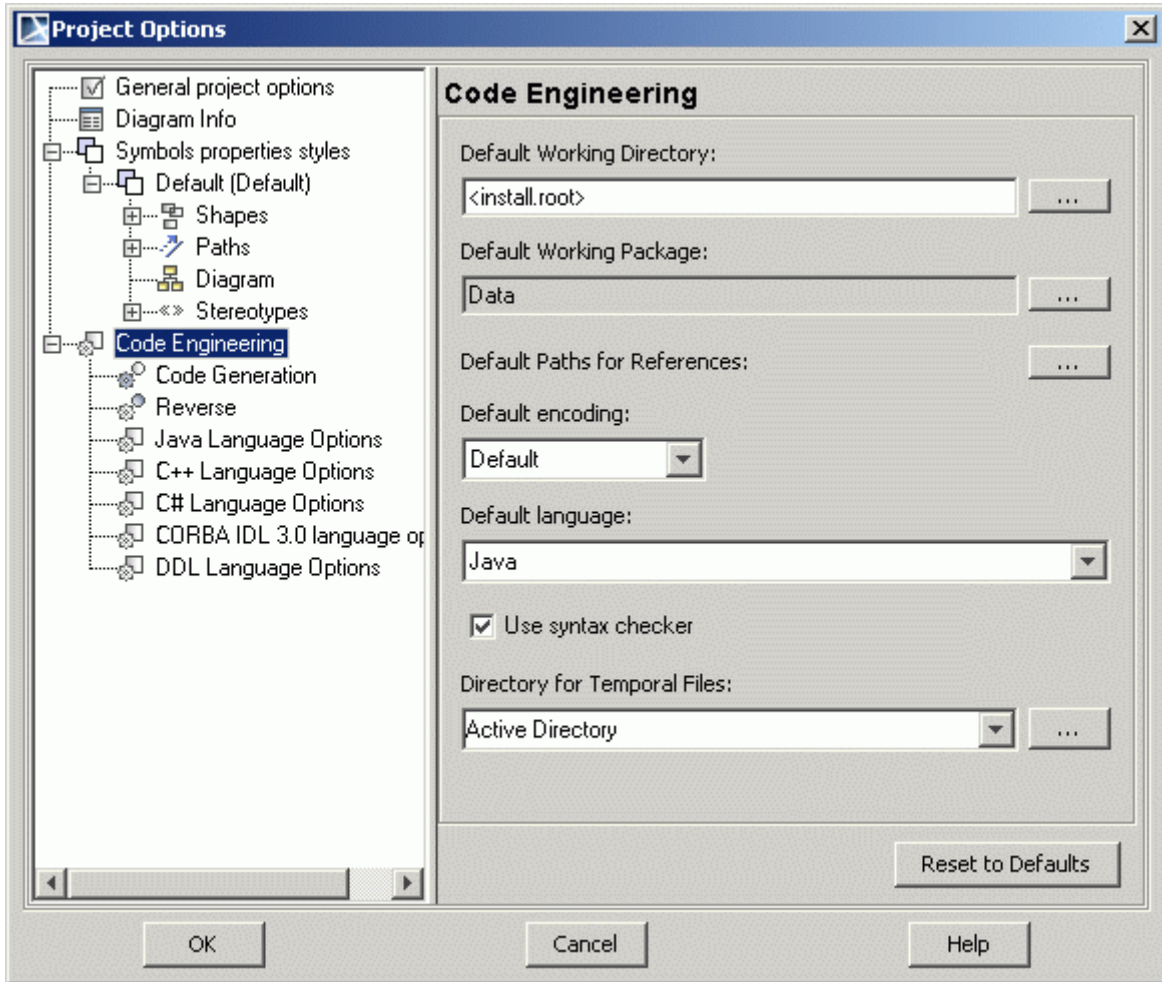


Figure 10 -- Project Options dialog box

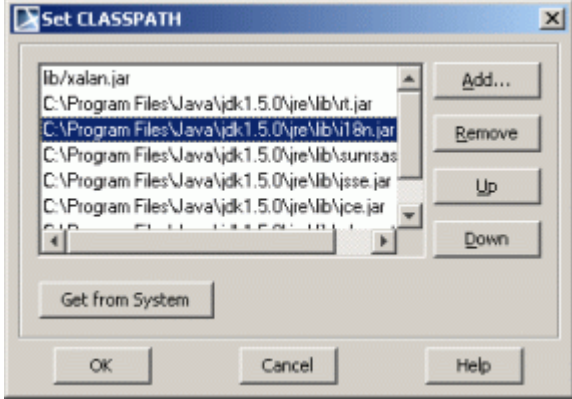
The **Project Options** dialog box has two main collections of customizable options, which are represented by the hierarchy tree on the left side of the dialog box:

- **Styles** – expands the tree hierarchy of all the styles defined within the project. You may use as many of these styles as you wish. See MagicDraw main User’s Manual, working with Projects Section.
- **Code engineering** – these options are found on the right side of the Project options dialog box:
  - **Default Working Directory** field - type the name or browse by clicking the button in the working directory.
  - **Default Working Package** - allows to define any package for reverse output or code generation. Model will be reversed or code generated from this specified package.
  - **Default Paths for References** - add specific profiles, modules, libraries to define where to search paths for references during reverse/code generation.
  - **Default Encoding** - a list of available encodings appears.

- **Default language** drop-down box – select the default generation language.
- **Use Syntax Checker** check box – when selected, the syntax checker runs while Code Engineering is executed
- **Directory for Temporal Files** - it can be **Active Directory**, **System** or define other by clicking “...” button..

Tab name	Description
<b>Code generation</b>	Set code generation options using the fields listed in the right side of the Project options dialog box. The Code generation area contains boxes that have the same functionality as in the Code generations options dialog box (see “MagicDraw Code Engineering File (*.mdr)” on page 10).
<b>Reverse</b>	Set reverse options for all reverse actions of the project using the options listed on the right side of the Project options dialog box. The Reverse area contains boxes that have the same functionality as in the Reverse options dialog box (see “Reverse” on page 13).

Tab name	Element name	Function
<b>Java Language Options</b> Set the generated code style for Java programming languages in the <b>Default</b> language field found on the right side of the <b>Project Options</b> dialog box.	<b>Generate opening bracket in new line</b>	Opens a bracket in the new line that is being generated.
	<b>Generate spaces</b>	Generates spaces inside an assignment and other operators.
	<b>Generate empty documentation</b>	Comment brackets are placed in your code, unless class in the model has no documentation.
	<b>Automatic “import” generation</b>	Automatic generation of "import" sentences according to classes that are referenced in the generated class.
	<b>Class count to generate import on demand</b>	Specify number of classes imported from a single package until all statements importing a single class are substituted with a statement importing an entire package.
	<b>Documentation Processor</b>	After selecting <b>Java Doc</b> processor, click the “...” button to open the <b>Documentation Properties</b> dialog box.
	<b>Style</b>	Two styles are available for documentation.

Tab name	Element name	Function
	<p><b>Use CLASSPATH</b></p>	<p>The '...' button is activated. Search a classpath for importing sentences generation in the Set classpath dialog box.</p>  <p><i>Figure 11 -- Set classpath dialog box</i></p> <p>Click the <b>Get from System</b> button to get CLASSPATH variable defined by operating system or click the Add button and select the classpath directory in the <b>Add Classpath</b> dialog box.</p>
	<p><b>Java Source</b></p>	<p>Available choices <b>1.4</b> or <b>5.0</b></p>
	<p><b>Header</b></p>	<p>Add the specific header to all your code files. Click the button and enter header text in the <b>Header</b> screen.</p> <p>You may also define \$DATE, \$AUTHOR, and \$TIME in the header.</p>
<p><b>C++ Language Options</b> Set the generated code style for C++ programming languages.</p>	<p><b>Generate opening bracket in new line</b></p>	<p>Opens a bracket in the newly generated line.</p>
	<p><b>Generate spaces</b></p>	<p>Spaces inside an assignment and other operators are generated.</p>
	<p><b>Generate empty documentation</b></p>	<p>Comment brackets are placed in your code, unless class in the model has no documentation.</p>
	<p><b>Generate methods body into class</b></p>	<p>Select check box to generate methods body into class.</p>
	<p>Documentation Style</p>	<p>Two styles are available for documentation.</p>
	<p><b>Use include path</b></p>	<p>Click the '...' button and then specify the path for the includes in the <b>Set Include Path</b> dialog box.</p>
	<p><b>Use explicit macros</b></p>	<p>Select check box. The '...' button is activated, click it and in the <b>C++ Macros</b> dialog box use a set of predefined macros.</p>
	<p><b>Header</b></p>	<p>Add the specific header to all your code files. Click the "..." button and enter header text in the <b>Header</b> screen.</p> <p>You may also define \$DATE, \$AUTHOR, and \$TIME in the header.</p>



# INTRODUCTION

## Global options for Code Engineering

Tab name	Element name	Function
<b>CORBA IDL 3.0 Language Options</b>	<b>Generate documentation</b>	Includes the documentation of an element in the comment.
	<b>Generate opening bracket in new line</b>	Opens a bracket in the new line generating.
	<b>Generate spaces</b>	Spaces inside an assignment and other operators are generated.
	<b>Generate empty documentation</b>	Comment brackets are placed in your code, unless class in the model has no documentation.
	<b>Generate imports</b>	Generation of "import" statements for classes that are referenced in the generated class.
	<b>Generate preprocessor directives</b>	Generates pre-processors directives.
	Documentation Style	Three styles are available for documentation.
	<b>Header "..."</b>	Add the specific header to all your code files. Click the "..." button and enter header text in the <b>Header</b> screen. You may also define \$DATE, \$AUTHOR, and \$TIME in the header.
<b>Set Include Path</b>	Specify the path for the "includes". Click the "..." button to open the <b>Select Folder</b> dialog box.	
<b>DDL Language Options</b>	<b>Generate opening bracket in new line</b>	Opens a bracket in the new line generating.
	<b>Generate spaces</b>	Spaces inside an assignment and other operators are generated.
	<b>Generate documentation</b>	Comment brackets are placed in your code, unless class in the model has no documentation.
	<b>Header</b>	Add the specific header to all your code files. Click the button and enter header text in the Header screen. You may also define \$DATE, \$AUTHOR, and \$TIME in the header.
<b>C# Language Options</b> Set the generated code style for C# programming languages.	<b>Generate opening bracket in new line</b>	Opens a bracket in the newly generated line.
	<b>Generate spaces</b>	Generates spaces inside an assignment and other operators.
	<b>Generate empty documentation</b>	Comment brackets are placed in your code, unless class in the model has no documentation.
	<b>Generate required "using" directives</b>	Automatic generation of "using" directives. This option facilitates the usage of namespaces and types defined in other namespaces.

# INTRODUCTION

## Global options for Code Engineering

---

Tab name	Element name	Function
	<b>Concatenate namespace names</b>	If not selected namespace names are separated into several lines. e.g. <pre>namespace A { namespace B {</pre>
	<b>Documentation</b> : • Processor  • Style	<ul style="list-style-type: none"><li>• Use C# XMI processor then generates c# xmi documentation for commenting the code.</li><li>• Select one of the listed comment styles.</li></ul>
	<b>Header</b>	Adds the specific header to all your code files. Click the '...' button and type header text in the <b>Header</b> dialog box. You may also define \$DATE, \$AUTHOR, and \$TIME in the header.
	<b>Conditional Symbols</b>	Add the conditional symbols, which can not be recognized and should be skipped during reverse. Click the '...' button and add conditional symbols in the <b>Define Conditional Symbols</b> dialog box.

## Java Documentation Properties dialog box

To open the **Java Documentation Properties** dialog box

In the **Project Options** dialog box, **Java Language Options** group, select the **Java Doc** processor in the **Documentation** field and click the “...” button to open the **Documentation Properties** dialog box.

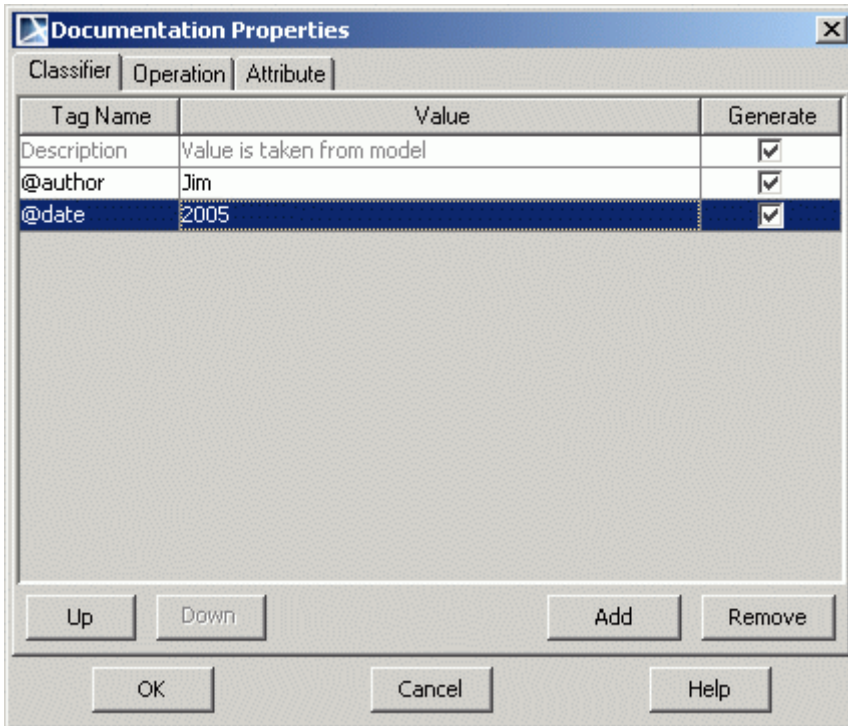


Figure 12 -- Documentation Properties dialog box

Box Name	Function
Tag Name	Type a tag name.
Value	Type the value of the tag.
Generate	The selected tag will be placed in the generated code as a comment before classifier (class or interface), operation or attribute.
Up	Moves the selected item up the list.
Down	Moves the selected item down the list.
Add	Adds a new item in the list.
Remove	Removes the selected item from the list.
OK	Saves changes and closes the dialog box.
Cancel	Closes the dialog box without saving changes.
Help	Displays MagicDraw Help.

## Round Trip

MagicDraw round trip keeps your code and model synchronized, and because Round trip traces all the model and code changes, you may freely change entity specifications without discarding code changes made outside the tool.

For example, Round Trip prevents a job from being damaged by code additions or changes when these steps are followed:

Within the tool, class Base is created.

1. Operation getInstance is added to class.
2. Code is generated
3. With external tool, programmer adds code to that operation.
4. With MagicDraw UML, operations name is changed to Instance.
5. Code is generated.

If the tool rewrites the whole code, these changes are made without corrupting the programmer's job. The name of the operation is changed, but the internals remain the same.

Round trip catches all changes in your project and controls the following actions:

---

- If the source code is not changed, it is not allowed to refresh UML model. The **Refresh** command from the set shortcut menu is unavailable.
- If the model is changed but the code remains the same (new members were added or their headers were changed), refresh is not allowed, and the **Refresh** command from the set shortcut menu is unavailable. When generating code according to changes, all changes in the model are written to the signatures of class members, leaving the old implementation in place.
- If the code is changed but the model remains the same, refresh can be executed: code will be reversed to the UML models. If the **Code Generation Options** dialog box appears when you are attempting to generate code, you may select a code action that differs from the UML model.
- If the code and model are changed while refreshing, all changes in the code are treated as new items and added to the model.
- If data in the model file is deleted, it will be restored while refreshing, even when the code has not been changed or the data itself is unimportant.

## Type Mapping Table

Languages supported by MagicDraw UML have their own built-in types. One language's type might have no matches in another language, or it might have multiple matches. Additionally, some names are interpreted differently in different languages. When performing code generation, therefore, problems may occur when switching between different languages. To avoid this, MagicDraw UML uses type-mapping tables to manage mapping between languages. It describes the rules of how one language's built-in types are converted to those of another language

## Files of Properties

The code can be generated out of prepared UML models. The mapping between the identifiers, used in the UML model and the language to which the model is being generated, should be implemented. This mapping includes the following sections:

- Build-in types (their default values)
- Generalization types
- Possible class declarations. Attributes and operations declaration and visibility modifiers
- Code generation options.

# INTRODUCTION

## Files of Properties

---

The separate prop file is created for every language that is supported by MagicDraw. Files are located in the <MagicDraw installation directory>/data folder. The file name pattern is lang.prop, where lang stands for the name of the programming language.

<b>Supported language</b>	<b>File of Properties</b>
<b>JAVA</b>	java.prop
<b>C++</b>	C++.prop
<b>CORBA IDL</b>	idl.prop
<b>JAVABYTECODE</b>	javabytecode.prop
<b>DDL</b>	ddl.prop
<b>CIL</b>	cil.prop
<b>CIL Disassembler</b>	cil disassembler.prop
<b>C#</b>	c#.prop
<b>EJB</b>	ejb.prop
<b>EJB 2.0</b>	ejb20.prop
<b>IDL</b>	idl.prop
<b>XML Schema</b>	xmlschema.prop
<b>WSDL</b>	wsdl.prop

Files of language properties are separated into sections where all related elements are grouped. You may edit existing entities, add new ones, and change the default values.

We strongly recommend that you edit default values only. In general, all the sections have the list of possible and default values for the element.

# JAVA CODE ENGINEERING

## Introduction

Java Code Engineering chapter describes how Java language elements are mapped to UML by MagicDraw, what profiles to use and describes Java code engineering properties.

Chapter "Java Mapping to UML", on page 26 describes general rules how each Java element is mapped to UML by MagicDraw and what profile is used. You will find an example and corresponding model in MagicDraw with marked properties used in Java to describe mapping rules.

Chapter "Java CE Properties", on page 56 introduces specific Java options.

## Abbreviations

<b>UML</b>	Unified Modeling Language
<b>JLS</b>	Java Language Specification
<b>CE</b>	Code Engineering
<b>CES</b>	Code Engineering Set
<b>JVM</b>	Java Virtual Machine

## References

JLS third edition

## Java support in MagicDraw

You may perform the following actions with MagicDraw:

- Import Java source code into model (reverse engineering).
- Generate Java code from the model (code generation).
- Apply changes to the source code from the model (round-trip). You may change Java declaration headers and apply them to already existing source code, however you cannot change method implementation.
- Create sequence diagram from the selected method body.
- Create model from the Java byte code.

## Java Mapping to UML

### Java Profile

UML specification does not provide elements to cover fully JLS, therefore MagicDraw is using UML stereotypes to mark UML class or interface to be some specific Java element.

Java stereotypes are provided in Java\_Profile.xml file in MagicDraw profiles directory. Some stereotypes have tagged values used for mapping special Java language elements or keywords, which are not mapped to the standard UML properties. All Java stereotypes are derived from the <<JavaElement>>. Another abstract stereotype <<JavaTypeElement>> is used to group all Java type elements. These two stereotypes are abstract and are not used directly.

Each other stereotype is used to represent appropriate Java element: <<JavaImport>> represents Java import, <<JavaOperation>> - Java operation, <<JavaClass>> - Java class type, etc.



Figure 13 -- Stereotypes

### Java referenced types

Java built-in types are used from the “UML Standard Profile”, which is automatically loaded with every new project.

**NOTE** UML Standard Profile by default is hidden. If you want see it, click the **Show Auxiliary Resources** button in the Browser.

Every referenced class from the other libraries (including JDK libraries) should be imported/created into project and referenced in CES reference path (by default reference path is “Data” package in the model).

MagicDraw resolves referenced classes from the specified class path (by default class path is boot class path taken from the JVM on which MagicDraw is started) and creates in appropriate package structure (for more information see "Sorting reversed classes according to the classpath", on page 28).

If referenced class is not found nor in model neither in class path, reference is created in the “Default” package.

Java Profile defines UML Class with a name “?”. It is used for mapping parameterized types.

#### Sorting reversed classes according to the classpath

References are sorted to packages according to the classpath entry.

See a sample in Figure 14 on page 28, there reverse result is represented of the following java code :

```
import javax.swing.*;
public class CustomFrame extends JFrame
{
    String mTitle;
}
```

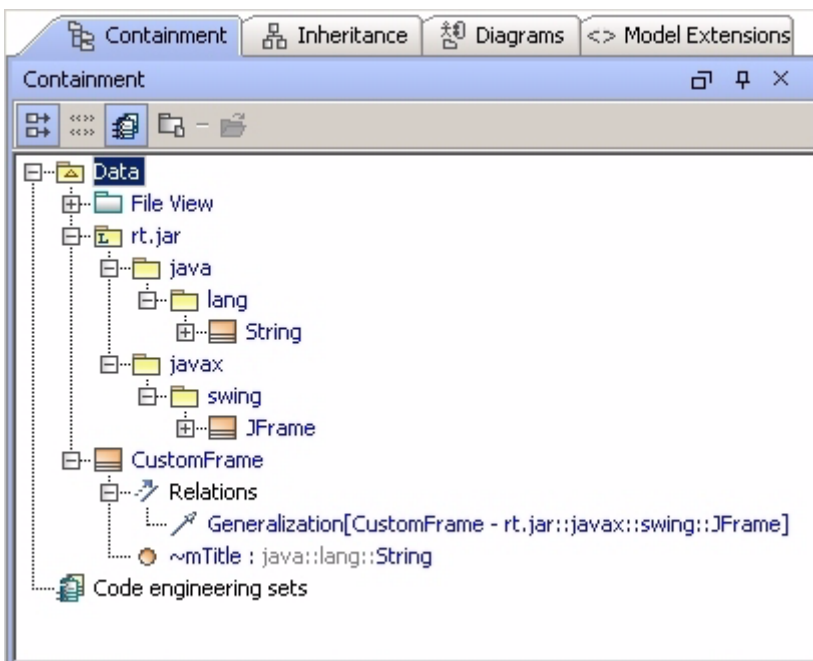


Figure 14 -- Sample: The String, JFrame and CustomFrame classes location after reverse



## Mapping to UML rules

### Package

Java package is mapped to the general UML package. It does not have any specific stereotypes and properties. However if UML package represents Java package, it must not have <<modelLibrary>> stereotype from standard UML profile. <<modelLibrary>> stereotype is used to show root package, where Java package tree ends and all parent packages, including <<modelLibrary>> are not part of Java package structure. In picture below is “java.lang.String” added into the “working package”. Packages “java” and “lang” represents general Java packages, but “working package” with “L” is stereotyped with <<modelLibrary>> and is not part of the Java.

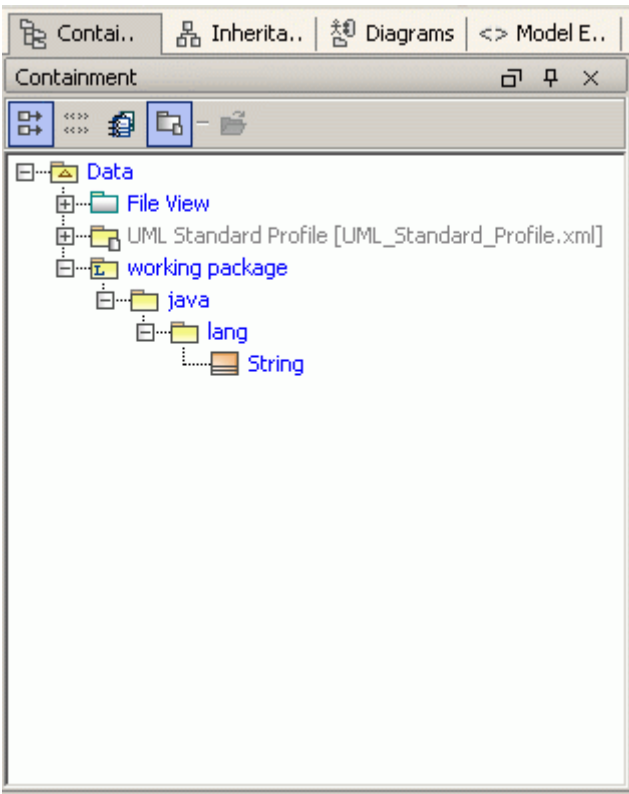


Figure 15 -- Package Structure

### Class

Java class is mapped directly to the UML Class with stereotype <<JavaClass>>. This stereotype is optional and if UML class doesn't have any stereotype, Java CE treats it as Java class. Class modifiers are mapped into UML Class properties or to the Java language properties for class, if no appropriate property is found in UML.

Java class fields, operations and inner classes are mapped to the appropriate UML Properties, UML Operations and UML Classes

Class mapping table

Java element	MagicDraw-UML element
Class declaration	UML Class with stereotype <<JavaClass>> (optional)
Class name	UML Class name

Java element	MagicDraw-UML element
Class documentation	UML Class Documentation
Class e	Is mapped to the UML Generalization relationship, where supplier is extended class and client is mapped class
Class implements clause	Is mapped to the UML Interface Realization relationship, where supplier is extended class and client is mapped class.
Visibility modifier	UML Class "Visibility" property
Abstract modifier	UML Class "Is Abstract" property
Final modifier	UML Class "Is Final Specialization" property
Static modifier	Java Language property "Static modifier"
Strictfp modifier	Java Language property "Strictfp modifier"

### Example

#### Java Source Code

```
/**
 * Comment of the class MyList
 */
public final class MyList extends ArrayList implements Cloneable
{
}
```

#### MagicDraw UML Model

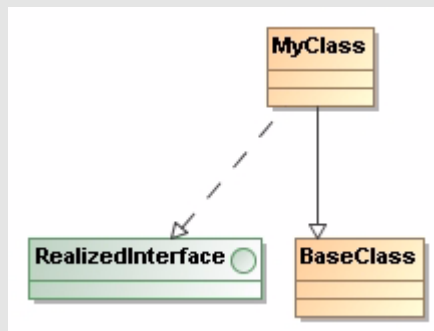


Figure 16 -- Class diagram

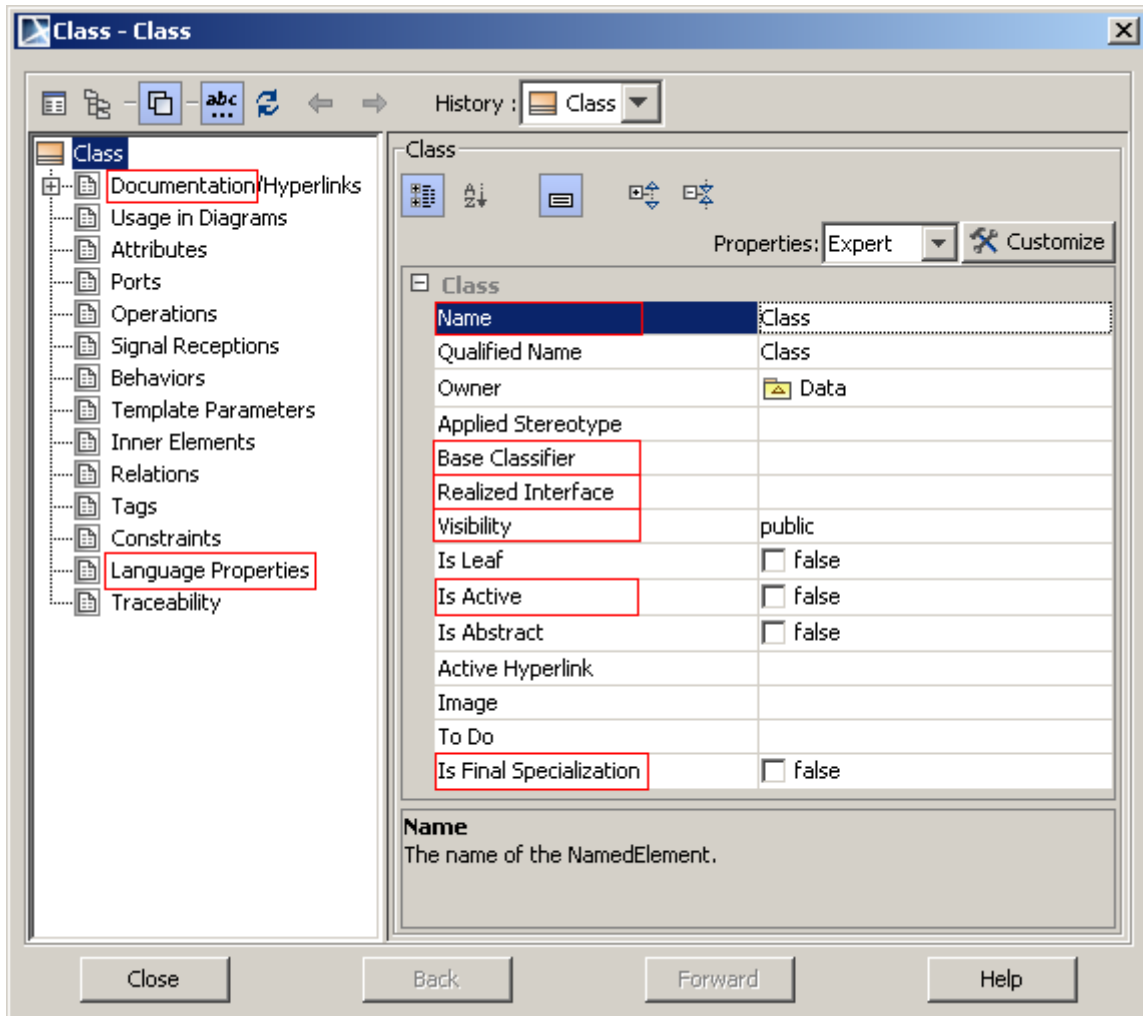


Figure 17 -- UML Class specification dialog

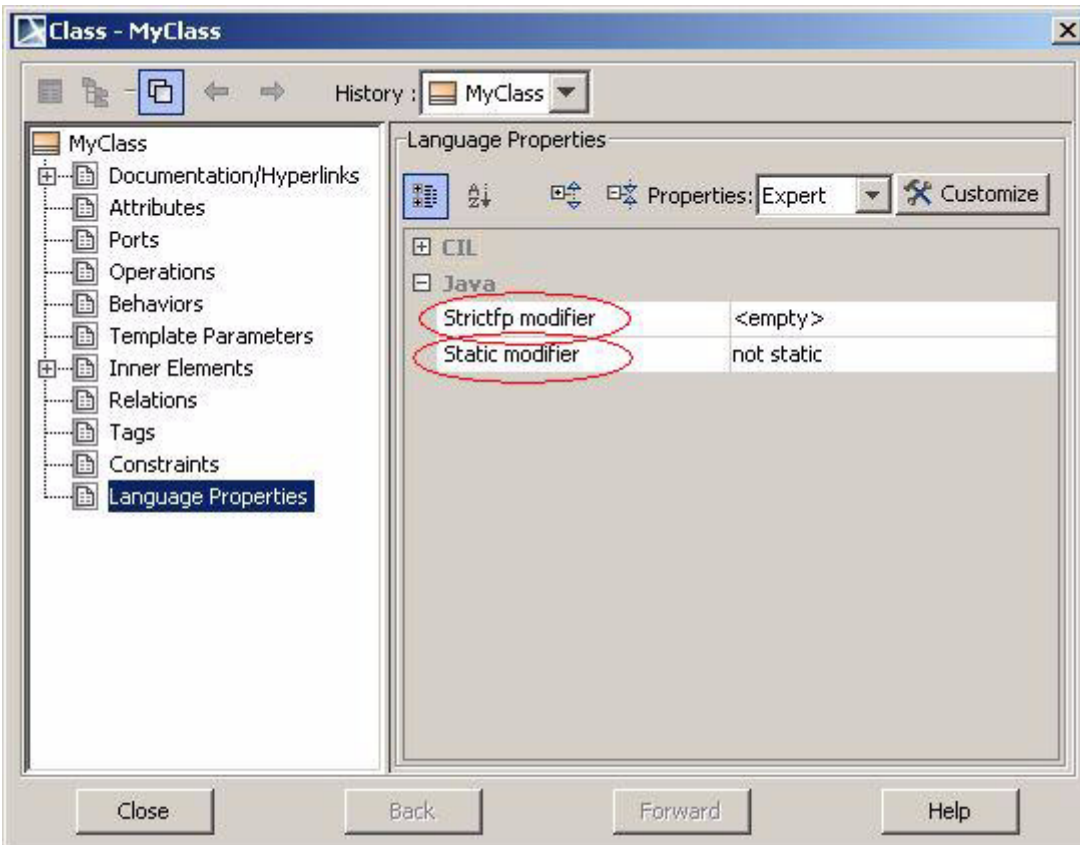


Figure 18 -- UML Class language properties

## Field

Java field is mapped directly to the UML Property with stereotype <<JavaField>>. This stereotype is optional and if UML class doesn't have any stereotype, Java CE treats it as Java field. Field modifiers are mapped into UML Property properties or to the Java language properties, if no appropriate property is found in UML.

Worth to know, that Java field type modifiers is mapped to the MagicDraw specific property "Type Modifier", but not to the UML Multiplicity.

Field mapping table

Java element	MagicDraw-UML element
Field declaration	UML Property, owned by UML Class, with stereotype <<JavaProperty>>(optional)
Field name	UML Property Name
Field documentation	UML Field Documentation
Field type	Is mapped to the UML Type property. It is reference to the UML Classifier, which by its package structure and name represents referenced Java class
Field type modifiers	Is mapped to the MagicDraw specified property "Type Modifier"
Visibility modifier	UML Property "Visibility" property
Final modifier	UML Property "Is Read Only" property

Static modifier	UML Property "Is Static" property
Transient modifier	Java Language property "Transient modifier"
Volatile modifier	Java Language property "Volatile modifier"

### Example

#### Java Source Code

```
public final class MyClass
{
    /**
     * myList comment
     */
    public static java.util.List myList;
}
```

#### MagicDraw UML Model

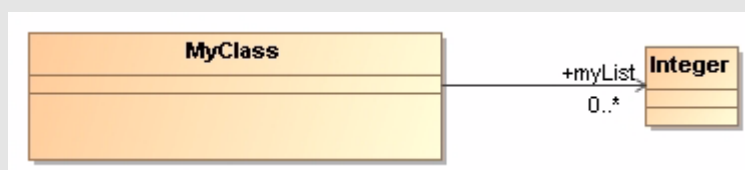


Figure 19 -- Class with property

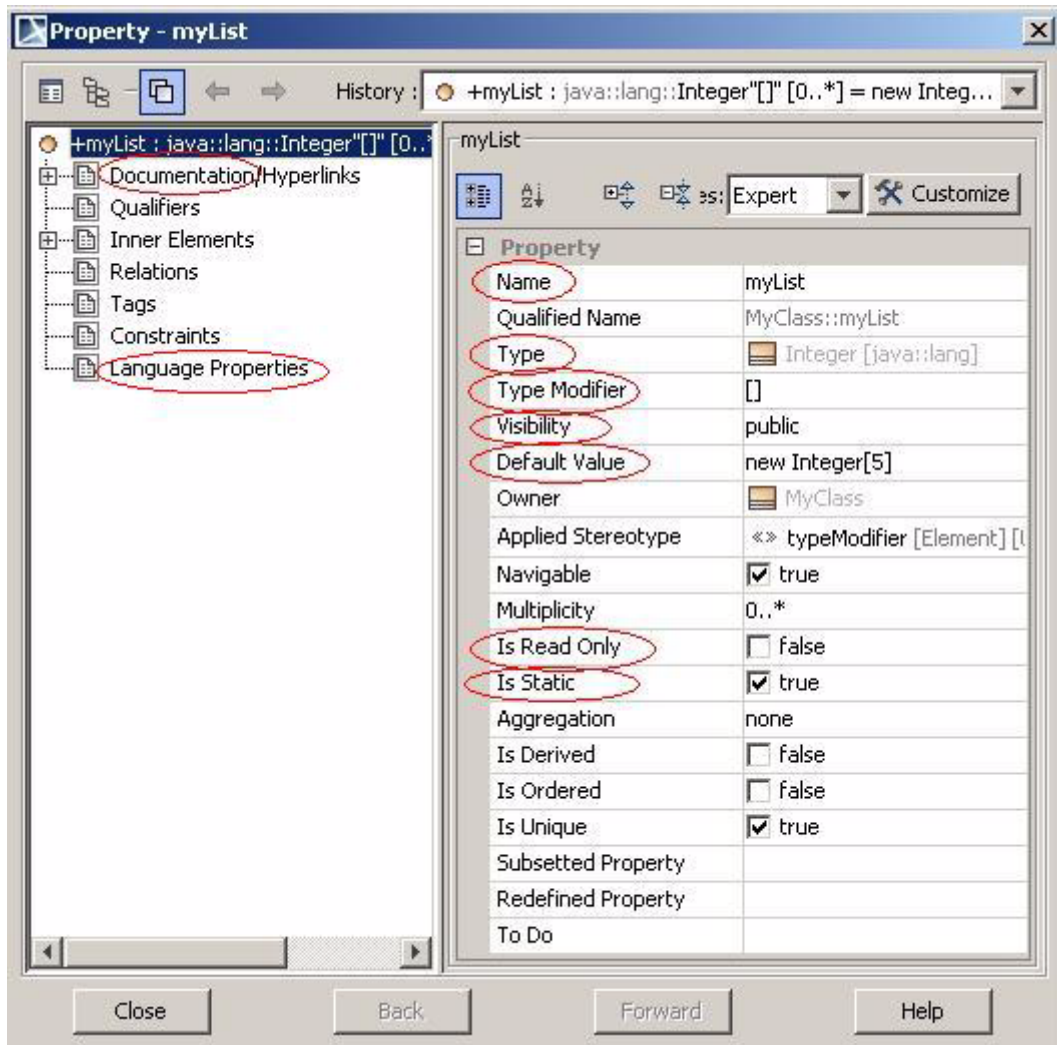


Figure 20 -- UML Property specification dialog

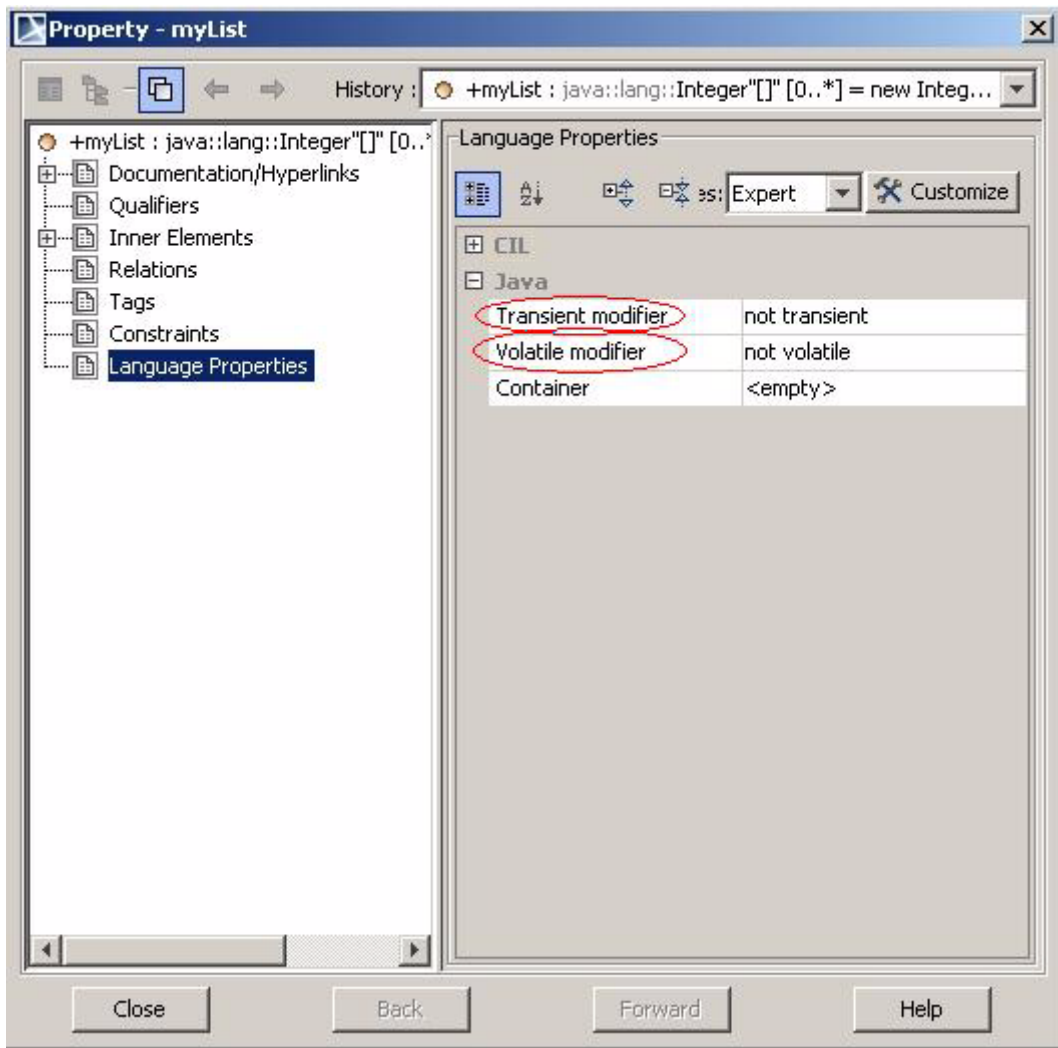


Figure 21 -- UML Property language properties

## Operation

Java operation is mapped directly to the UML Operation with stereotype <<JavaOperation>>. This stereotype is optional and if UML class doesn't have any stereotype, Java CE treats it as Java operation. Operation modifiers are mapped into UML Property properties or to the Java language properties, if no appropriate property is found in UML.

Java operation return type is mapped to the UML Type property of UML Parameter with "Return" direction kind. Java Operation parameters are mapped to the UML Parameters. Direction kind is set "In" for "final" parameters.

Worth to know, that Java parameter type modifiers is mapped to the MagicDraw specific property "Type Modifier", but not to the UML Multiplicity.

Operation mapping table

Java element	MagicDraw-UML element
Operation declaration	UML Operation, owned by UML Class, with stereotype <<JavaOperation>>(optional)
Operation name	UML Operation Name



Operation documentation	UML Operation Documentation
Parameters list	UML Operation "Parameters" list
Return type	Is mapped to the UML Parameter type with "return" direction kind (resides in UML Operation parameters list). It is reference to the UML Classifier, which by its package structure and name represents referenced Java class
Return type modifiers	Is mapped to the MagicDraw specified property "Type Modifier"
Visibility modifier	UML Operation "Visibility" property
Final modifier	UML Operation "Is Leaf" property
Abstract modifier	UML Operation "Is Abstract" property
Static modifier	UML Operation "Is Static" property
Synchronized modifier	UML Operation "Concurrency" kind "guarded"
Throws list	UML Operation "Raised Exception" list. It is list of references to the UML Classes, which by its package structure and name represents referenced Java exception class
Native modifier	Java Language property "Native modifier"
Strictfp modifier	Java Language property "Strictfp modifier"

### Parameter mapping table

Java element	MagicDraw-UML element
Parameter declaration	UML Parameter, owned by UML Operation, with stereotype <<JavaParameter>>(optional)
Parameter name	UML Parameter Name
Parameter documentation	When is used JavaDoc preprocessing, it is mapped to UML Parameter Documentation, else it is part of UML Operation Documentation.
Return type	Is mapped to the UML Type property. It is reference to the UML Classifier, which by its package structure and name represents referenced Java class.
Return type modifiers	Is mapped to the MagicDraw specified property "Type Modifier"
Final modifier	Direction kind "in" of UML Parameter

### Example

#### Java Source Code

```
public class MyList
{
    /**
     * Operation Comment
     */
    public abstract void foo(final List list) throws
    IllegalArgumentException;
}
```

#### MagicDraw UML Model



Figure 22 -- Class with operation

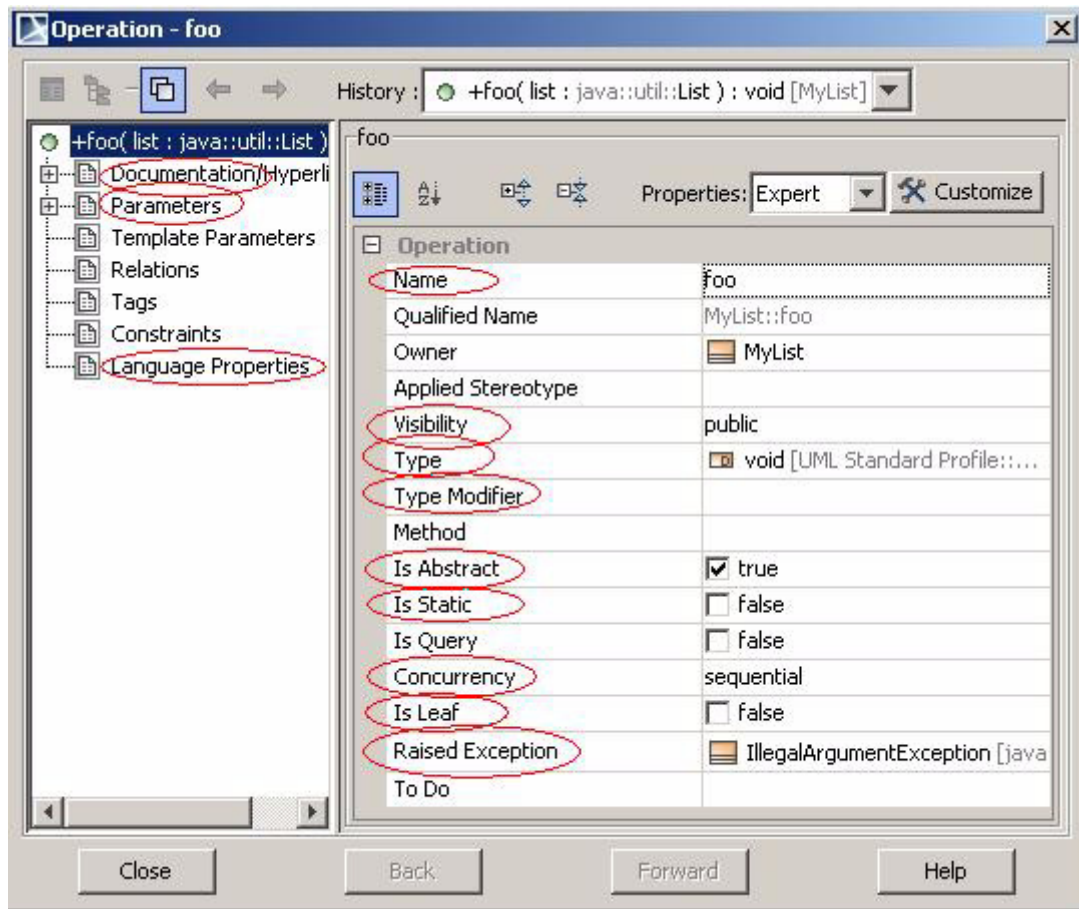


Figure 23 -- UML Operation specification dialog

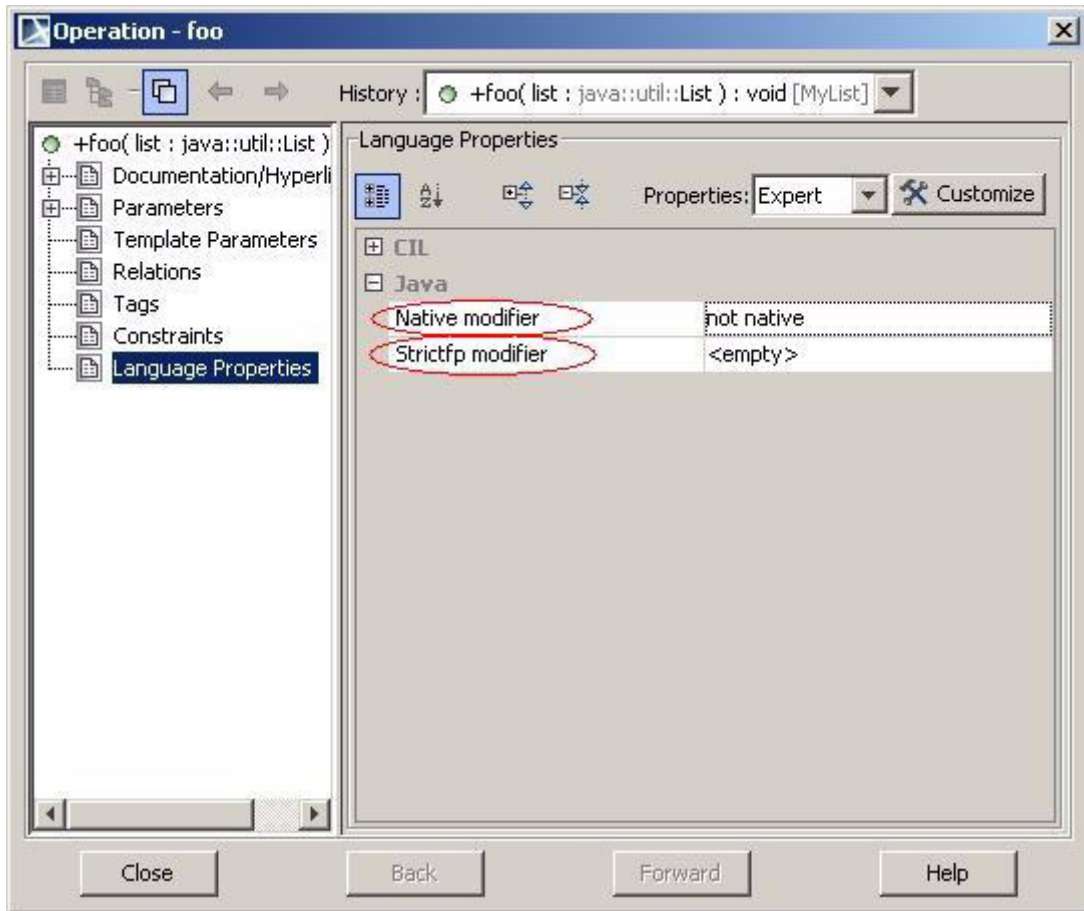


Figure 24 -- UML Operation language properties

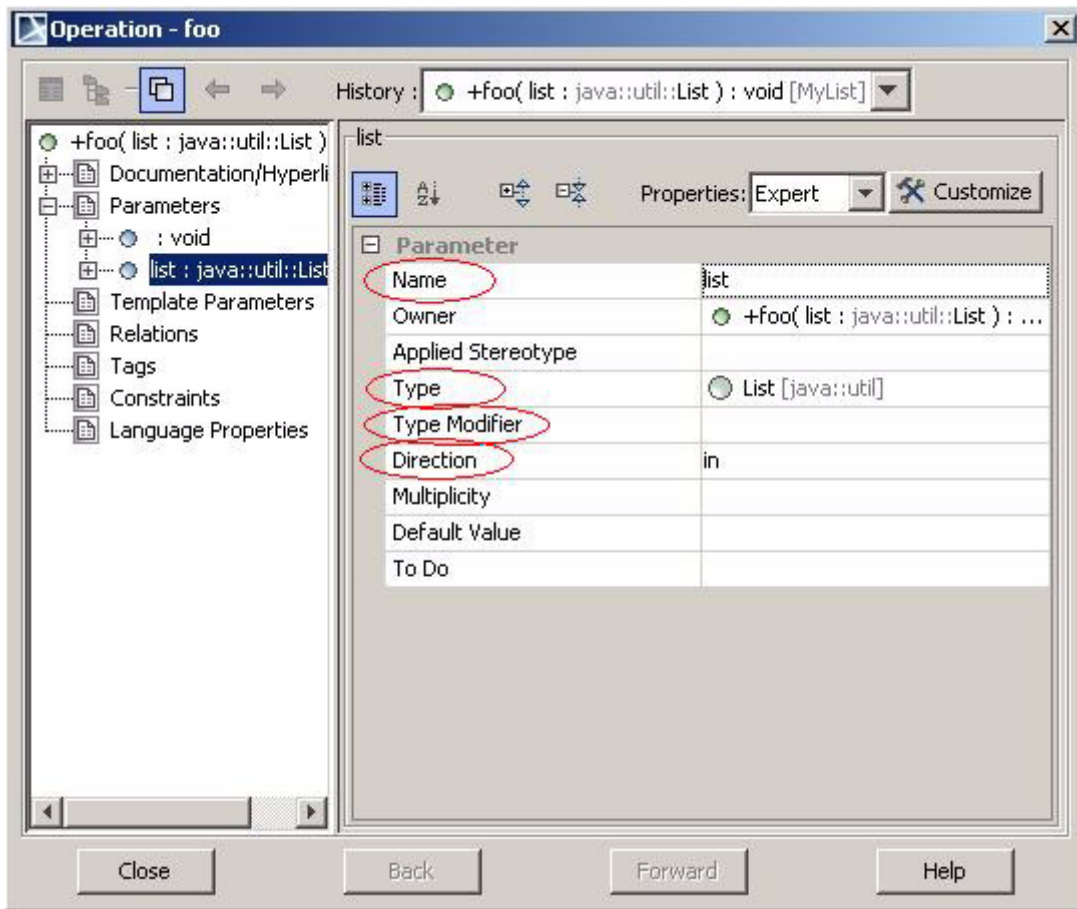


Figure 25 -- UML Parameter specification dialog

## Interface

Java interface is mapped directly to the UML Interface with stereotype <<JavaInterface>>. This stereotype is optional and if UML Interface doesn't have any stereotype, Java CE treats it as Java interface. Interface modifiers are mapped into UML Interface properties or to the Java language properties for interface, if no appropriate property is found in UML.

All mapping rules used in Java class mapping is applicable to the Java interface. See "Class" on page 29.

## Enumeration

Java enumeration is mapped directly to the UML Class with stereotype <<JavaEnumeration>>. Enumeration modifiers are mapped into UML Class properties or to the Java language properties for interface, if no appropriate property is found in UML.

Java enumeration literals are mapped to the UML Property, but with stereotype <<JavaEnumerationLiteral>>. All contained fields, operations and inner classes are mapped to appropriate UML Properties, UML Operations, UML Classes.

All mapping rules used in Java class mapping is applicable to the Java enumeration. See "Class" on page 29.

### Example

### Java Source Code

```
enum MyInterface
{
}
```

MagicDraw UML Model



Figure 26 -- UML Enumeration

## Enumeration Literal

Java enumeration literal is mapped directly to the UML Property with stereotype `<<JavaEnumerationLiteral>>`. It is not required to specify any specific modifiers for enumeration literal.

### Enumeration literal mapping table

Java element	MagicDraw-UML element
Literal declaration	UML Parameter with stereotype <code>&lt;&lt;JavaEnumerationLiteral&gt;&gt;</code> , owned by UML Class with stereotype <code>&lt;&lt;JavaEnumeration&gt;&gt;</code>
Literal name	UML Property Name
Literal documentation	When is used JavaDoc preprocessing, it is mapped to UML Parameter Documentation, else it is part of UML Operation Documentation.

### Example

#### Java Source Code

```
enum MyEnumeration
{
    ONE, TWO, THREE;

    int attribute1;
    String attribute2;
}
```

MagicDraw UML Model

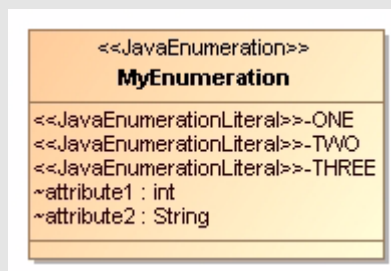


Figure 27 -- UML Class representing Java enumeration with enumeration literals

### Annotation type

Java annotation declaration is mapped directly to the UML Interface with stereotype <<JavaAnnotation>>. Annotation modifiers are mapped into UML Interface properties. Annotation members are mapped to the UML Interface operations with stereotype <<JavaAnnotationMember>>.

Annotation mapping table

Java element	MagicDraw-UML element
Annotation declaration	UML Interface with stereotype <<JavaAnnotation>> (optional)
Annotation name	UML Interface name
Annotation documentation	UML Interface Documentation
Visibility modifier	UML Interface "Visibility" property

#### Example

##### Java Source Code

```
/**
 * Comment of annotation
 */
public @interface Annotation
{
}
```

##### MagicDraw UML Model



Figure 28 -- UML Interface representing Java annotation type

### Annotation Member

Java annotation member is mapped directly to the UML Operation, owned by the interface stereotyped as <<JavaAnnotation>>. Operation, by itself, can have stereotype <<JavaAnnotationMember>>, but it is optional, unless you are going to specify default value for it.

Java annotation member type is mapped to UML Type property of UML Parameter with "Return" direction kind.

Annotation member mapping table

Java element	MagicDraw-UML element
Annotation member declaration	UML Operation with stereotype <<JavaAnnotationMember>> (Optional), owned by UML Interface with Stereotype <<>JavaAnnotation>>.
Annotation member name	UML Operation Name

Annotation member documentation	UML Operation Documentation
Annotation member type	Is mapped to the UML Parameter type with "return" direction kind (resides in UML Operation parameters list)
Annotation member type modifiers	Is mapped to the MagicDraw specified property "Type Modifier"
Default value	{JavaAnnotationMemberDefaultValue} tagged value of <<JavaAnnotationMember>> stereotype. Stereotype is set on UML Operation



**Example**

**Java Source Code**

```
/**  
 * Comment of annotation  
 */  
public @interface Annotation  
{  
    int    id();  
    String name() default "[unassigned]";  
}
```

**MagicDraw UML Model**

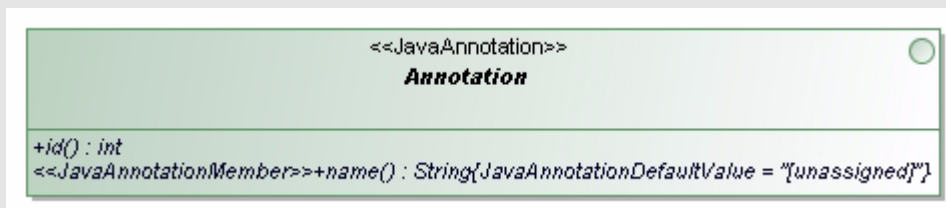


Figure 29 -- UML Interface representing Java annotation type with Java annotation members

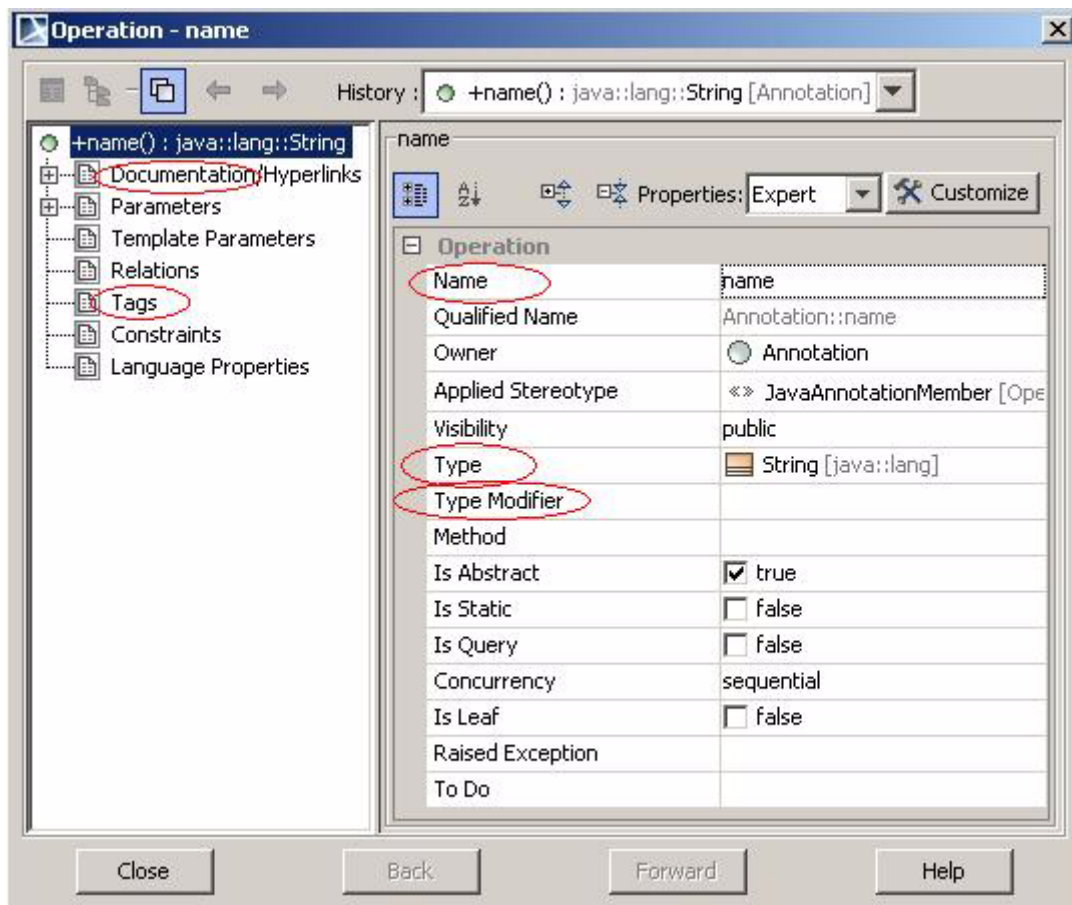


Figure 30 -- UML Operation, representing Java annotation member, specification dialog

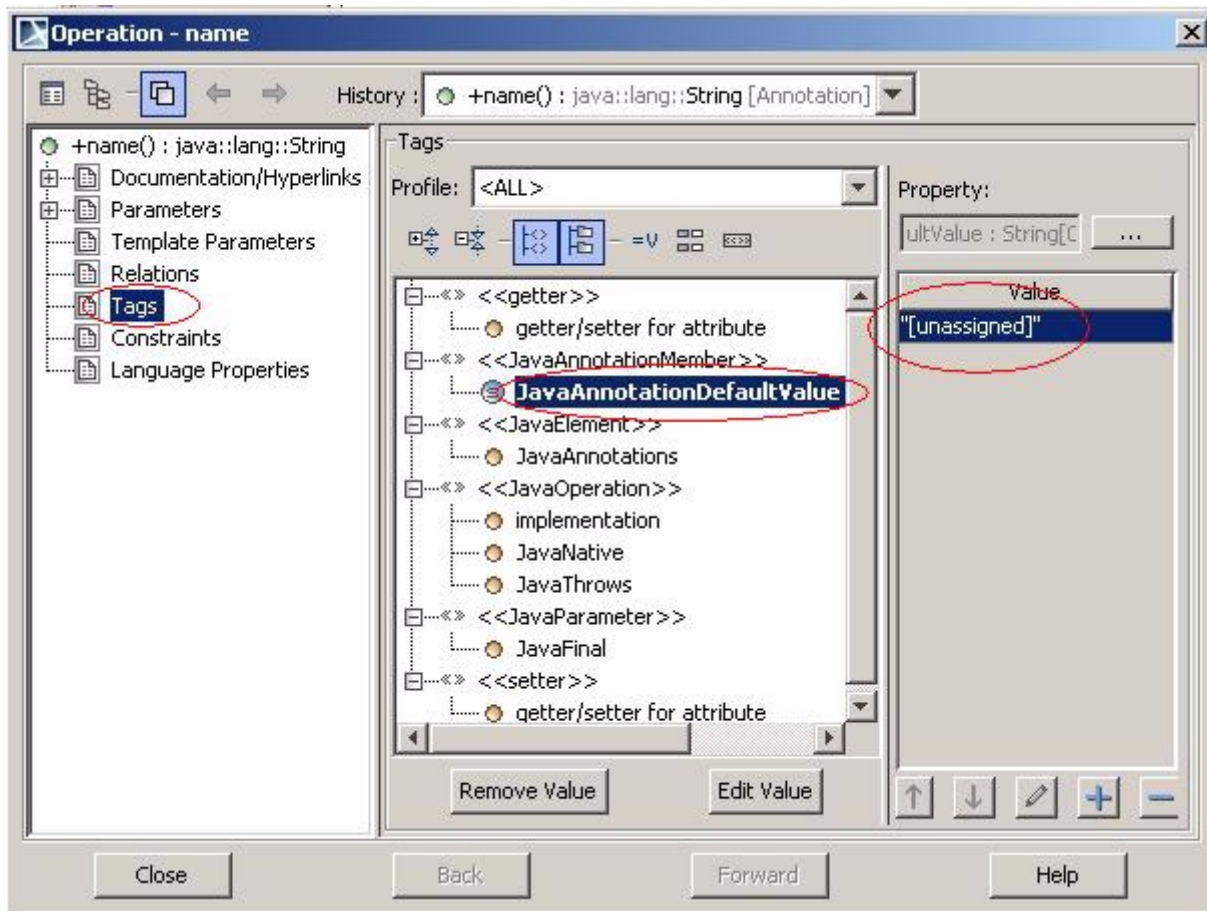


Figure 31 -- Default value set for UML Operation, representing Java annotation member

## Annotations Usage

Java element can be annotated. Such annotation is mapped top the {JavaAnnotations} tagged value of the stereotype <<JavaElement>>. <<JavaElement>> is base stereotype for all stereotypes, used in Java mapping, and it can be used directly or any other stereotype derived from it. Annotation is mapped as simple string value.

### Example

### Java Source Code

```
public class Test
{
    @Annotation
    (
        id      = 2,
        name = "Rick"
    )
    public void foo() {}
}
```

### MagicDraw UML Model

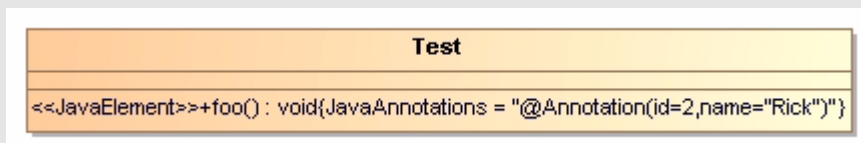


Figure 32 -- Java annotation usage

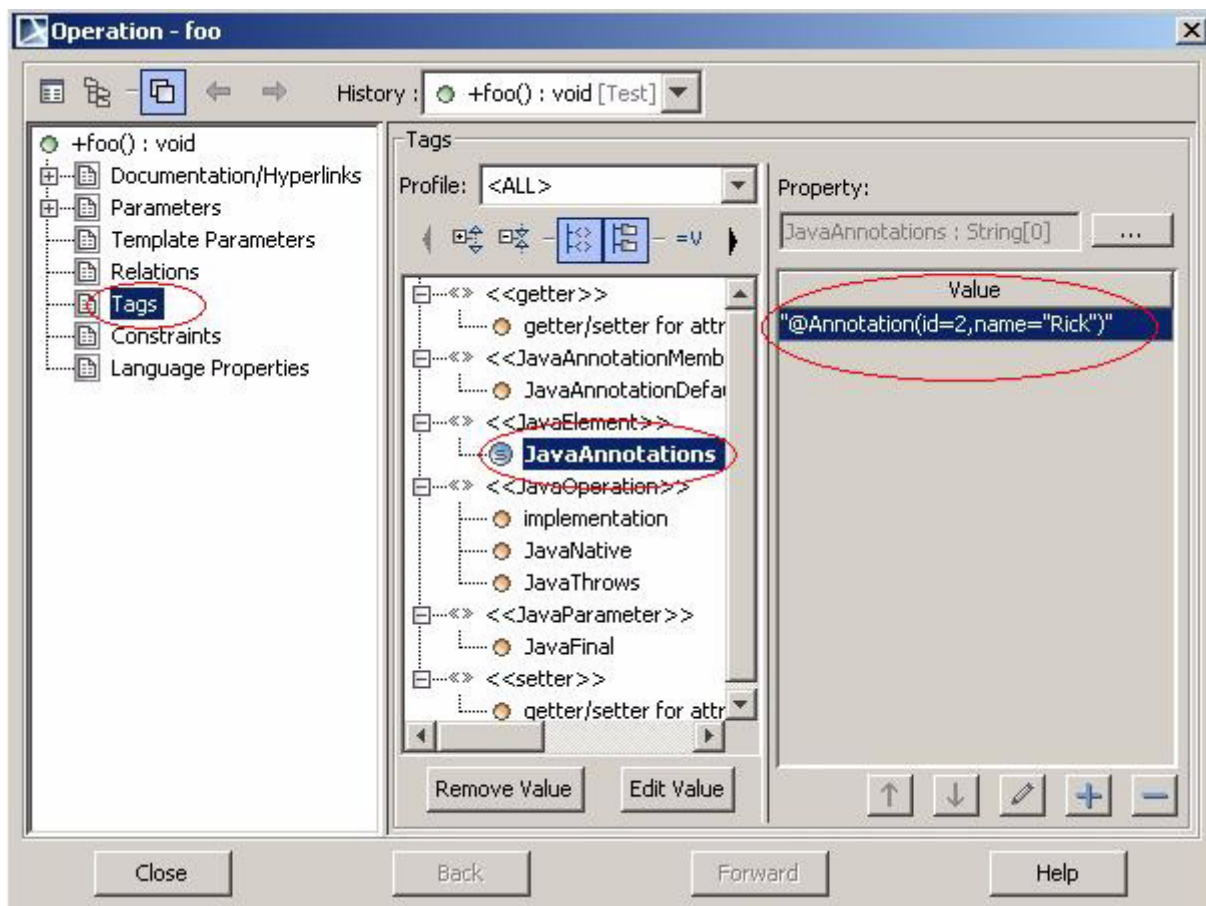


Figure 33 -- Annotation used on operation "foo"

## Type Variables

Type variables are mapped to the UML Template Parameter of UML Class, Interface or UML Operation, regarding to what Java element has type variables. If bound type are present, they are mapped into the UML

Class or UML Interface connected with UML Generalization or UML Interface Realization as a supplier and client is UML Class, which is “Parametered Element” of the UML Template Parameter.

Type variable mapping table

Java element	MagicDraw-UML element
<b>Type variable declaration</b>	Is mapped to the UML Template Parameter. This UML Template Parameter is of the Class type from the UML Metamodel. UML Template Parameter has property “Parametered Element” of the UML Template Parameter. Metamodel type “Class” is taken from the “UML Standard profile/UML 2 Metamodel”
<b>Type variable name</b>	Is mapped to the UML Class name. This UML Class is “Parametered Element”
<b>Type bounds</b>	If bound type is a Java class, it is mapped to the General class of the “Parametered Element”. If bound type is a Java interface, than it is mapped to the Realized Interface.

### Example

#### Java Source Code

```
public class Test <E extends Cloneable>
{
    E attribute;
}
```

#### MagicDraw UML Model

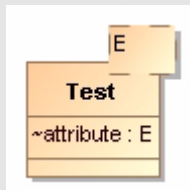


Figure 34 -- UML Class with template parameter E, representing Java type variable

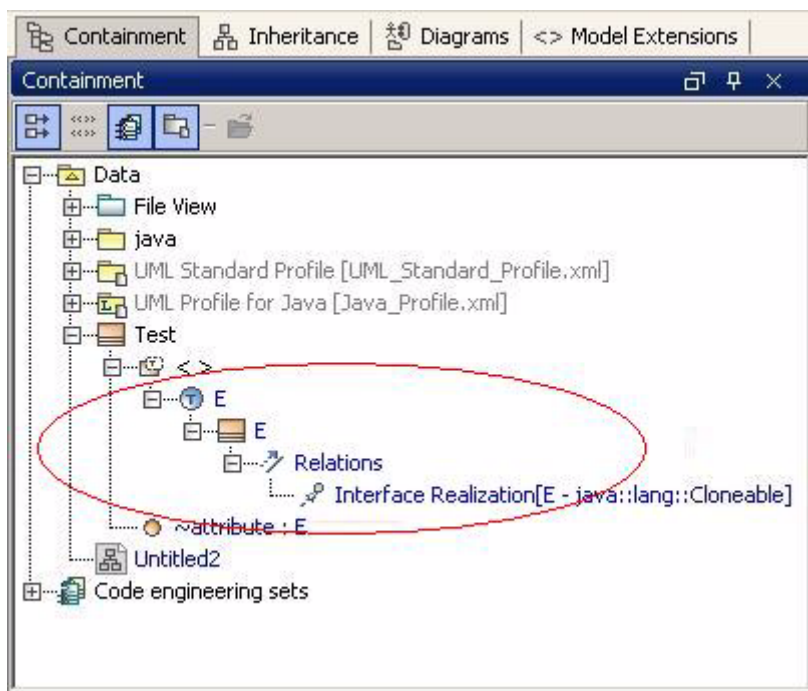


Figure 35 -- Marked UML Template Parameter, with residing UML Class named E (Parametered Element)

Note, that residing UML Class E in UML Template Parameter is realizing interface “java.util.Cloneable” and this class is used as type for UML Class attribute.

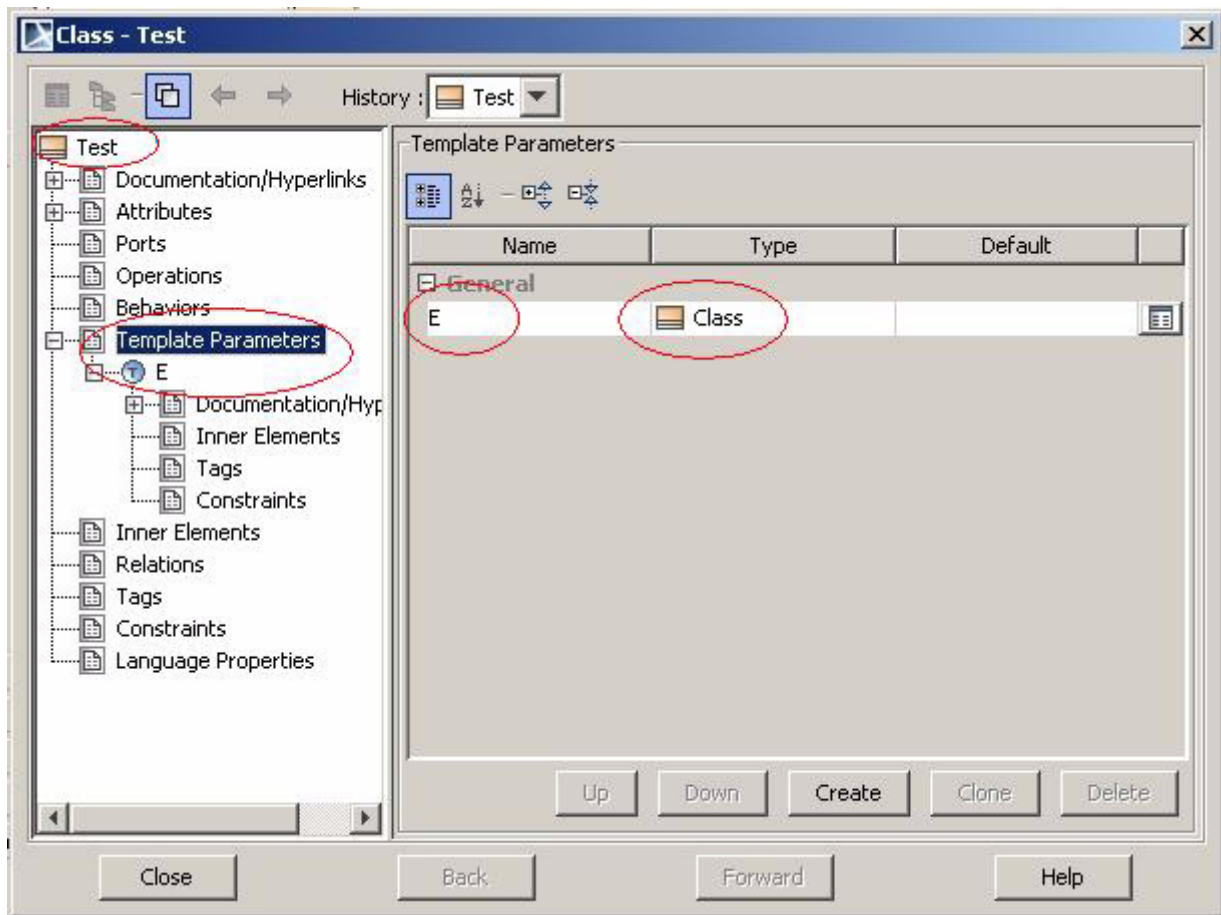


Figure 36 -- Template parameters in UML Class specification dialog

### Parameterized Type

Parameterized types are mapped to the general UML Classifier connected with UML Template Binding to the UML Classifier. Supplier of this binding link is UML Classifier with UML Template Parameters and represents Java generic type with type parameters. Client of UML Template Binding is UML Classifier of the same UML type as supplier is. Java type parameters are mapped directly to the UML Template Parameter Substitution of the UML Template Binding.

Parameterized type mapping table

Java element	MagicDraw-UML element
Parameterized type	In UML it is called bounded elements, which is connected to the UML Classifier by the UML Template Binding as a client. Client must be of the same type as is supplier. Supplier must have at least one UML Template Parameter.
Parameter for type	Is mapped to the UML Template Parameter Substitution of the UML Template Binding. Each UML Template Parameter from the supplier must be substituted by the UML Template Parameter Substitution. Type of type parameter is any reference to the UML Classifier from the model which is set as "Actual" value of the UML Template Parameter Substitution

Type modifiers of the parameter	It is mapped to the MagicDraw property "Type modifiers" of the UML Template Parameter Substitution
Wildcard	UML Class with a name "?" from the Java Profile is used as "Actual" value in UML Template Parameter Substitution.
Wildcard with bounds	Java bounding type is mapped to the UML Classifier and it is used as "Actual" value in UML Template Parameter Substitution. "? extends" or "? super" bounding is mapped to the appropriate tag {JavaArgumentBount} value "extends" or "super" of the <<Java-TypeArgument>> stereotype applied to the UML Template Parameter Substitution.

### Example

When you have type with type variables represented in the model, you can create parameterized type for Java. For this you need to create empty UML Classifier of the type the template classifier (type with type variables) is. Then create a UML Template Binding and create UML Template Parameter Substitution for UML Template Parameters.

In order to create “java.util.List<String>” type, we need to create UML Interface first, with UML Template Parameter representing “java.util.List”.



Figure 37 -- UML Interface representing “java.util.List” with template parameter

Then create another UML Interface and connect with List interface with UML Template Binding.

Note, that in order to draw UML Template Binding, client element must have at least one UML Template Parameter created

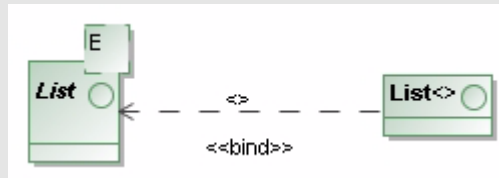


Figure 38 -- UML Template Binding between template class and bounded element

You need to open UML Template Binding specification dialog and create UML Template Parameter Substitution for appropriate UML Template Parameter.



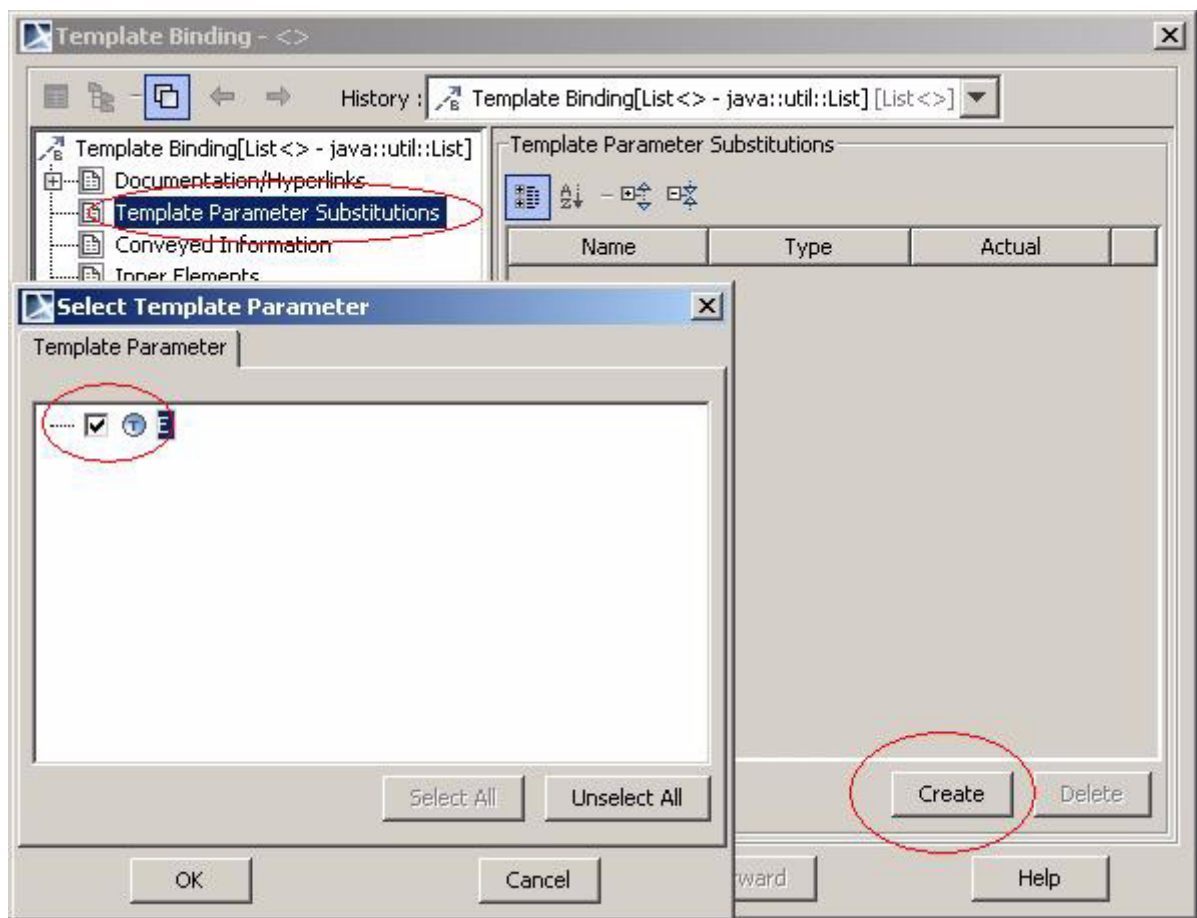


Figure 39 -- Creating UML Template Parameter Substitution

For created substitution element, you need to select actual value - at current situation it is UML class representing "java.lang.String".

Now we have created parameterized type, which can be used in the model to represent type java.util.List<String>

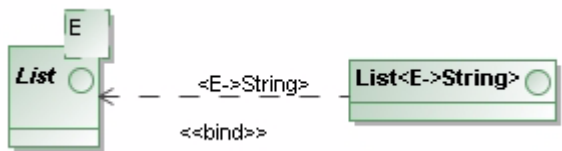


Figure 40 -- Created parameterized type for "java.util.List<String>"

## Imports

parameters

Java element	MagicDraw-UML element
--------------	-----------------------

Type import	Mapped to the UML Element Import with stereotype <<JavaImport>> (optional). Supplier is imported UML Classifier, which represents imported Java type, and client is UML Classifier which requires imported element.
Package import	Mapped to the UML Package Import with stereotype <<JavaImport>> (optional). Supplier is imported UML Package, which represents imported Java package, and client is UML Classifier which requires imported elements.
Static import for all static members	Mapped to the UML Element Import with stereotype <<JavaStaticImport>>. {JavaImportAll} tag of <<JavaStaticImport>> must have "true" value. Supplier is imported UML Classifier, owner of static members, which are imported, and client is UML Classifier which requires imported elements.
Static import for single static members	Mapped to the UML Element Import with stereotype <<JavaStaticImport>>. {JavaImportAll} tag of <<JavaStaticImport>> must have "false" value and tag {JavaImportedMember} must have reference to the imported member. Supplier is UML Classifier, owner of static member, which is imported, and client is UML Classifier which requires imported elements.  If there are several static members imported, {JavaImportedMember} can have listed all of them.

### Example

### Java Source Code

```
import java.util.List;  
import java.util.*;  
import static java.lang.Math.*;
```

```
public class Test  
{  
}
```

### MagicDraw UML Model

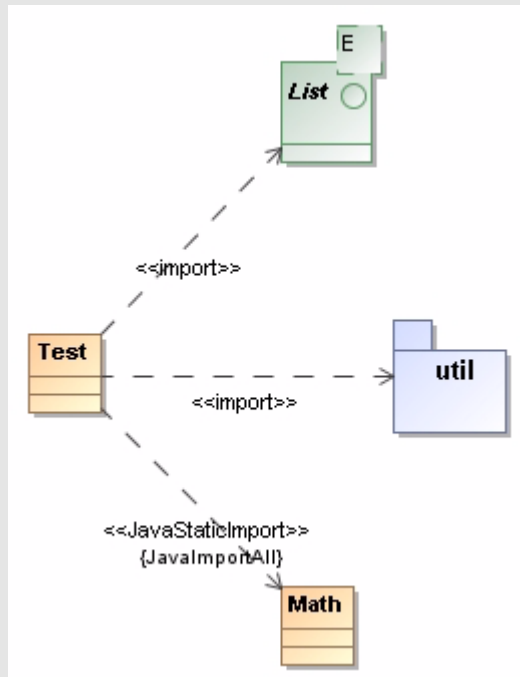


Figure 41 -- Java imports in diagram

## Java CE Properties

### Java Reverse Properties

These options are visible every time you attempt to reverse source code. Here we will describe marked options in Figure 42 on page 56. Other options are common for all languages and are described in "Reverse", on page 13.

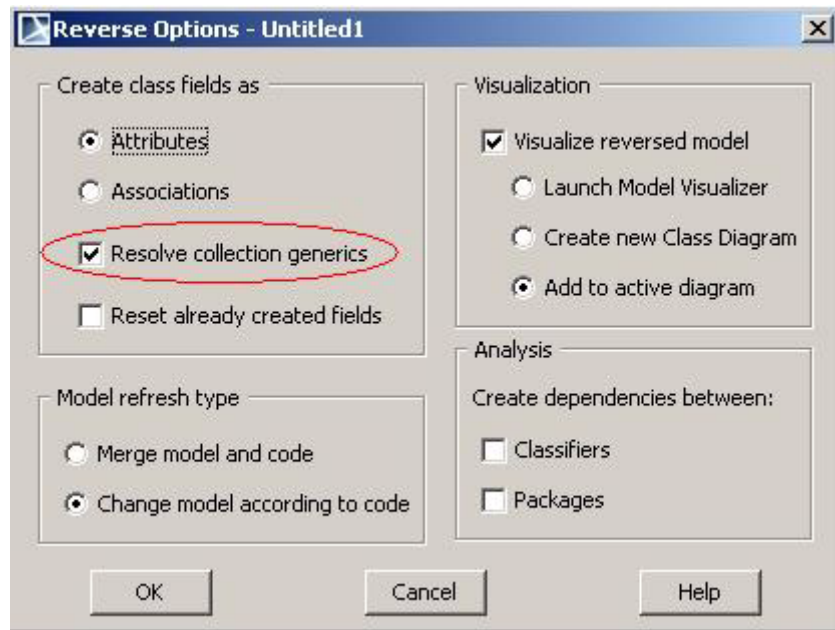


Figure 42 -- Java reverse properties

### Resolve Collection Generics Option

By default this option is turned on. Since the JLS 3, in Java was introduced parameterized types and to all Java collections were added type variables. Now, on reverse engineering now it is possible to find out what type is in Java collection and make association directly to the contained type instead of the Java collection.

On reverse engineering, it finds out Java parameterized collection and retrieves Java type which is used in container. This type is set as a "Type" to the UML Property. Container type is set to the UML Property Java language property "Container" as simple string.

#### Example

**Source Code Sample**

```
public class Test
{
    java.util.List<String> attribute;
}
```

**MagicDraw UML Model**

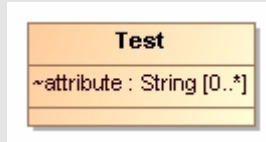


Figure 43 -- Attribute type, retrieved from collection

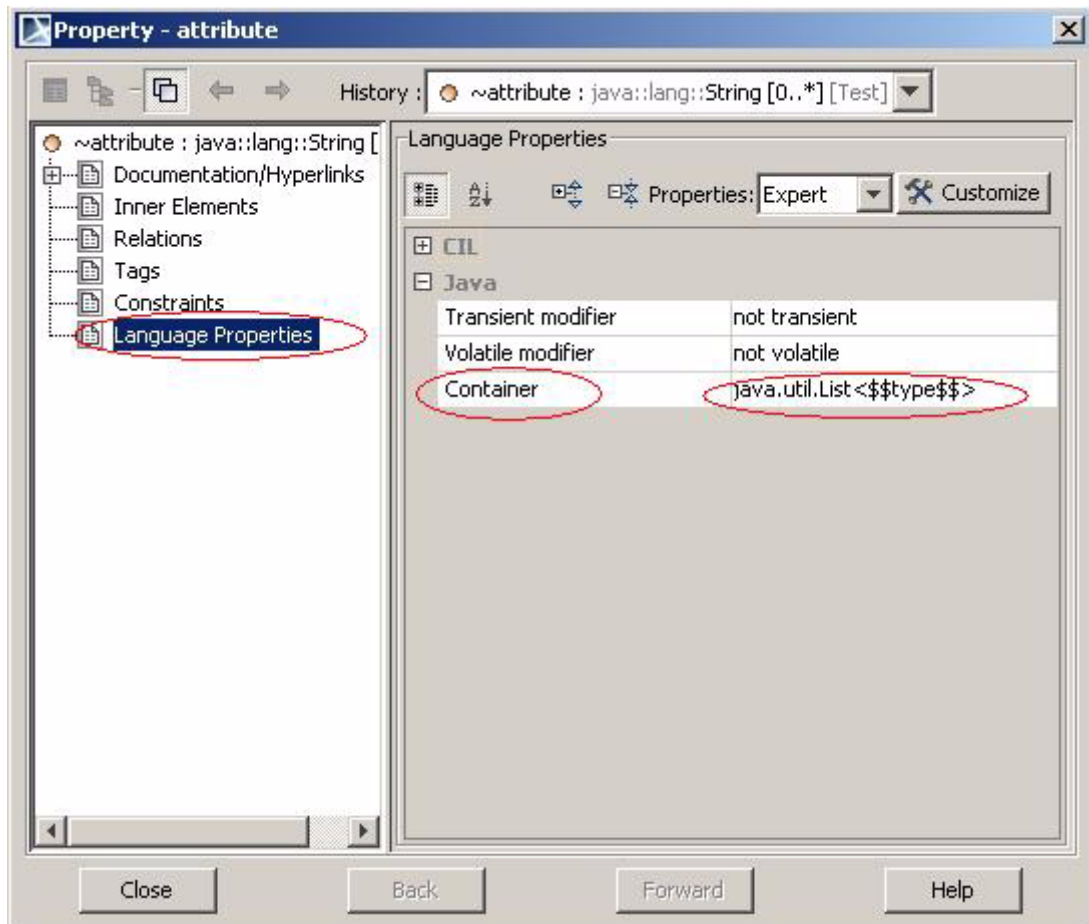


Figure 44 -- Collection type in UML Property specification

Note, that \$\$type\$\$ shows where should be in lined UML Property type on code generation.

## Java Language Options

You can find these options at the Options-> Project-> Code Engineering-> Java Language Options.

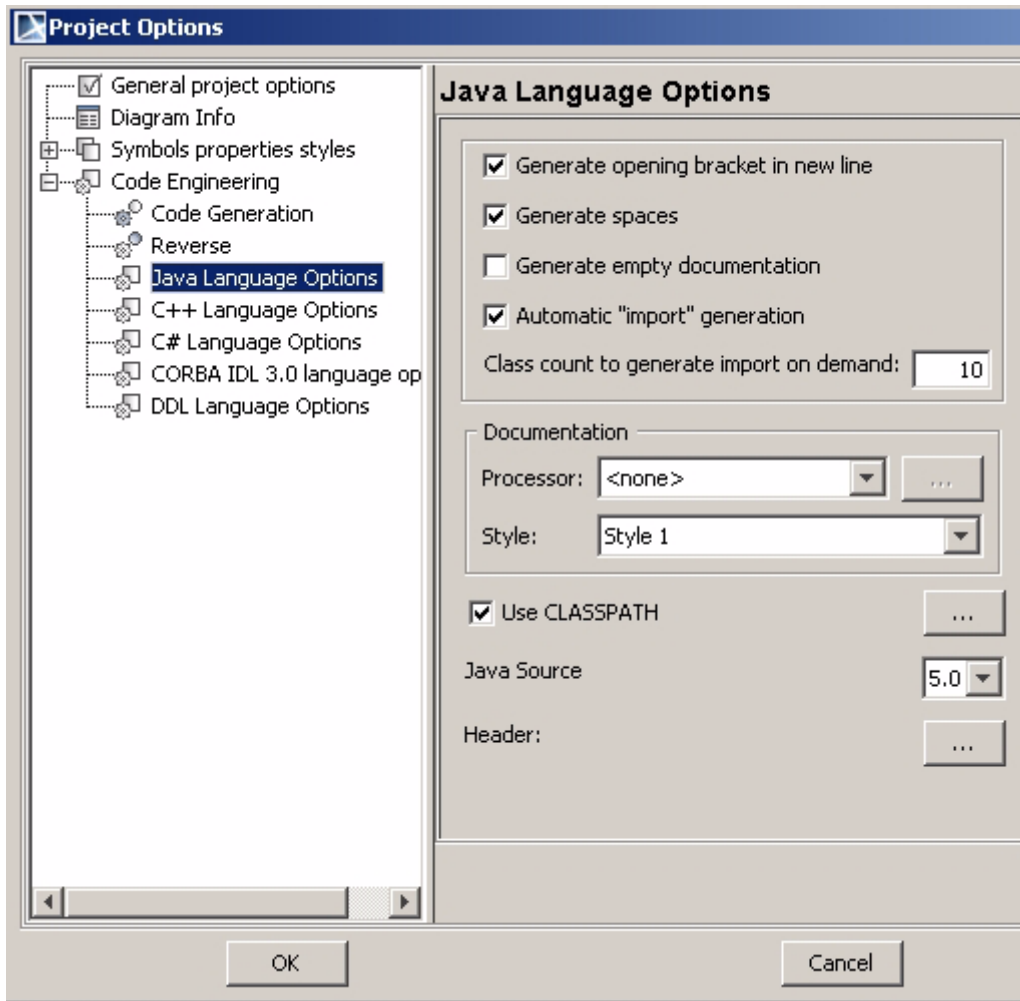


Figure 45 -- Java language options

## Generate Opening Bracket In New Line

By default is turned on. If element (Java type or operation) is generated for the first time into source, curly bracket is generated from the new line, if options is on, or in the same line as declaration header ends, if option is off.

## Generate Spaces

By default it is turned on. If option is on - adds additional space after open bracket and before close bracket in parameter declaration list.

## Generate Empty Documentation

By default is turned off. If option is turned on, MagicDraw is generating documentation to the source even, if there are no documentation in model. Just adds Java documentation start and ending symbols.

Here is generated class header with options turned on. UML Class A doesn't have documentation, but still in code is added documentation elements.

```
/**
 *
 */
class A
```

## Automatic “import“ Generation

By default is turned on. If option is turned on, all required imports are added automatically by used references on code generation. If this option is off, than imports are managed by user using specific mapping (see “Imports” on page 53).

Note, that if option is turned off, than no imports are created automatically and all imports, retrieved from source code on reverse engineering, are mapped into UML relationships as described in “Imports” on page 53.

## Class Count To Generate Import On Demand

By default value is 10. It means, that if there will be 9 references from the same package to different types, then imports will be generated explicitly to these classes, but if there will be 10 and more references, than it will be generated one import to the package.

This option is valid only, when “Automatic import generation“ is turned on.

## Documentation

Java documentation has two options by itself. It has **Processor** and **Style**. **Style** is used to define how to format documentation by adding some comments. There are predefined two styles:

Style 1

```
/**
 *
 */
```

Style 2

```
/**
 *
 */
```

Processor is responsible for analyzing documentation context. There are two types <none> and Java Doc.

<none> options does nothing with documentation and just set it as is on element.

Java Doc is processing documentation by resolving parameter tags or on code generation building documentation by collecting comments from the UML Parameters and adding missed tags for thrown exceptions or return.

There are additional Java Doc options (button “...”). These options can be used to declare what tags would not be generated or what order to use on code generation or perhaps to add some additional tag to documentation for all elements.

Note, that **Java Doc** processor splits operation documentation for UML Parameter and UML Operation on source code reverse and on code generation UML Parameter documentation is used to build Java documentation for operation.

## Classpath

You can define classpath here by referencing **jar** files or **class** files directories.

This options is used by Java reverse engineering. If referenced element is not found in model for some reasons, that it is searched in this defined path. And if class is matched by name, this class is added into model.

By default, MagicDraw imports boot classpath of the JVM, on which is running.

### Java Source

By default is Java 5.0. There are options **1.4** and **5.0**. If you are reversing older specification source code, where, for example “**enum**” is not a keyword and can be a variable name, then you will need to choose **1.4** Java source, else MagicDraw parser can emit error.

### Header

It is a header for a newly generated Java files. There can be added some template string which will be preprocessed on writing to source code.

Template strings

\$FILE_NAME	File name, without a path
\$DATE	System date
\$TIME	System time
\$AUTHOR	User name on the system

## Method Implementation Reverse

Java reverse to Sequence diagram functionality allows visualizing Java method implementation with UML Sequence diagram. Created from method Sequence diagram cannot be updated, every time new diagram should be generated.

To launch **Sequence Diagram from Java Source Wizard** and specify options needed for the reverse

---

- You are able to reverse any operation from the Browser: right click an operation, choose **Reverse Implementation** and launch **Sequence diagram from Java Source Wizard**.
- From the **Tools** menu, choose **Model Visualizer**, and then choose **Sequence Diagram from Java Source Wizard**.
- When reversing, in the **Reverse Options** dialog box, choose Launch Model Visualizer and then choose **Sequence Diagram from Java Source Wizard**.

The more detailed example of how this functionality works, see MagicDraw Tutorials.pdf, which is locate in <MagicDraw installation directory>, manual folder.

### Sequence Diagram from Java Source Wizard

Sequence Diagram from Java Source Wizard is the primary tool for reversing s sequence diagram from Java method. It contains four steps that are described below.



### STEP 1 Specify Name and Package.

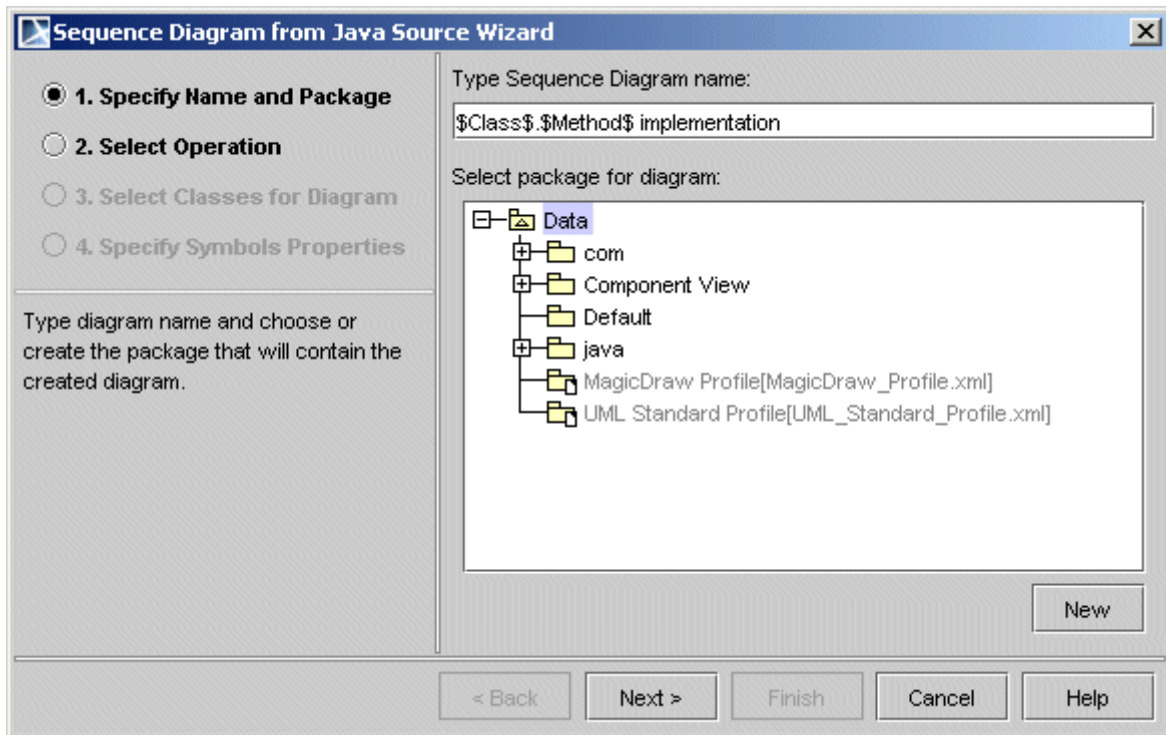
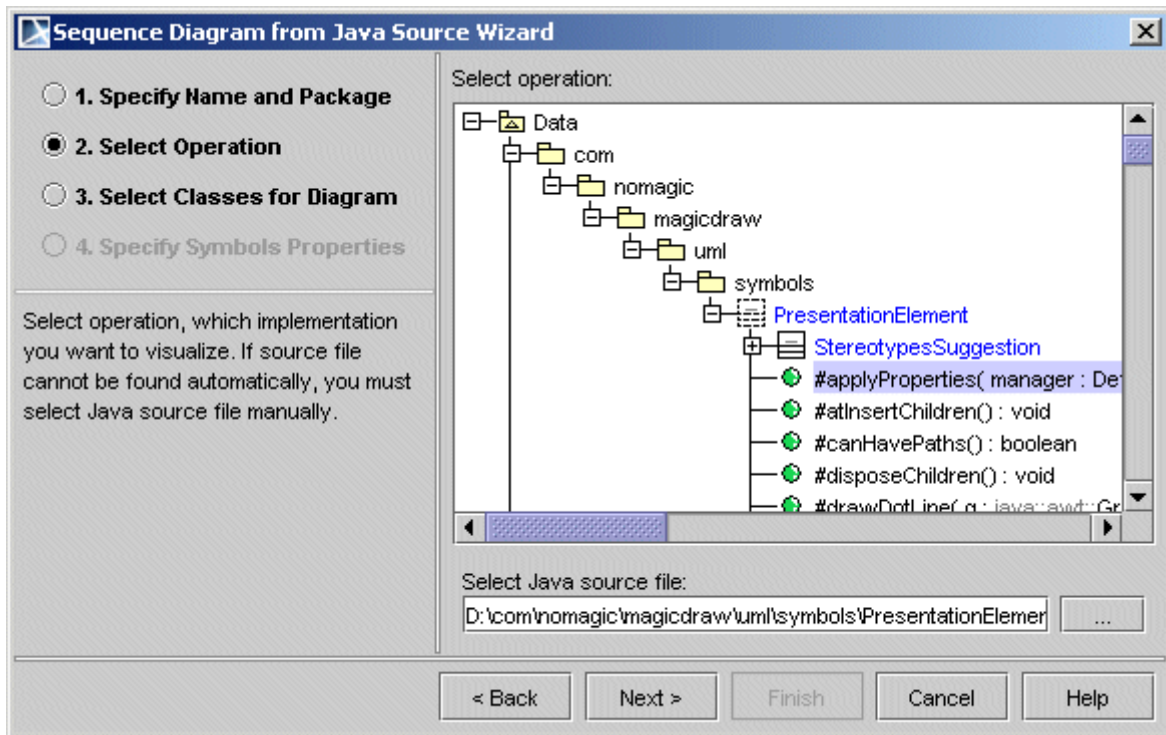


Figure 46 -- Sequence Diagram from Java Source Wizard

In this step, type the name of the newly created sequence diagram. Be default class name and selected operation name with a word “implementation” will be included in the sequence diagram name.

Also choose the package that will contain created sequence diagram. If you want to create a new package and place there a sequence diagram, click the **New** button and define package parameters in the **Package Specification** dialog box.

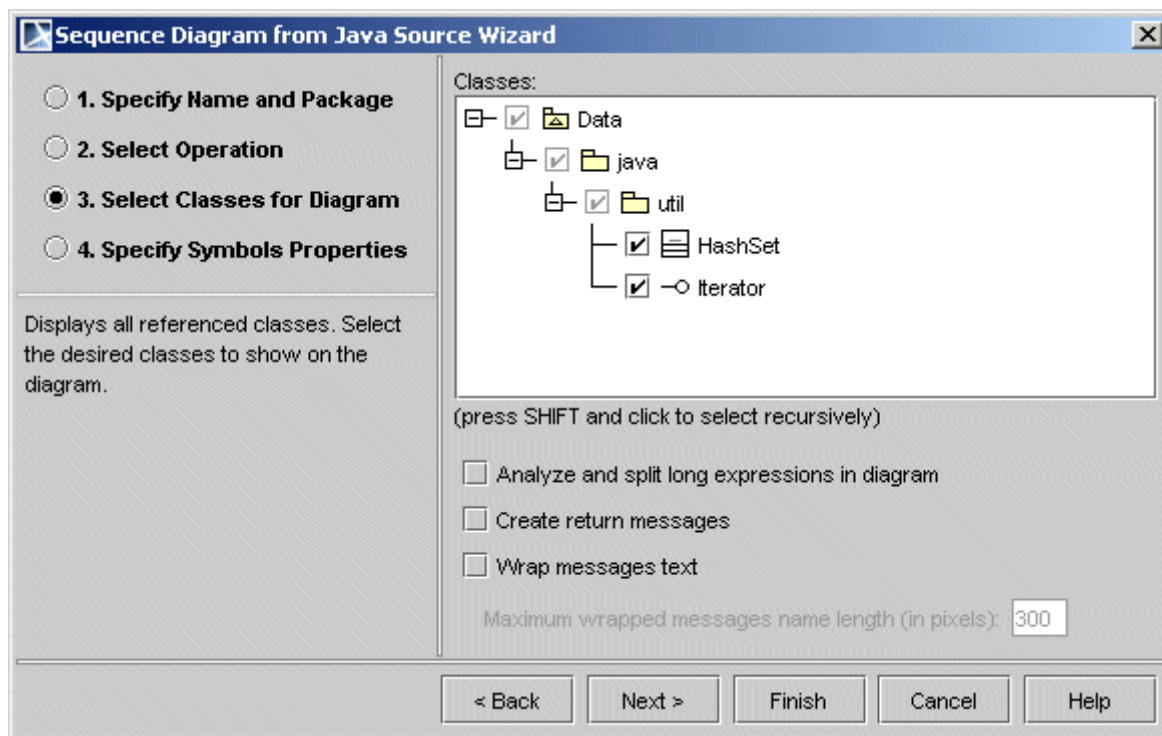
## STEP 2 Select Operation



In this step, select an operation for which you want to create a sequence diagram. If the Java source file is not shown you must select it manually.

**IMPORTANT** To specify implementation files, we suggest, before reversing, to specify Java **Default working directory** in the **Project Options** dialog box (specify root folder where all source files can be found).

### STEP 3 Select Classes for Diagram



In the Select Classes for Diagram step, all referenced classes are displayed. Select the desired classes and instances of those classes will be added into diagram with call messages to them.

- Select the **Analyze and split long expressions in diagram** check box if expression contains calls and cannot be displayed as call message. Then every call will be shown as separate call message with temporary variable initialization.
- Select the **Create return message** check box, if you want to display return message for every call message.
- Select the **Wrap message text** check box and specify the maximum message text length in pixels, to wrap longer message.

## STEP 4 Specify Symbols Properties

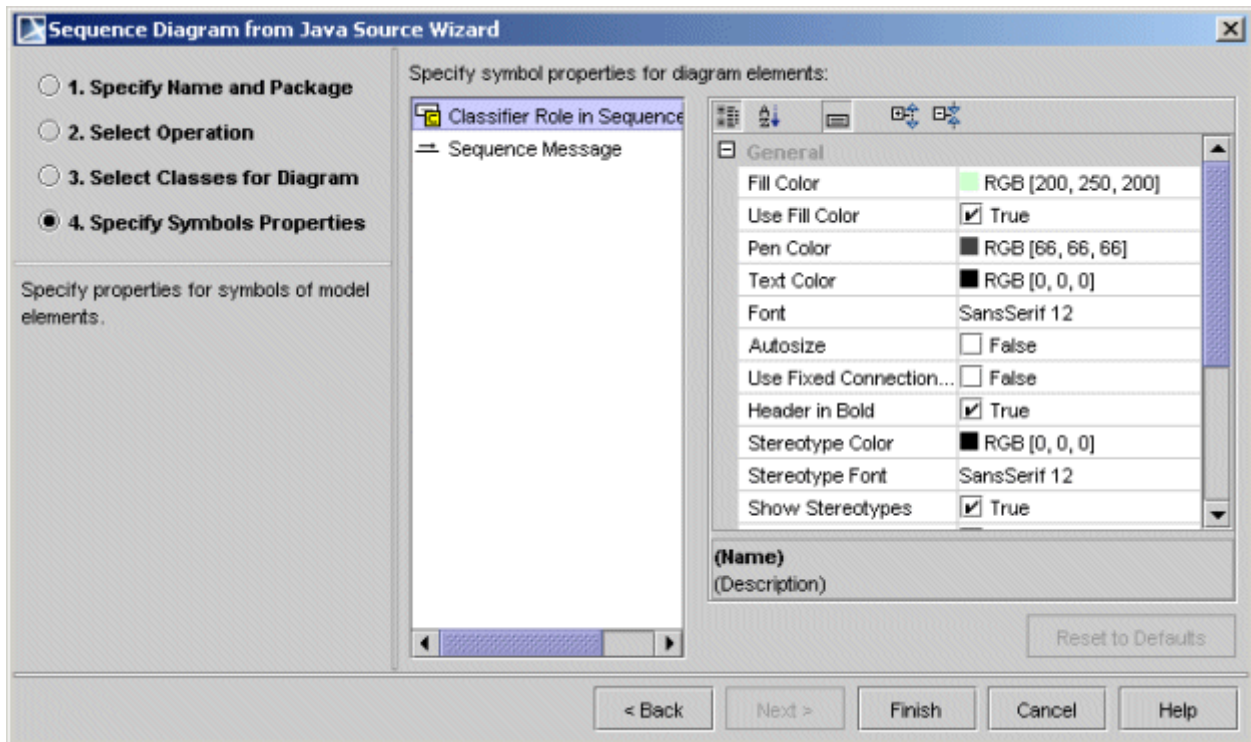


Figure 47 -- Sequence Diagram from Sequence Wizard. Specify Symbols Properties

In this step, define symbols properties for lifelines and messages.

# C++ CODE ENGINEERING

## Abbreviations

GUI	Graphical User Interface
CE	Code Engineering
CES	Code Engineering Set
AST	Abstract Syntax Tree
RT	Round-Trip forward and backward code engineering without code loss
CLR	Common Language Runtime
DSL	Domain Specific Language

## References

ISO/IEC 14882 C++ ANSI spec	ANSI_C++_Spec_2003.pdf
MSDN Library - Visual Studio 2005	ECMA-372 C++/CLI Language Specification

## C++ ANSI Profile

The ANSI C++ profile is the base for all other C++ profiles. Other specialized C++ profiles need to inherit from this profile (by creating a generalization in MD).

## Data Types

Fundamental types defined by ANSI (ANSI spec 3.9.1) are mapped to UML data type.

Each fundamental type with a modifier (signed, unsigned, short, long) can be declared in different order, but it is the same type. For example, short int or int short. Only versions defined as “type” by ANSI simple-type-specifiers are created as datatype in the profile. (See ANSI spec 7.1.5.2) If an UML synonym data type is found during the reverse process, then a class is created with this name.

Each fundamental type has 3 corresponding cv-qualified versions: const, volatile, and const volatile (or volatile const). See *Const Volatile qualified type* for mapping.

UML data types defined in the MD UML profile contain char, int, double, float, or void. We use these datatypes for mapping C++ type.

These datatypes are MD specific. In UML 2 only Integer, Boolean, String, and UnlimitedNatural are defined and they are not defined as datatypes, but as primitives. In the current version of MD UnlimitedNatural is not defined in any profile and Integer and boolean (b is lowercase) are datatypes.

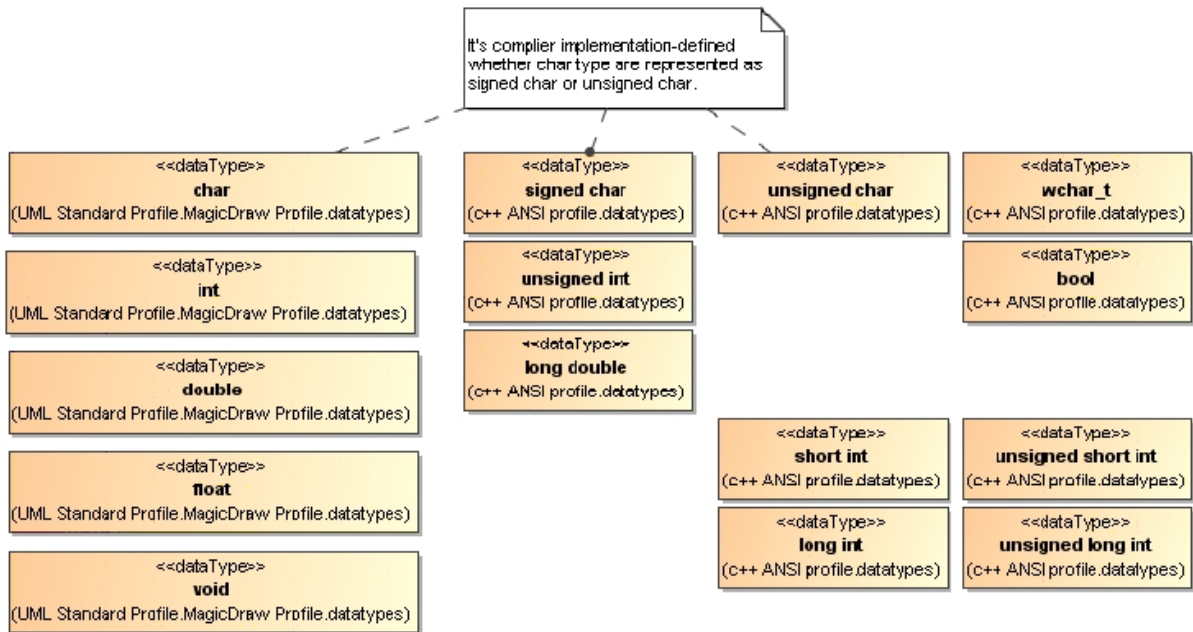


Figure 48 -- C++ datatype

## char

char types are represented as signed char and unsigned char according to the compiler implementation. char does not have a synonym.

## signed char

signed char does not have a synonym.

## unsigned char

unsigned char does not have a synonym.

## int

Synonyms for int are

- signed
- signed int
- int signed

## unsigned int

Synonyms for unsigned int are

- unsigned
- int unsigned

### **unsigned short int**

Synonyms for unsigned short int are

- unsigned short
- short unsigned
- unsigned int short
- short unsigned int
- short int unsigned
- int short unsigned
- int unsigned short

### **unsigned long int**

Synonyms for unsigned long int are

- unsigned long
- long unsigned
- unsigned int long
- long unsigned int
- long int unsigned
- int long unsigned
- int unsigned long

### **long int**

Synonyms for long int are

- long
- int long
- signed long
- long signed
- signed long int
- signed int long
- long signed int
- long int signed
- int signed long
- int long signed

### **short int**

Synonyms for short int are

- short
- int short
- signed short
- short signed
- signed short int

- signed int short
- short signed int
- short int signed
- int signed short
- int short signed

### **double**

double does not have asynonym.

### **long double**

long double does not have a synonym

### **float**

float does not have a synonym.

### **void**

void does not have a synonym.

### **bool**

bool does not have a synonym.

### **wchar\_t**

wchar\_t does not have asynonym.

## **Stereotypes**

All C++ stereotypes are based on <<C++Element>> stereotypes.

Constraints described in this chapter are for information only, syntax of these constraints need to be checked with the future OCL interpreter.

<<C++Class>>, <<C++Operation>>, <<C++Parameter>>, <<C++Attribute>>, <<C++LiteralValue>>, <<C++Include>>, <<C++Generalization>>, and <<C++TemplateParameter>> are invisible stereotypes and are used only to store C++ language properties. These stereotypes and their tag definitions are used by the DSL framework.



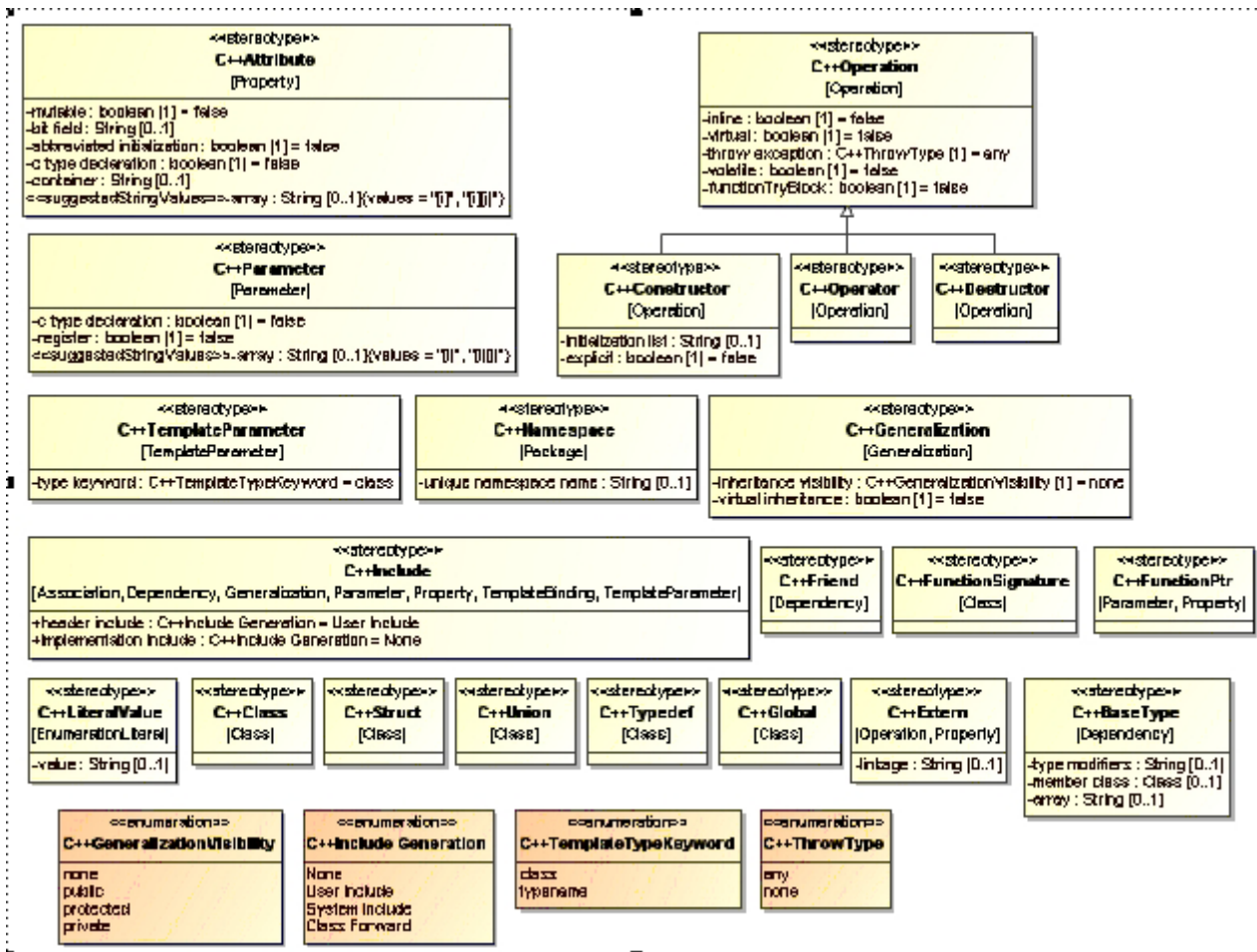


Figure 49 -- C++ Stereotype tag definitions

### C++Operation

<<C++Operation>> is an invisible stereotype used to include language properties for any C++ operation.

Name	Meta class	Constraints
C++Operation	Operation	Const function <pre>void f() const;</pre> Constraint: Only valid for member function <pre>if isQuery then   stereotype- &gt;select (name='C++Global') -&gt;isEmpty()</pre>
Tag	Type	Description
inline	boolean[1]=false	Inline function <pre>inline a();</pre>
throw exception	C++ThrowType[1] =any	Exception specification. operation.raisedExpression is not empty, the throw expression is generated. <pre>void f() throw(int);</pre> If operation.raisedExpression is empty and throw expression is none, a throw expression without argument is generated. <pre>void f() throw ();</pre> If operation.raisedExpression is empty and throw expression is any, does not generate a throw keyword. <pre>void f();</pre>
virtual	boolean[1]=false	Virtual function <pre>virtual a();</pre> Constraint: Only valid for member function and non static <pre>stereotype- &gt;select (name='C++Global')   -&gt;isEmpty() and IsStatic = false</pre>
volatile	boolean[1]=false	Volatile function <pre>void f() volatile;</pre>
funtionTryBlock	boolean[1]=false	Function try block <pre>void f() try {}</pre>

### C++Operator

<<C++Operator>> stereotype is used to define a C++ operator function. This stereotype extends the <<C++Operation>> stereotype. See **Function operator** for more information.

Name	Meta class	Constraints
C++Operator	Operation	operator function <pre>T&amp; operator+(T&amp; a);</pre> Constraint: name start with "operator"

**C++Parameter**

<<C++Parameter>> is an invisible stereotype used to include language properties for any C++ function parameter.

name	Meta class	Constraints
C++Parameter	Parameter	
Tag	Type	Description
c type declaration	boolean[1]=false	Declare parameter's type in C style C style: void a (enum Day x); C++ style: void a (Day x);
register	boolean[1]=false	Register parameter void a (register int x);
array	String[0..1]	C++ array definition void a (int x [2] [2]);

### C++Attribute

<<C++Attribute>> is an invisible stereotype used to include language properties for any C++ variable.

name	Meta class	Constraints
C++Attribute	Property	Constraint for code generation. It is valid to have a default value for any kind of attribute, but it is illegal to initialize a member variable within its definition. (Eg. <code>class A { int x = 1; };</code> ) if <code>defaultValue.size() &gt; 0</code> then owner.stereotype->exists(name='C++Global') or (isStatic = true and typeModifiers.contains("const"))
Tag	Type	Description
abbreviated initialization	boolean[1]=false	Initialize the attribute with the abbreviate form. <code>int x(5);</code> Constraint owner.stereotype->exists(name='C++Global')
bit field	String[0..1]	Bit field declaration <code>int x:2;</code> Constraint: Only valid for member function stereotype->select(name='C++Global')->isEmpty()
c type declaration	boolean[1]=false	Declare attribute's type in C style C style: <code>enum Day x;</code> C++ style: <code>Day x;</code>
container	String[0..1]	container of the attribute. \$ character is replaced by the attribute type. <code>vector&lt;\$&gt; x;</code>
mutable	boolean[1]=false	Attribute mutable modifier. <code>mutable int x;</code> Constraint: Only valid for member function stereotype->select(name='C++Global')->isEmpty()
array	String[0..1]	C++ array definition <code>int x[2][2];</code>

### C++LiteralValue

<<C++LiteralValue>> is an invisible stereotype used to include language properties for any C++ enum field. See [Enumeration](#) for more info.

name	Meta class	Constraints
C++LiteralValue	EnumerationLiteral	
Tag	Type	Description
value	String[0..1]	Value definition of an enum field. (A valid C++ expression) <code>enum Day {Mon = 2};</code>

### C++Friend

<<C++Friend>> stereotype is used to define C++ friend relationship. See [Friend declaration](#) for more info.

Name	Meta class	Constraints
C++Friend	Dependency	Client is Class or Operation and supplier is Class ( <code>client.oclIsTypeOf(Class)</code> or <code>client.oclIsTypeOf(Operation)</code> ) and <code>supplier.oclIsTypeOf(Class)</code>

### C++Struct

<<C++Struct>> stereotype is used to define C++ struct. See [Struct](#) for more info.

name	Meta class	Constraints
C++Struct	Class	

### C++Typedef

<<C++Typedef>> stereotype is used to define C++ typedef. See [Typedef](#) for more info.

name	Meta class	Constraints
C++Typedef	Class	A typedef does not contain operation and attribute <code>feature-&gt;isEmpty()</code> A <<C++BaseType>> dependency is defined

### C++Union

<<C++Union>> stereotype is used to define C++ union. See [Union](#) for more info.

name	Meta class	Constraints
C++Union	Class	

### C++Global

<<C++Global>> stereotype is used to define global functions and variables. (functions and variables outside a class/struct/union declaration). See ["Global functions and variables", on page 94](#) for more info.

name	Meta class	Constraints
C++Global	Class	Only 1 <<C++Global>> class into a package <code>owner.ownedElement-&gt;select (</code> <code>    stereotype-&gt;select (name='C++Global')) .size()=1</code> All operations and attributes are public <code>feature-&gt;forall(visibility = #public)</code>

### C++Namespace

<<C++Namespace>> stereotype is used to define C++ namespace. See [Namespace](#) for more info.

name	Meta class	Constraints

C++Namespace	Package	
<b>Tag</b>	<b>Type</b>	<b>Description</b>
unique namespace name	String[0..1]	Unnamed namespace namespace { }

### C++Constructor

<<C++Constructor>> stereotype is used to define C++ constructor. This stereotype extends <<C++Operation>> stereotype. See [Class Constructor/Destructor](#) for more info.

<b>Name</b>	<b>Meta class</b>	<b>Constraints</b>
C++Constructor	Operation	name = owner.name
<b>Tag</b>	<b>Type</b>	<b>Description</b>
explicit	boolean[1]=false	Explicit constructor explicit a();
initialization list	String[0..1]	Constructor initialization a() : x(1) { }

### C++Destructor

<<C++Destructor>> stereotype is used to define C++ destructor. This stereotype extends <<C++Operation>> stereotype. See [Class Constructor/Destructor](#) for more info.

<b>Name</b>	<b>Meta class</b>	<b>Constraints</b>
C++Destructor	Operation	name = "~"+owner.name

### C++Extern

<<C++Extern>> stereotype is used to define C++ extern variable. See [Variable extern](#) for more info.

<b>Name</b>	<b>Meta class</b>	<b>Constraints</b>
C++Extern	Operation, Property	owner.stereotype->exists(name='C++Global')
<b>Tag</b>	<b>Type</b>	<b>Description</b>
linkage	String[0..1]	Linkage specification extern "C"

### C++FunctionPtr

<<C++FunctionPtr>> stereotype is used to define C++ function pointer. See [Function pointer](#) for more info.

<b>Name</b>	<b>Meta class</b>	<b>Constraints</b>
C++FunctionPtr	Parameter, Property	
<b>Tag</b>	<b>Type</b>	<b>Description</b>
signature	Operation	The signature of the function (C++ function pointer definition without the operation name)

member class	Class	The class used for pointer to member function.
--------------	-------	--

### C++FunctionSignature

<<C++FunctionSignature>> stereotype is used as a container to model C++ function pointer. See [Function pointer](#) for more info.

Name	Meta class	Constraints
C++FunctionSignature	Class	The class cannot have properties. properties->isEmpty()

### C++Class

<<C++Class>> stereotype is an invisible stereotype used to include language properties for any C++ variable.

Name	Meta class	Constraints
C++Class	Class	

### C++BaseType

<<C++BaseType>> stereotype is used to link base type of a typedef.

Name	Meta class	Constraints
C++BaseType	Dependency	Client is type of Class with <<C++Typedef>> stereotype.
Tag	Type	Description
type modifiers	String[0..1]	Type modifiers of the typedef or function pointer.
member class	Class[0..1]	Memberclass of typedef or function pointer.
array	String[0..1]	Array definition.

### C++Include

<<C++Include>> stereotype is used to keep the information about include type used for generating include and forward class declaration.

Name	Meta class	Constraints
C++Include	Association, Dependency, Generalization, Parameter, Property, TemplateBinding, TemplateParameter	Client is type of Component
Tag	Type	Description
header include	String	The value of tag is one of the following <ul style="list-style-type: none"> <li>• None</li> <li>• User Include</li> <li>• System Include</li> <li>• Class Forward</li> </ul>
implementation include	String	The value of tag is one of the following <ul style="list-style-type: none"> <li>• None</li> <li>• User Include</li> <li>• System Include</li> <li>• Class Forward</li> </ul>

The *header include* tag is used when the client component has header file extension, “\*.h”, otherwise the *implementation include* tag will be used.

The *User Include* tag value is used for generating user include, such as #include “test.h”.

The *System Include* tag value is used for generating system include, such as #include <string.h>

The *Class Forward* tag value is used for generating forward class declaration.

### C++TemplateParameter

<<C++Template Parameter>> is used to keep type keyword between class and typename for template parameter declaration.

Name	Meta class	Constraints
C++TemplateParameter	TemplateParameter	
Tag	Type	Description
type keyword	C++TemplateTypeKeyword=class	The value of tag is one of the following <ul style="list-style-type: none"> <li>• class</li> <li>• typename</li> </ul>

### C++Generalization

<<C++Generalization>> is used for information related to generalization and interface realization.

Name	Meta class	Constraints
------	------------	-------------




C++Generalization	Generalization, InterfaceRealization	
<b>Tag</b>	<b>Type</b>	<b>Description</b>
inheritance visibility	C++GeneralizationVisibility[1]= none	The value of tag is one of the following <ul style="list-style-type: none"> <li>• None</li> <li>• public</li> <li>• protected</li> <li>• private</li> </ul> <p>Example: class A : private B {};</p>
virtual inheritance	boolean[1]=false	Virtual inheritance Example: class A : virtual B {};

## Mapping

This chapter describes the mapping between C++ and UML.

### Class

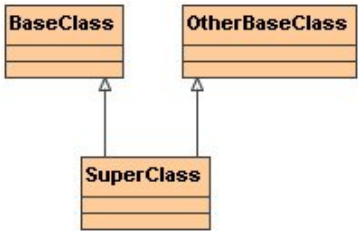
C++ class map to a UML class

Code	MD-UML
<pre>class A { };</pre>	

### Base Class Definition

Base class definition is mapped to UML generalization, a generalization is created between the base class and the super class.

Access visibility (public, protected and private) and virtual properties of the base class are mapped to C++ language properties of the UML generalization.

Code	MD-UML
<pre>class BaseClass {}; class OtherBaseClass {}; class SuperClass :     public BaseClass,     protected virtual OtherBaseClass { };</pre>	

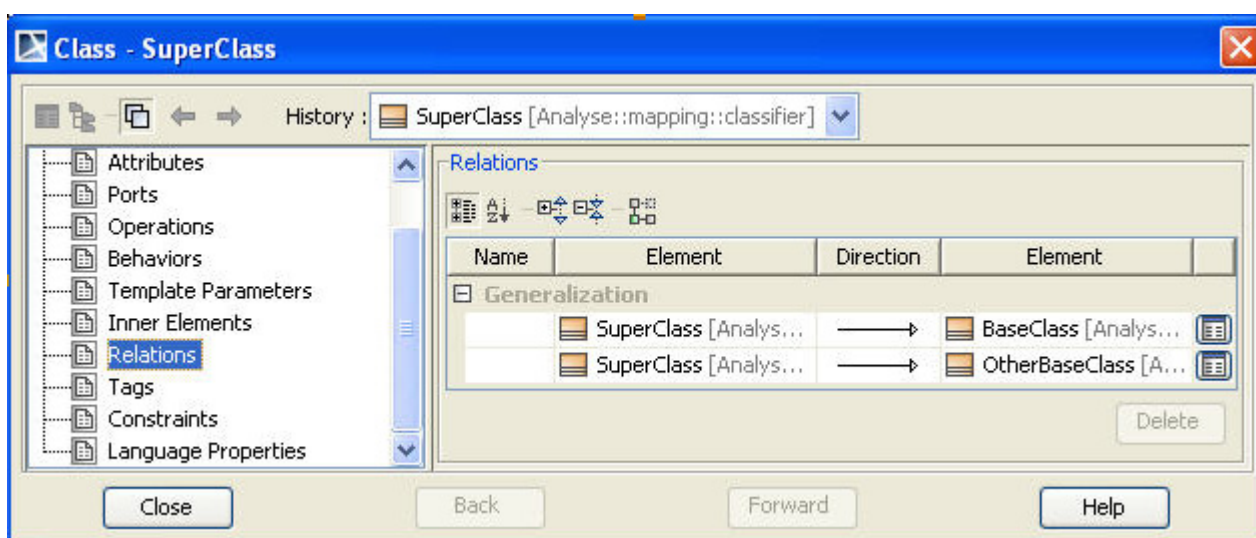


Figure 50 -- SuperClass Generalization relationship

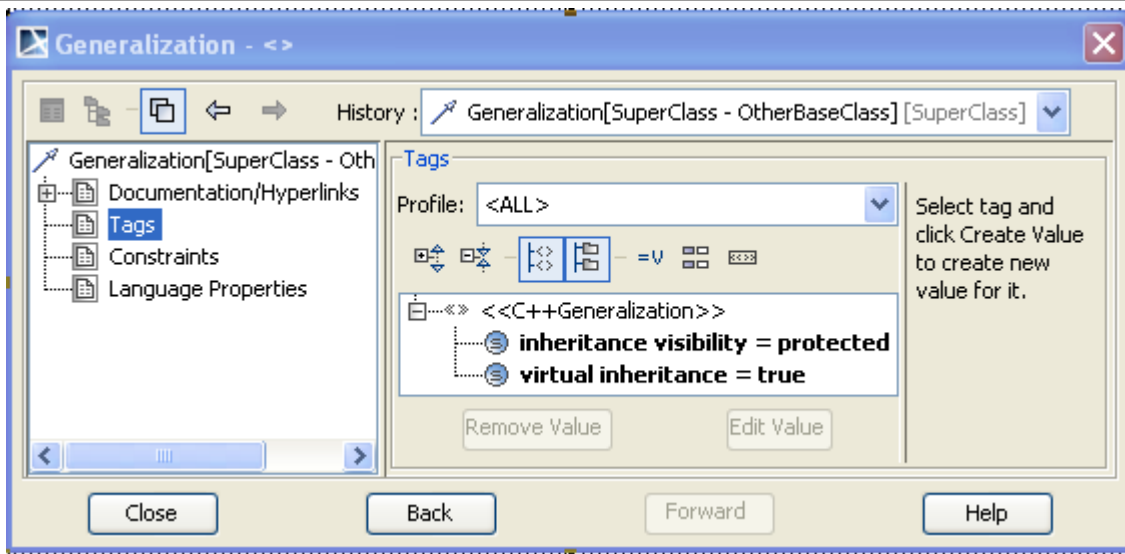


Figure 51 -- Generalization language properties

## Class Member Variable

Class member variables are mapped to UML attributes. See [Variable](#) for more info.

Code	MD-UML
<pre>class ClassVariable {     int age;     char* name; };</pre>	

## Class Member Function

Class member functions are mapped to UML operations. See [Function](#) for more info.

Code	MD-UML
<pre>class ClassFunction { public:     void simpleFunc();     float paramFunc(int x, char y); };</pre>	

## Class Constructor/Destructor

C++ class constructor and destructor are mapped to UML operation with <<C++Constructor>> stereotype and <<C++Destructor>> stereotype. See [C++Constructor](#) and [C++Destructor](#) for more info.

Code	MD-UML
------	--------

<pre>class ConstructClass { public:     ConstructClass();     ~ConstructClass(); }</pre>	 <p><b>ConstructClass</b></p> <pre>&lt;&lt;C++Constructor&gt;&gt;+ConstructClass() &lt;&lt;C++Destructor&gt;&gt;~ConstructClass()</pre>
--	---

## Variable

C++ variable is mapped to UML attribute, the variable type is mapped to the attribute type.

Code	MD-UML
int age;	

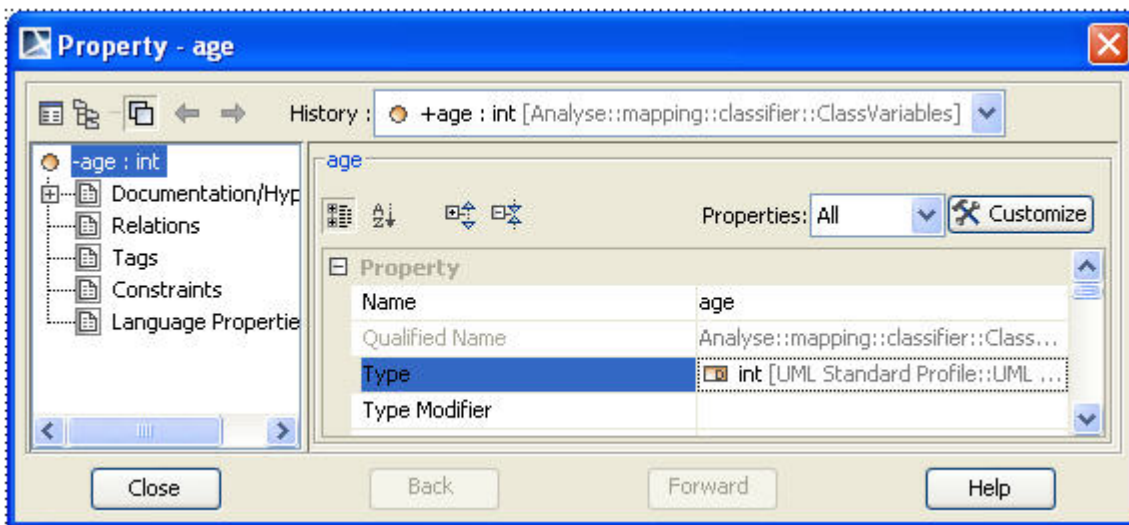


Figure 52 -- Variable property age

C++ type pointer/reference is mapped to Type Modifier property of the attribute. \$ character is replaced by the type name.

Code	MD-UML
int* ptrVar; int& refVar	

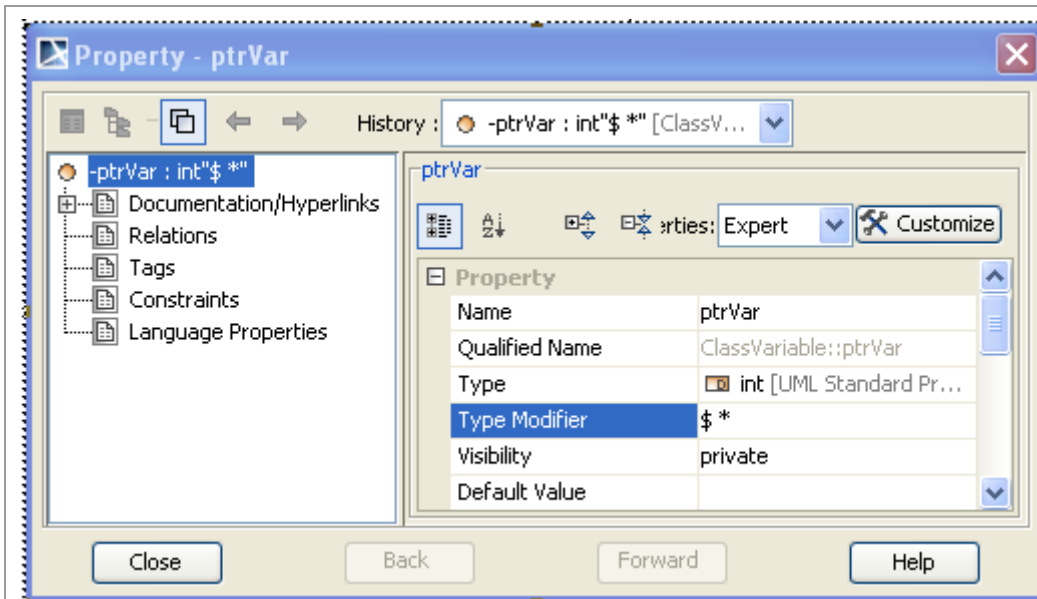


Figure 53 -- Variable property ptrVar

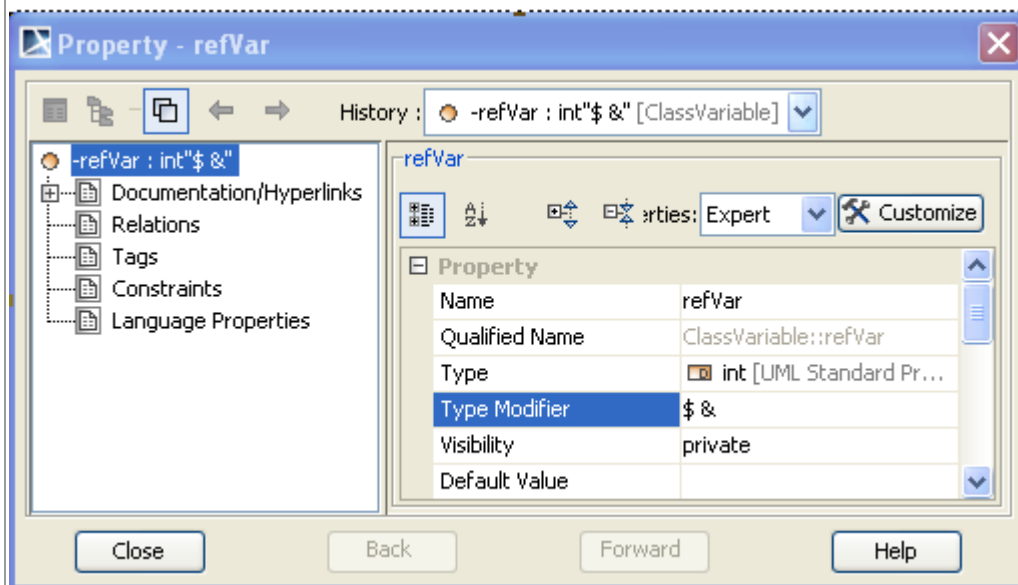
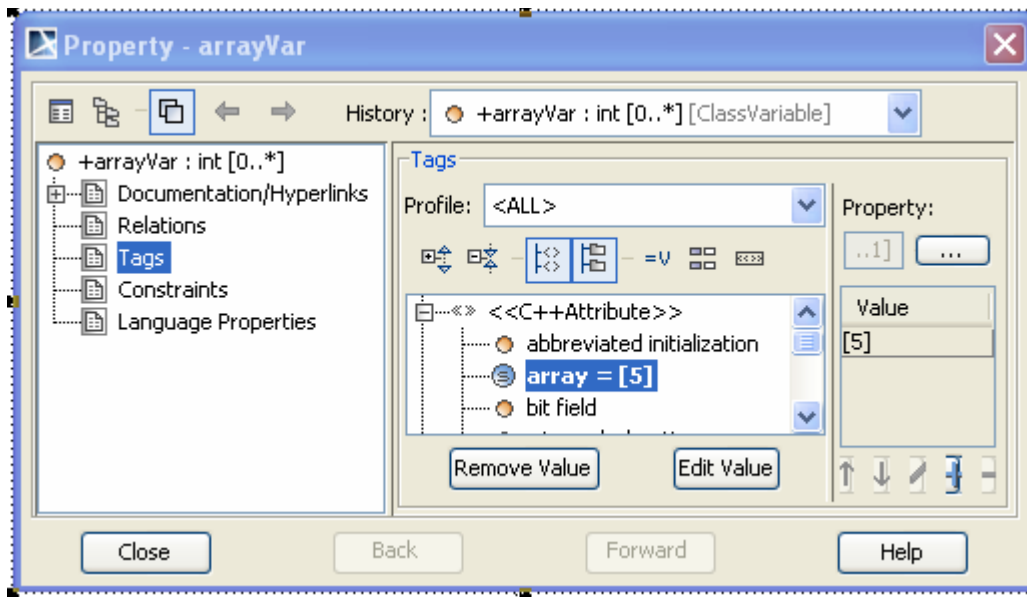


Figure 54 -- Variable property refVar

C++ array type is mapped to *array* tag value of the attribute. If *array* is set, then multiplicity property of UML attribute is set to "[0..\*]"

Code	MD-UML
<code>int arrayVar[5];</code>	

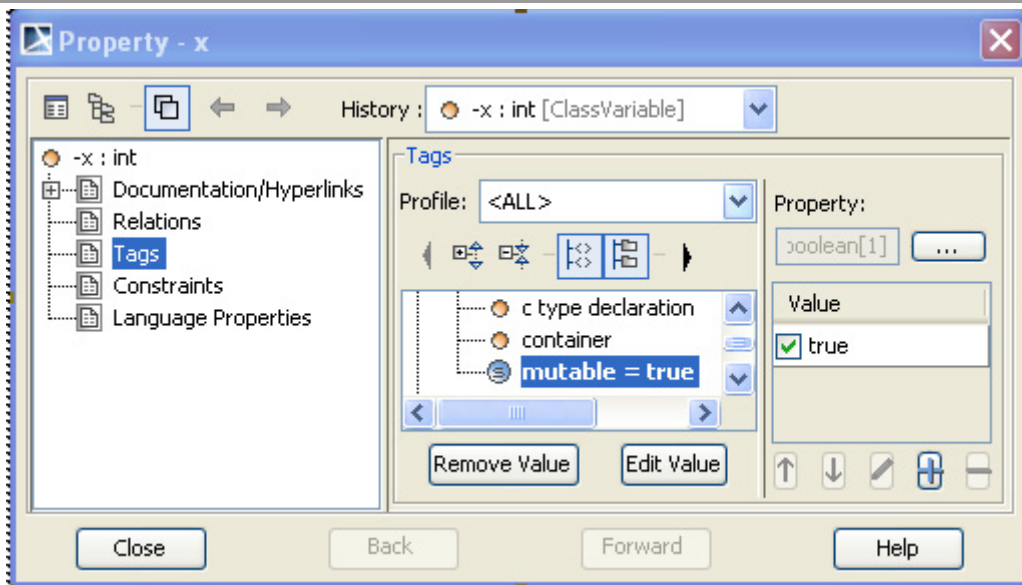


### Variable modifiers

Mutable variable modifiers are mapped to UML attribute's language properties *Mutable*.


Constraint: only member variable can be mutable (global variable can not).

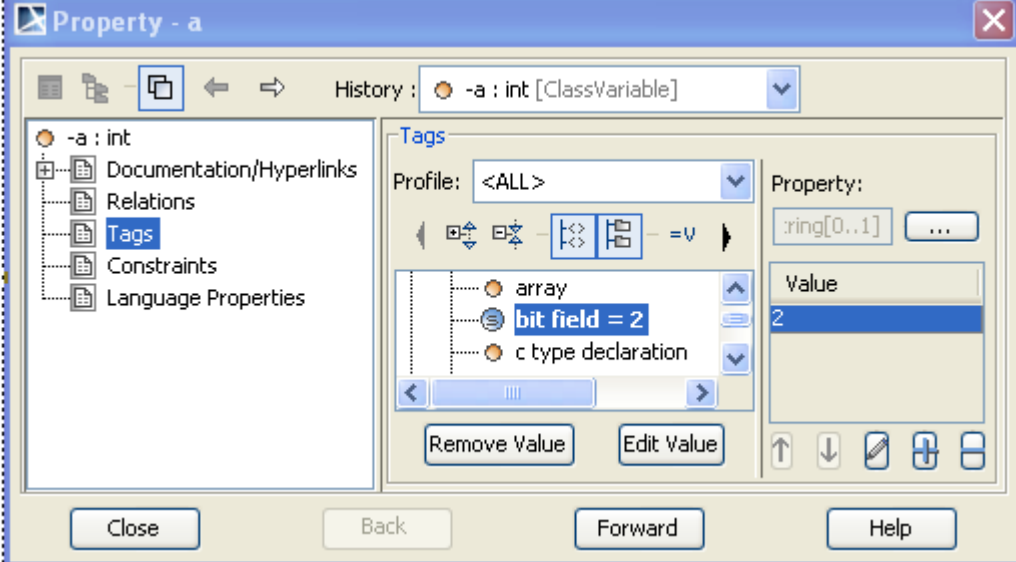
Code	MD-UML
<code>mutable int x;</code>	



Bit field is mapped to *Bit field* tag value.

Code	MD-UML

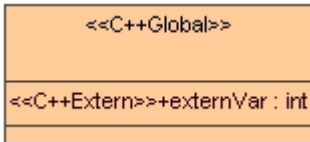
<pre>struct BitStruct {     int a:2; };</pre>	
---	--

### Variable extern

C++ extern variable is mapped to <<C++Extern>> stereotype. Linkage tag value is used to specify the kind of linkage “C” or “C++”, if linkage is empty (or without value) extern without linkage is generated.

See [C++Extern](#) for more info.

Code	MD-UML
<pre>extern int externVar;</pre>	

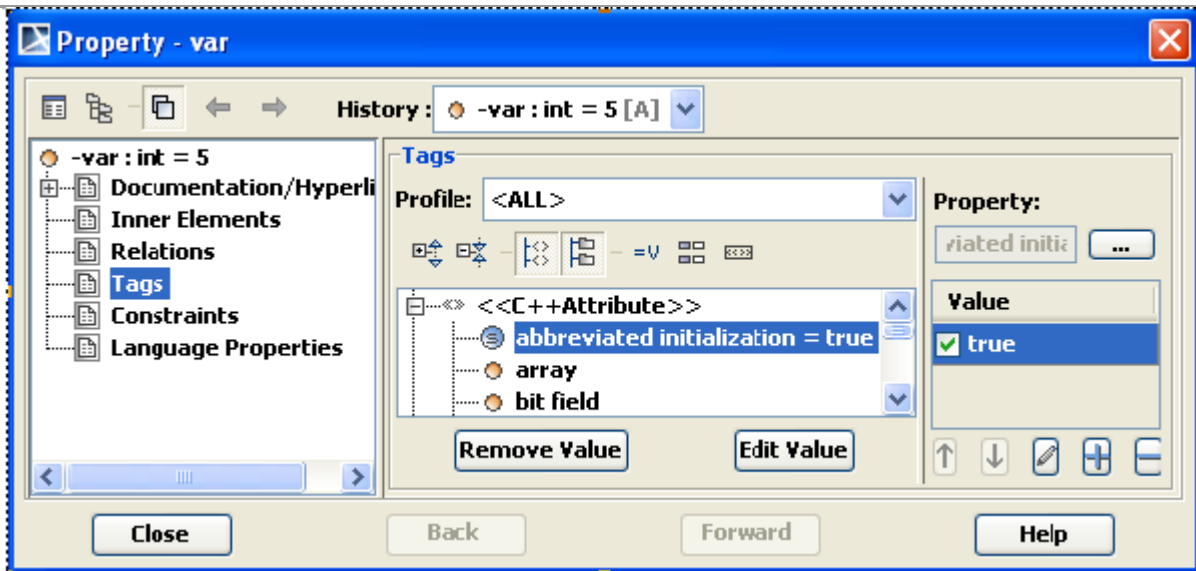
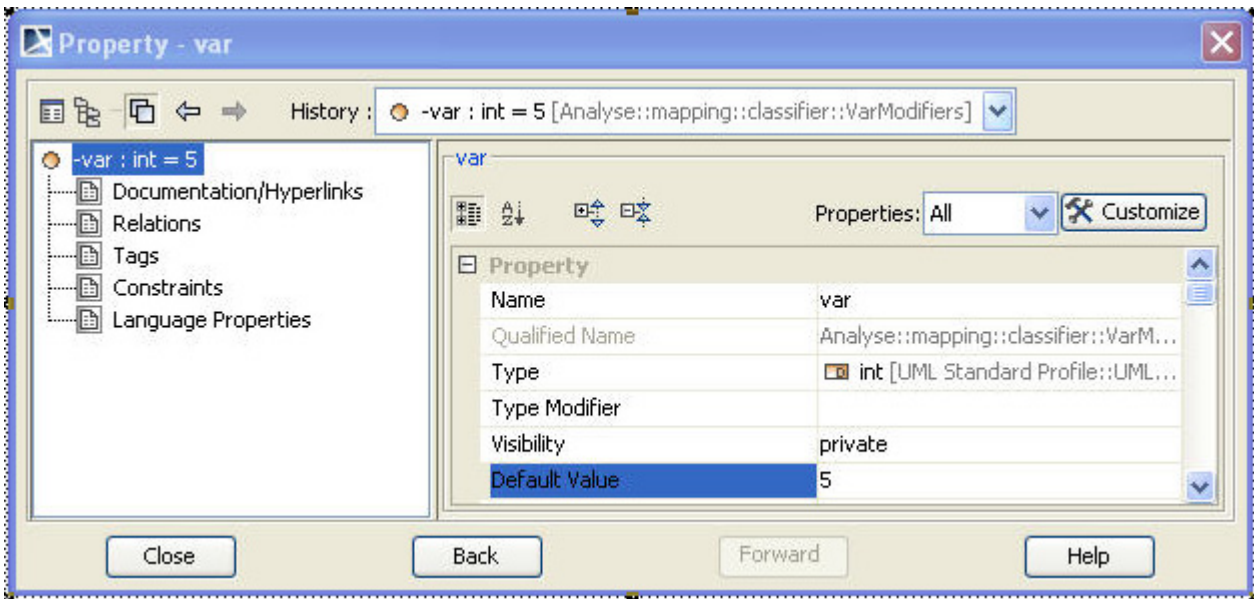
### Variable default value

Variable initial value is mapped to UML attribute’s default value.

Variable initial value set using function style method is mapped to UML attribute’s default value and attribute’s language property *Abbreviated Initialization* set to true.

Constraint: Only “static const” member variables can be initialized, and they can not be initialized using function style method.

Code	MD-UML
<pre>int var = 5; int var2(10);</pre>	



## Const Volatile qualified type

C++ const and volatile modifiers for attribute/function parameter are mapped to *Type Modifiers* properties .

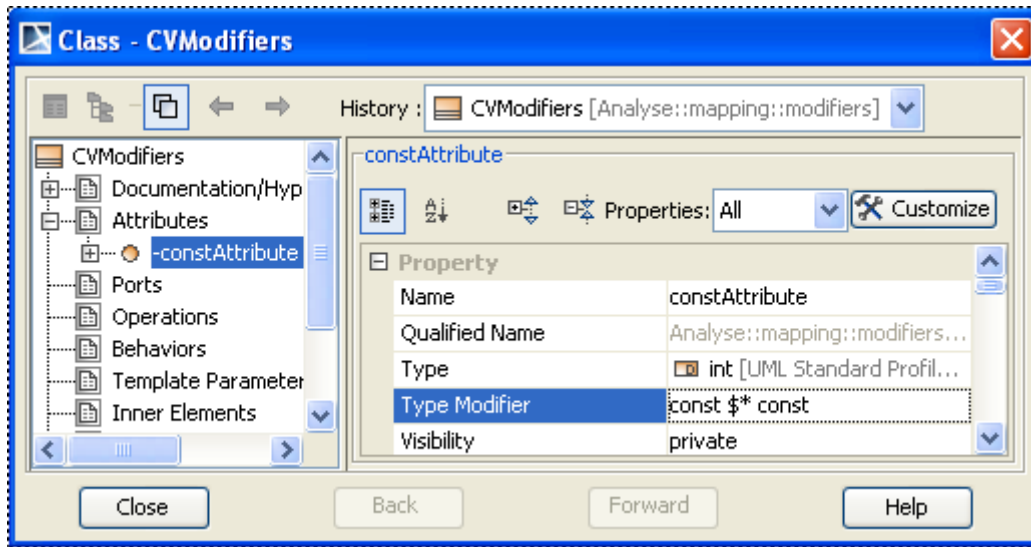
For const attribute, the property *Is Read Only* is set to true during reverse.

The character \$ into Type Modifier value is replaced by the type name.

Constraint : If the property *Is Read Only* is set and *Type Modifiers* is not set to const or const volatile (set to const, or an error message will display during syntax check)

Code	MD-UML
<pre>class CVModifiers {     const int* const constAttribute; }</pre>	<pre>classDiagram     class CVModifiers {         -constAttribute : int{readOnly}     }     </pre>





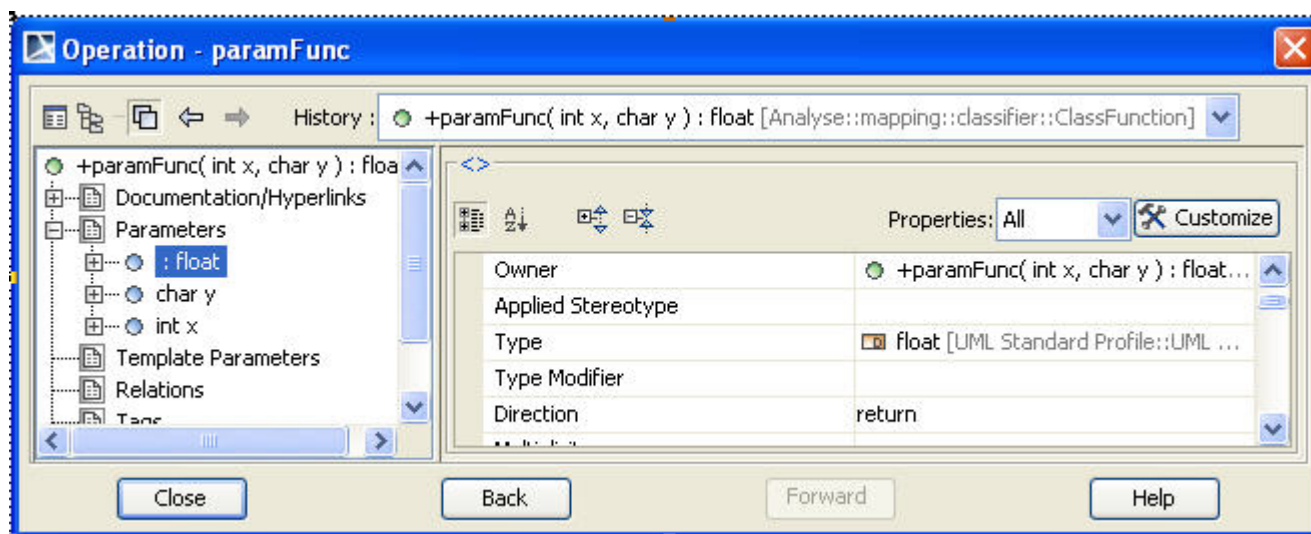
## Function

C++ function is mapped to UML operation, parameter of function is mapped to UML parameter with property *direction* set to "inout", return type of function is mapped to UML parameter with property *direction* set to "return". Type of parameter is mapped to type of UML parameter.

C++ default parameter value is mapped to *defaultValue* property of UML parameter.

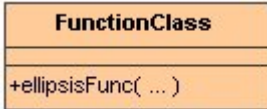
Pointer, reference and array type of parameter are mapped to property *Type Modifier* of parameter. See [Variable modifiers](#) for more info.

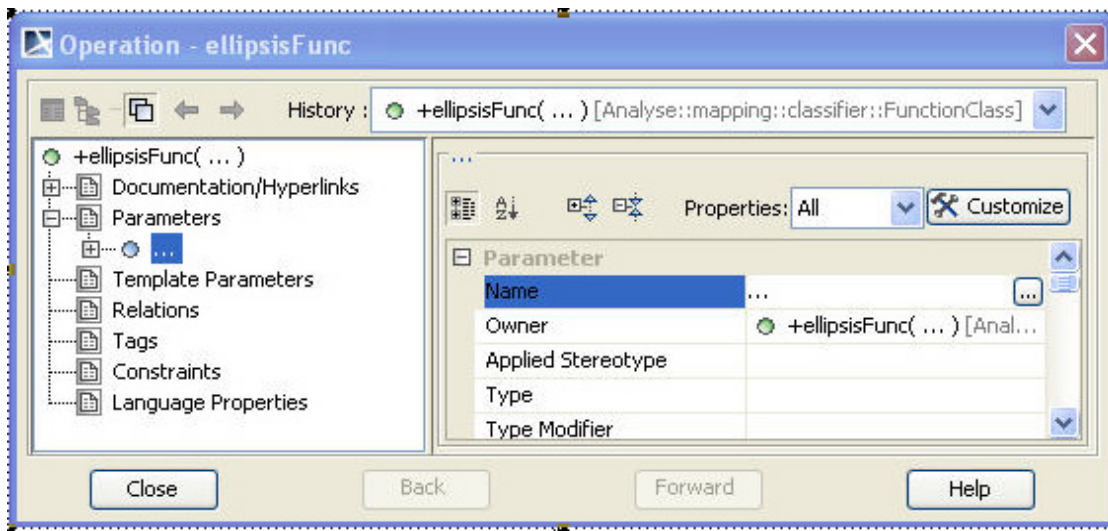
Code	MD-UML
<code>float paramFunc(int x, char x);</code>	



## Function variable-length parameter list

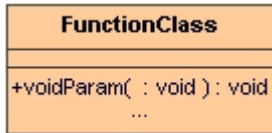
C++ function variable-length parameter list is mapped to a UML parameter with name “...” (dot 3 times) and without type.

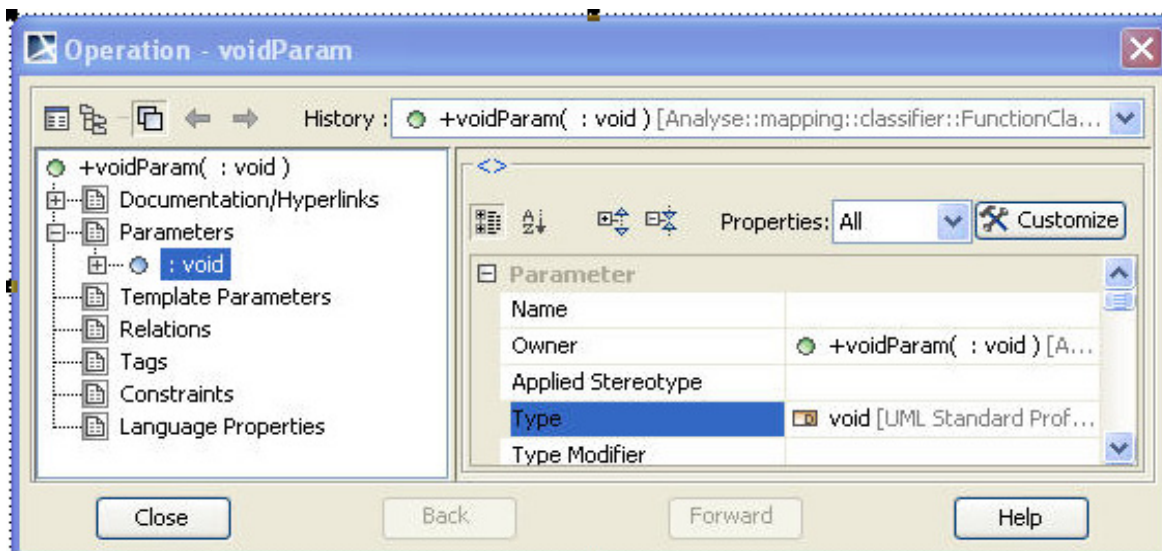
Code	MD-UML
<pre>Class FunctionClass { public:     EllipsisFunc(...); };</pre>	



## void parameter

C++ void function parameter is mapped to a UML parameter without name and with type “void”.

Code	MD-UML
<pre>Class FunctionClass { public:     void voidParam(void); };</pre>	

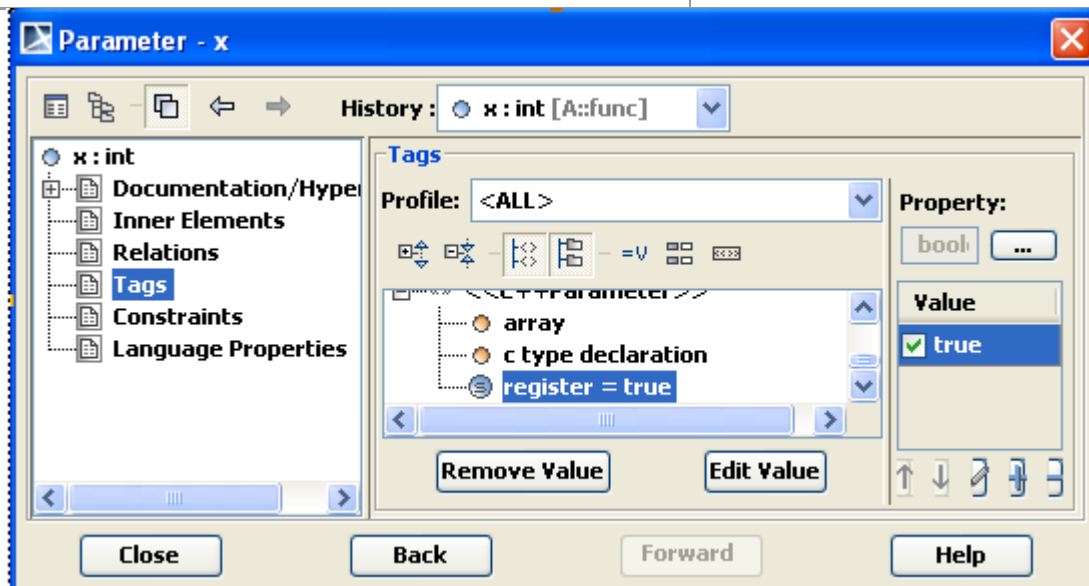


## Register parameter

C++ register parameter is mapped to UML parameter language property Register

Depending on the compiler, register can be limited on some types (int, char).

Code	MD-UML
<pre>class RegisterParamClass {     void registerParam(register int x); };</pre>	



## Function modifiers

C++ function modifiers are mapped to Language properties of Operation.

Virtual function is mapped to *Virtual modifier* property.

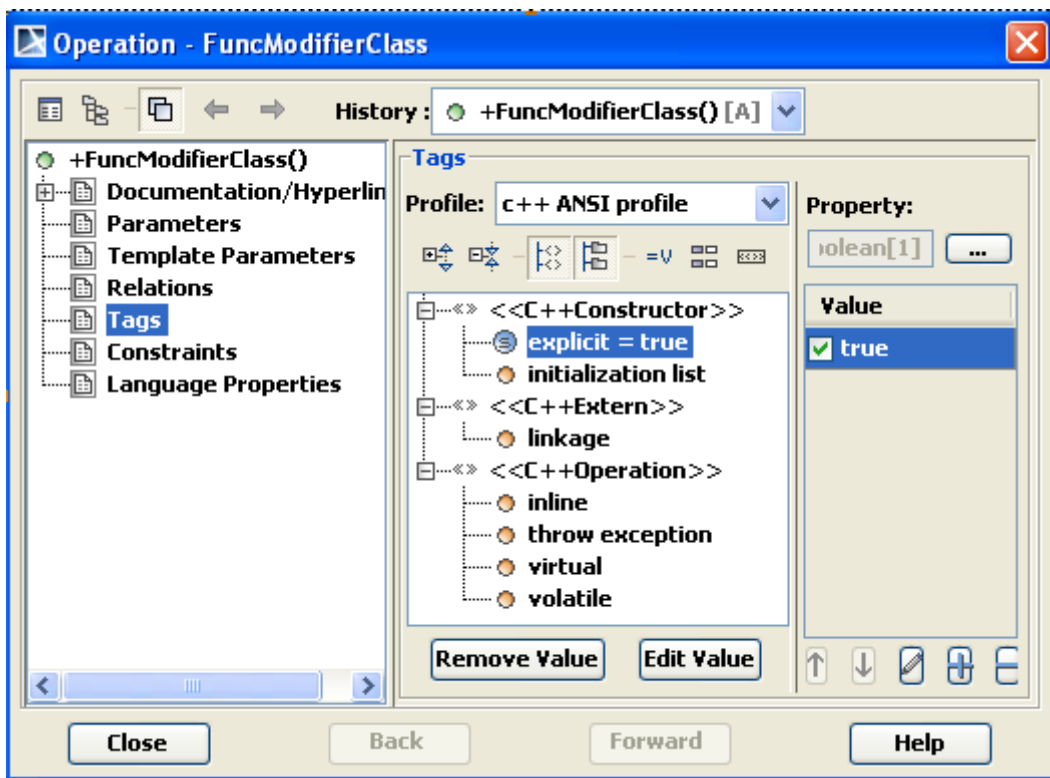
Inline function is mapped to *Inline modifier* property.

Explicit function is mapped to *Explicit modifier* property. Constraint: explicit is only valid for constructor.

Const function is mapped to UML operation *Is Query* property.

Volatile function is mapped to Tag value *volatile*.

Code	MD-UML
<pre>class FuncModifierClass {     explicit FuncModifierClass(); };</pre>	

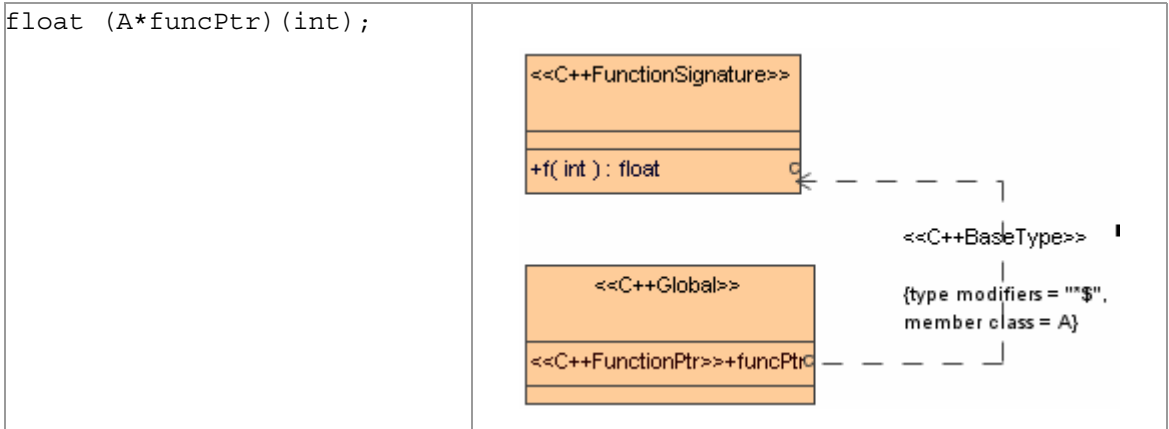


## Function pointer

C++ function pointer type is mapped to attribute/parameter with `<<C++FunctionPtr>>` stereotype, a dependency with `<<C++BaseType>>` stereotype link from the attribute/parameter to the operation in a `<<C++FunctionSignature>>` class, and type modifiers of the dependency is set to `*$`.

Member function pointer use the same mapping, and member class tag of `<<C++BaseType>>` stereotype point to a class.

Code	MD-UML



## Function operator

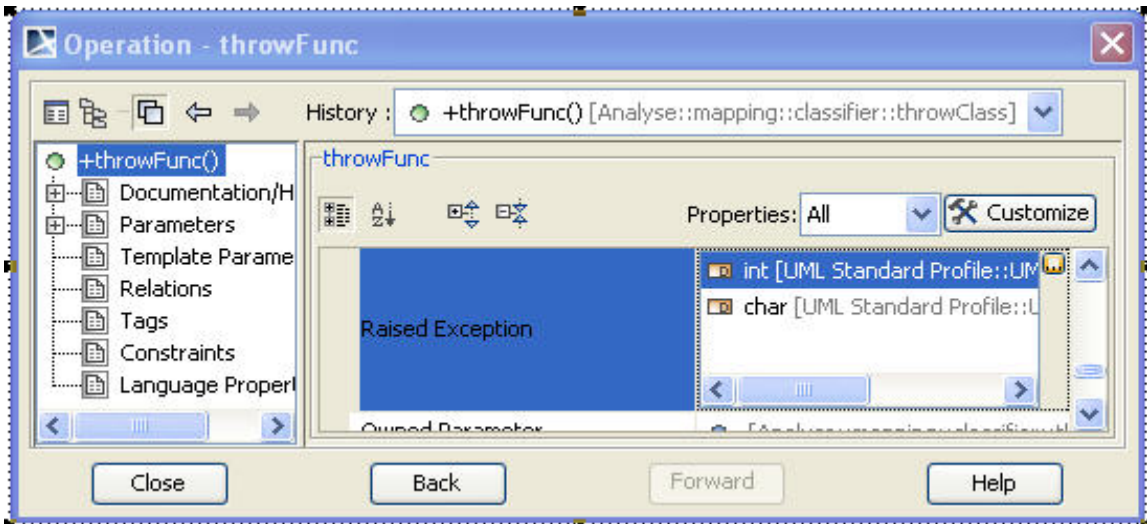
C++ function operator is mapped to normal function with the C++ operator name mapped to UML operation name. See [C++Operator](#) for more info.

Code	MD-UML
<pre>Class Op { Public:     Op operator+(Op x); };</pre>	

## Exception

C++ exception is mapped to UML operation's *raised exception* properties. If *raisedExpression* is empty, and *throw exception* tag is set to *none* a throw without parameter is generated. If *raisedExpression* is empty, and *throw exception* tag is set to *any* throw keyword is not generated. If the tag *throw exception* is not set, then generate specific *raisedExpression*, or do not generate throw if *raisedExpression* is empty.

Code	MD-UML
<pre>void throwFunc() throw (int,char);</pre>	



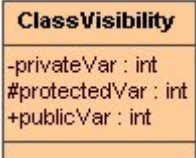
### Visibility

Variables and function visibility are mapped using the UML *visibility* property.

Members of C++ class without access visibility specified are private.

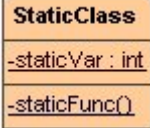
Members of C++ struct or union without access visibility specified are public.

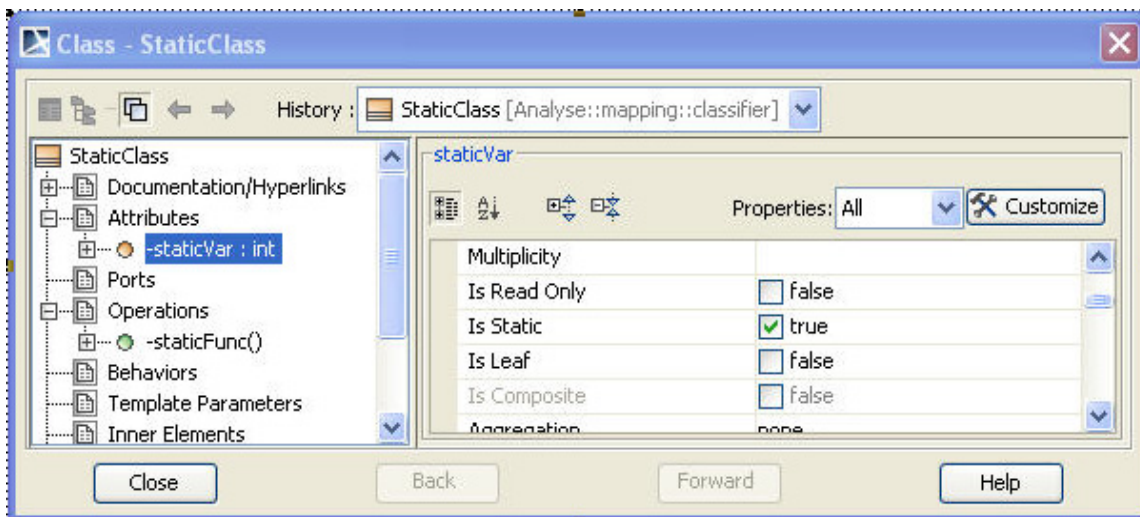
Variables and functions outside a class/struct/union are public.

Code	MD-UML
<pre>class ClassVisibility {     int privateVar; protected:     int protectedVar; public:     int publicVar; };</pre>	

### Static members

Static variables and functions are mapped to UML *Is Static* property.

Code	MD-UML
<pre>class StaticClass {     static int staticVar;     static void staticFunc(); };</pre>	



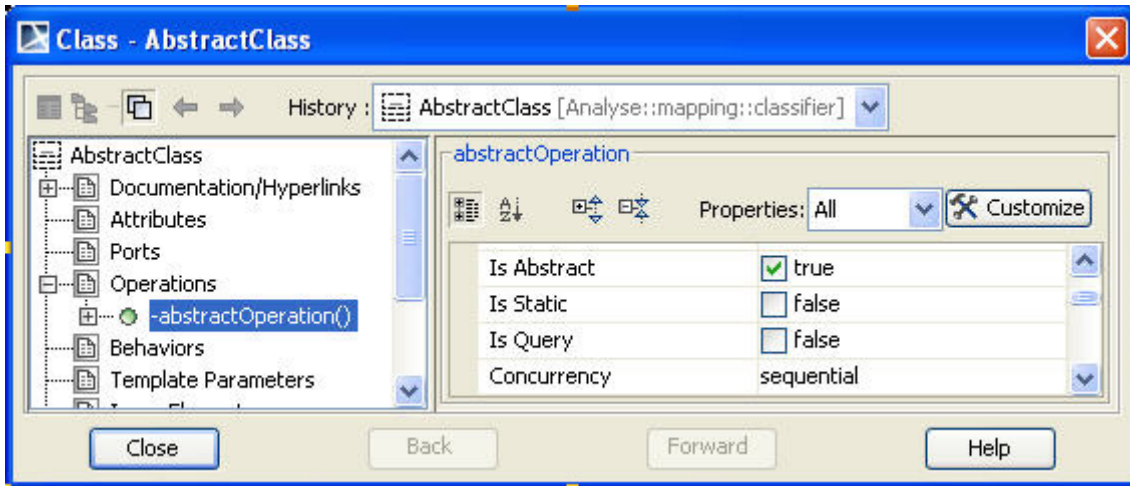
### Pure virtual function and abstract class

Pure virtual C++ function is mapped to UML operation with property *Is Abstract* set to true. If one or more functions are abstract in a class, the property *Is Abstract* of the UML class is set to true.



Constraint: if no operation is abstract, the class can not be abstract.

Code	MD-UML
<pre>class AbstractClass {     virtual abstractOperation()=0; };</pre>	



## Friend declaration

C++ friend function is mapped with a <<C++Friend>> stereotyped dependency relationship between the function (an UML operation) and the friendClass. This relationship grants the friendship to the friendClass. See [C++Friend](#) for more info.

Code	MD-UML
<pre>class ClassB { public:     friend void friendFunc(); }; void friendFunc();</pre>	

C++ friend member function is mapped with a <<C++Friend>> stereotyped dependency relationship between the member function and the friend class. This relationship grants the friendship to the friend class.

Code	MD-UML
<pre>class ClassD {     void func(ClassC c); }; class ClassC {     friend void ClassD::func(ClassC c); };</pre>	

C++ friend class are mapped with a <<C++Friend>> stereotyped dependency relationship between the class and the friendClass. This relationship grants the friendship to the friend class.

Code	MD-UML
<pre>class FriendClass { public:     friend class ClassA; }; class ClassA { };</pre>	

## Struct

C++ struct are mapped to a UML class with stereotype <<C++Struct>>. See [C++Struct](#) for more info.

**NOTE** The current version of MD use class's language property "Class Key"

Code	MD-UML
<pre>struct MyStruc { };</pre>	

## Union

C++ union is mapped to a UML class with stereotype <<C++Union>>. See [C++Union](#) for more info.

**NOTE** The current version of MD use class's language property "Class Key"

Code	MD-UML
<pre>union MyUnion { };</pre>	

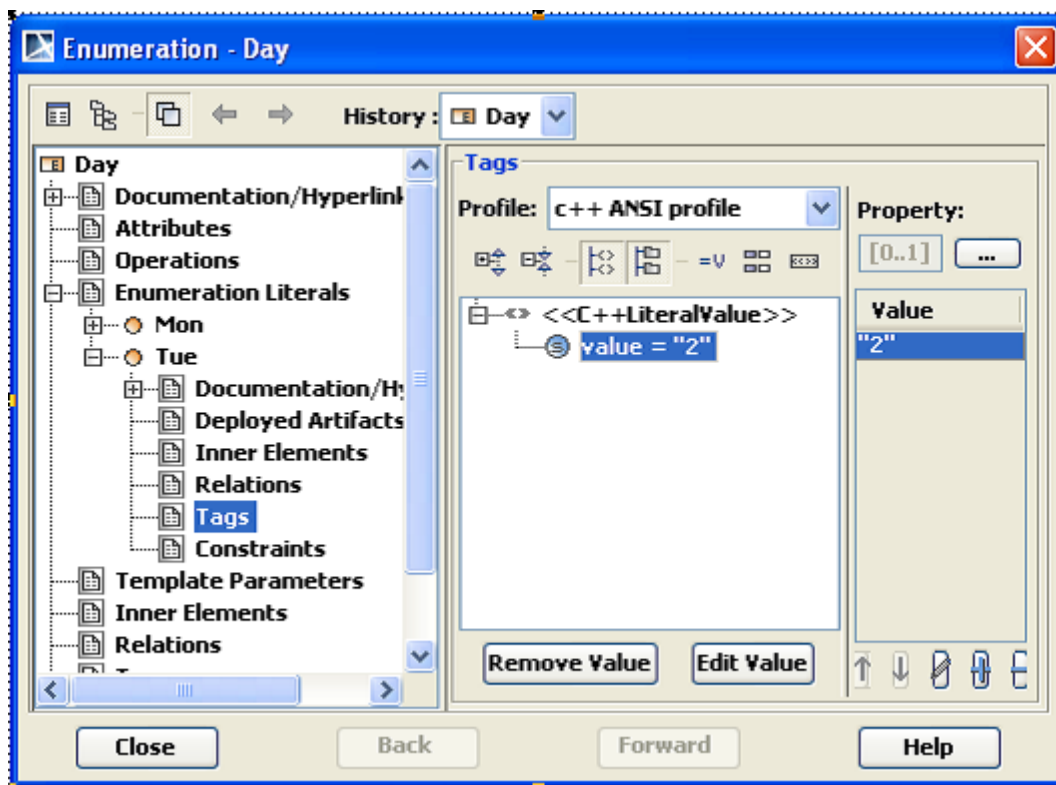
## Enumeration

C++ enum is mapped to UML enumeration. C++ enum fields are mapped to UML enumeration literals.

C++ enum field with a specified value is mapped to tag value of <<C++LiteralValue>> stereotype..

Code	MD-UML
<pre>enum Day {     Mon,     Tue=2 };</pre>	





## Typedef

C++ typedef is mapped to a class with `<<C++Typedef>>` stereotype. A `<<C++BaseType>>` dependency links to the original type.

Type modifiers tag of `<<C++BaseType>>` dependency is used to define type modifiers. \$ character is replaced by the type name.

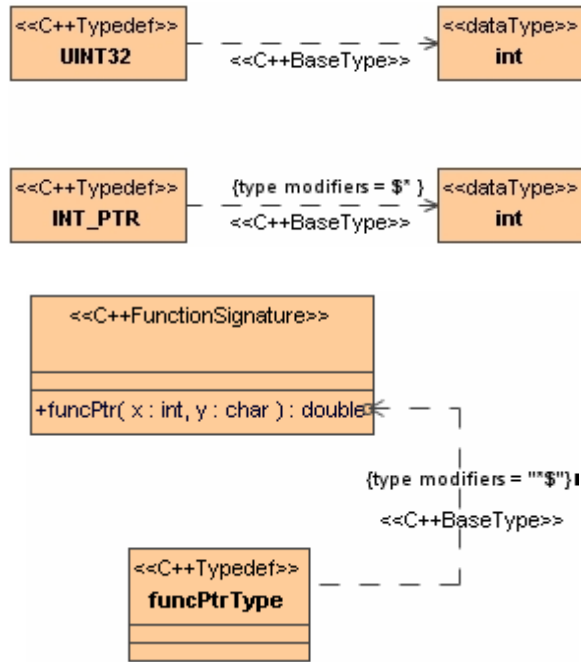
A typedef on a function pointer is mapped by linking a `<<C++BaseType>>` dependency to an operation and type modifiers tag of `<<C++BaseType>>` dependency is set to `*$`. Operation signature can be stored in a `<<C++FunctionSignature>>` class.

See [C++ Typedef](#) for more info.

Code

MD-UML

```
typedef int UINT32;
typedef int* INT_PTR;
typedef double (*funcPtrType)(int,
char);
;
```



## Namespace

C++ namespace is mapped to a UML package with the stereotype <<C++Namespace>>. See [C++Namespace](#) for more info.

Unnamed namespace is named unnamed+index number of unnamed namespace (start at 1), and *unique namespace name* tag is set to the source file path+:+index number of unnamed namespace (start at 0).

Code	MD-UML
<pre>namespace n {     namespace m {     } }</pre>	<p>The diagram shows a UML package structure where a large package labeled &lt;&lt;namespace&gt;&gt; <b>n</b> contains a smaller package labeled &lt;&lt;namespace&gt;&gt; <b>m</b>.</p>

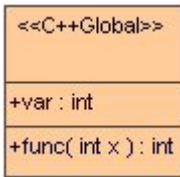
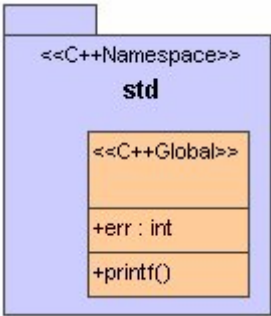
## <sup>1</sup>Global functions and variables

Global functions and variables are mapped to operations and attributes into an unnamed class with stereotype <<C++Global>>. <<C++Global>> class resides in its respective namespace, or in a top package .

See [C++Global](#) for more info.

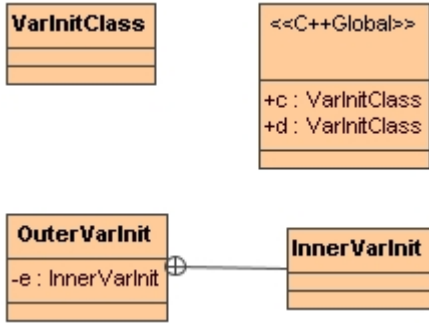
Code	MD-UML

1.

<pre>int var; int func(int x);</pre>	
<pre>namespace std {     int err;     void printf(); }</pre>	

### Class definition

Variables can be created after a class/struct/union declaration. These variables are mapped to UML attribute, and placed in their respective namespace/global/class container.

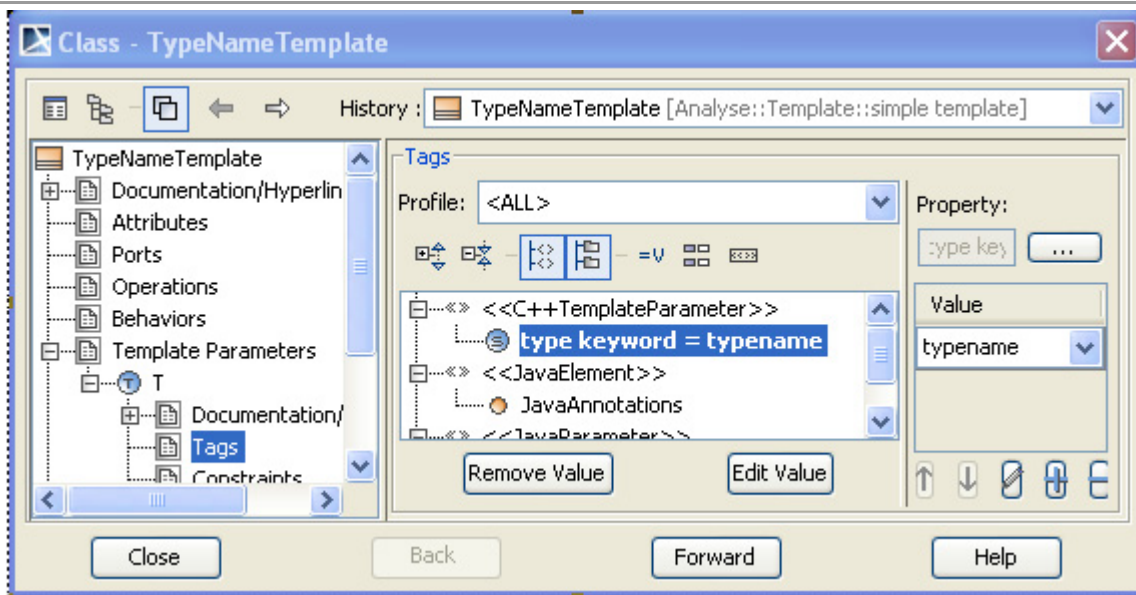
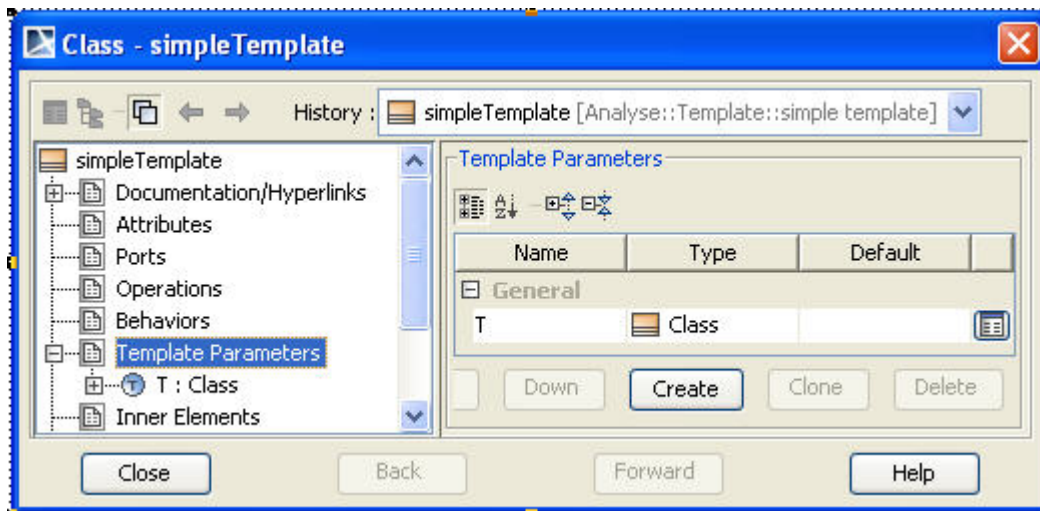
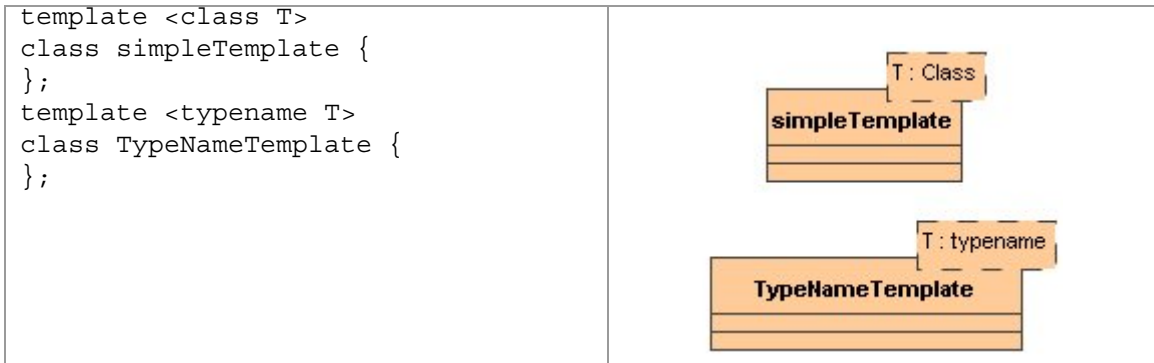
Code	MD-UML
<pre>class VarInitClass { } c, d; class OuterVarInit {     class InnerVarInit {     } e; };</pre>	

### Class Template Definition

C++ template class is mapped to UML class with template parameters properties added.

Type of template parameter is always set to UML Class. To generate/reverse typename keyword, *type keyword* tag is set to *typename*.

Code	MD-UML
------	--------




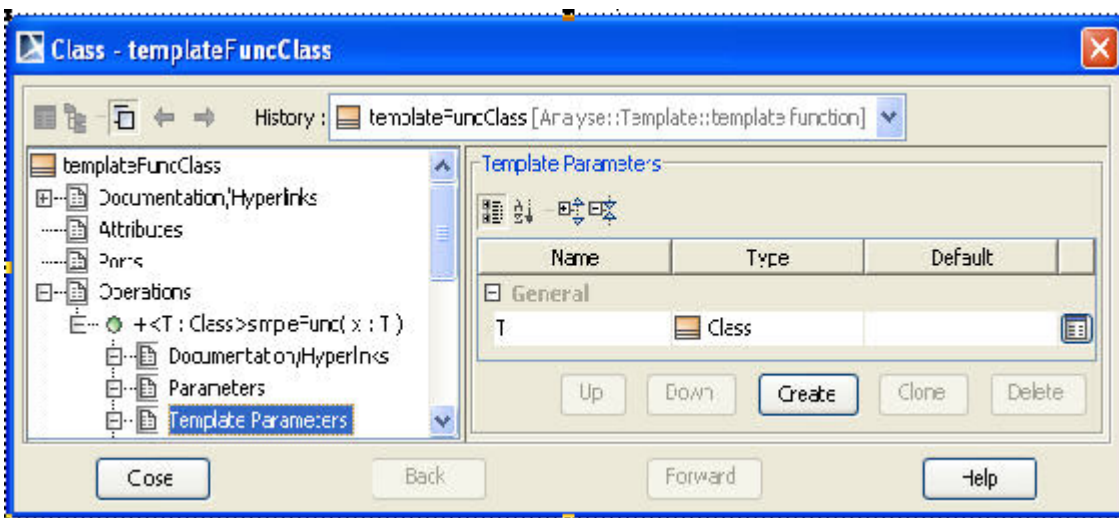
## Function Template Definition

C++ template function is mapped to UML operation with template parameters properties added.

C++ template function overload is mapped to a normal function. (the same name with the same number of parameter, but different type of parameter)

New style of template function overloading is mapped to a normal function. (the same name with the same number of parameter, but different type of parameter) and a template binding relationship is created between the overload operation and the template operation, with specific template parameter substitutions.

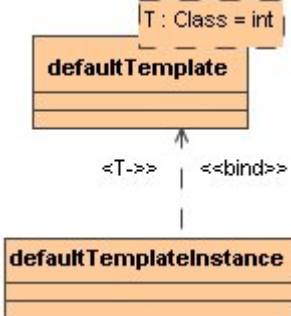
Code	MD-UML
<pre>template &lt;class T&gt; void simpleFunc(T x); // overload old style void simpleFunc(int x); // overload new style template&lt;&gt; void simpleFunc&lt;char&gt;(char x);</pre>	 <p>The diagram shows a class-like structure with the stereotype &lt;&lt;C++Global&gt;&gt;. It lists three methods: +&lt;T : Class&gt;simpleFunc(x : T), +simpleFunc(x : int), and +simpleFunc(x : char).</p>

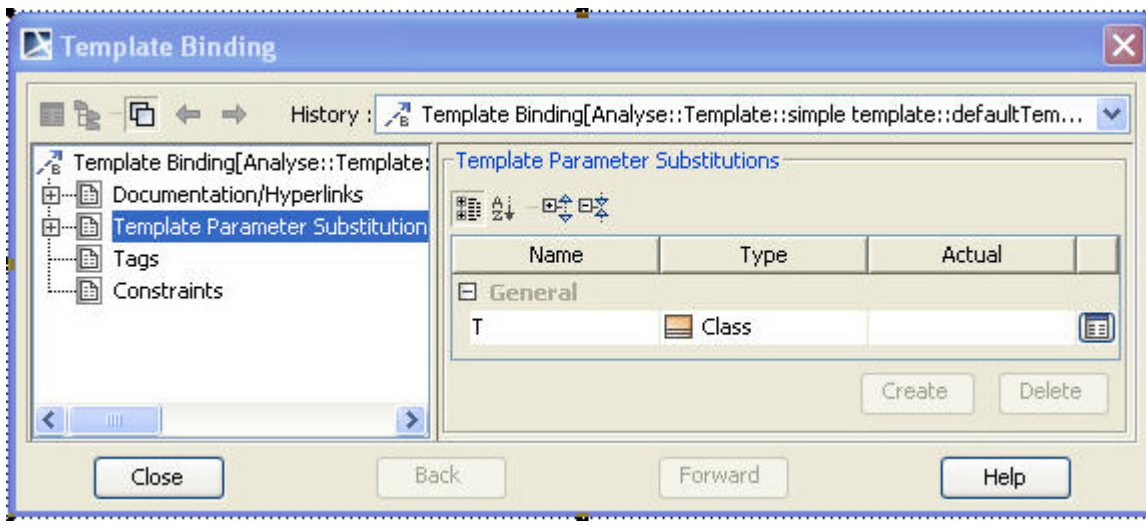
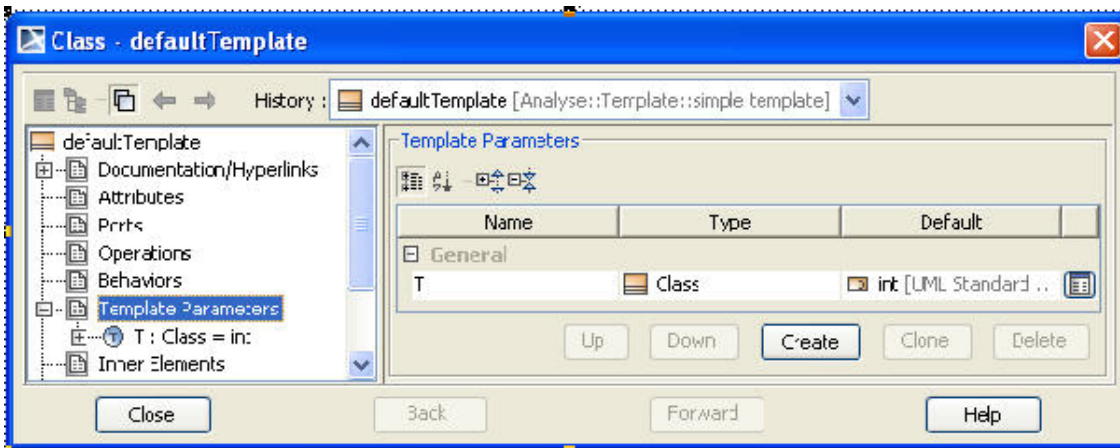


## Default template parameter

C++ default template parameter is mapped to UML default template parameters.

Instantiation using the default template parameter is mapped using a template binding relationship with an empty *actual* property for the *template parameter substitution*.

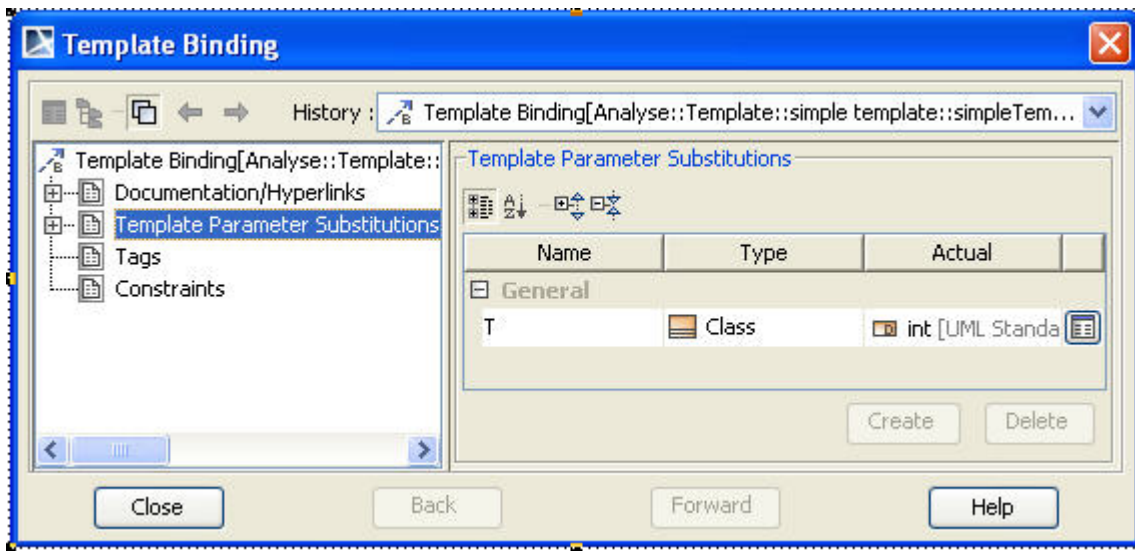
Code	MD-UML
<pre>template &lt;class T=int&gt; class defaultTemplate { };</pre>	 <p>The diagram shows two classes: 'defaultTemplate' and 'defaultTemplateInstance'. 'defaultTemplate' has a stereotype 'T : Class = int'. A binding relationship arrow points from 'defaultTemplateInstance' to 'defaultTemplate', with the label '&lt;T-&gt;&gt;' on the left and '&lt;&lt;bind&gt;&gt;' on the right.</p>



## Template instantiation

Template instantiation are mapped to template binding relationship between the template class and the instantiate class, the *template parameter substitution* of the binding relationship is set using the template argument.

Code	MD-UML
<pre> template &lt;class T&gt; class simpleTemplate { };  simpleTemplate&lt;int&gt; simpleTemplateInstance;                     </pre>	



For template argument using template instantiation as argument, an intermediate class is created with the specific binding



Code	MD-UML
<pre> template &lt;class T&gt; class T1Class { }; template &lt;class T&gt; class T2Class { };  T1Class&lt;T2Class&lt;int&gt;&gt; ...                     </pre>	

For template argument using multiple template instantiations in an inner class (b<int>::c<char>), the intermediate class instance is created in the outer class instance.

Code	MD-UML
<pre> template &lt;class T&gt; class b {     template &lt;class T&gt;     class c {     }; };  b&lt;int&gt;::c&lt;char&gt; ...                     </pre>	

Example of complex template instantiation. Containment relationships are placed on diagram for information only, these relationships are not created during a reverse process. Containment relationship is modeled by placing a class into a specific class/package. See Containment tree below the diagram.



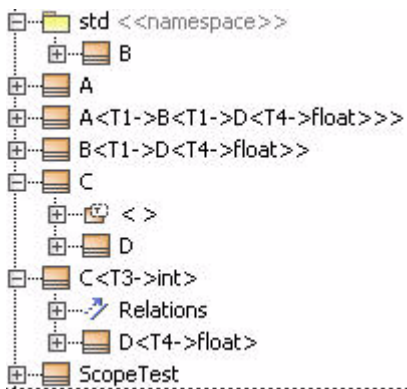
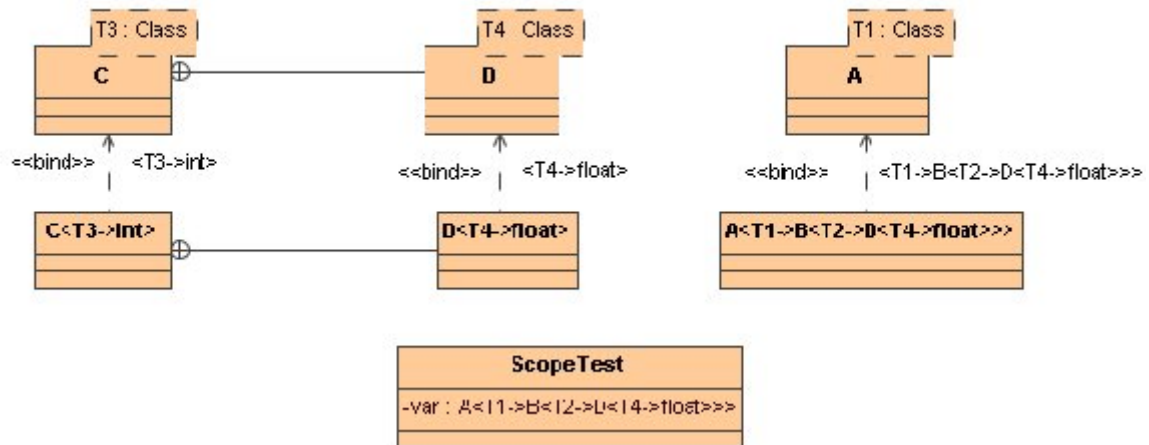
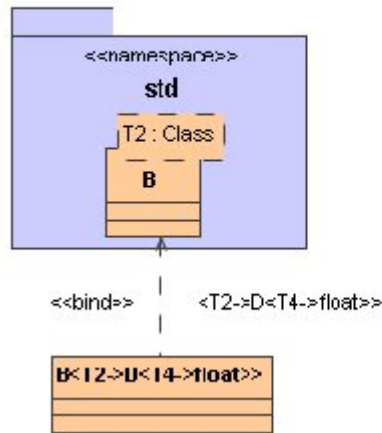
```

template <class T1>
class A {};

namespace std {
    template <class T2>
    class B {};
}

template <class T3>
class C {
public:
    template <class T4>
    class D {};
};

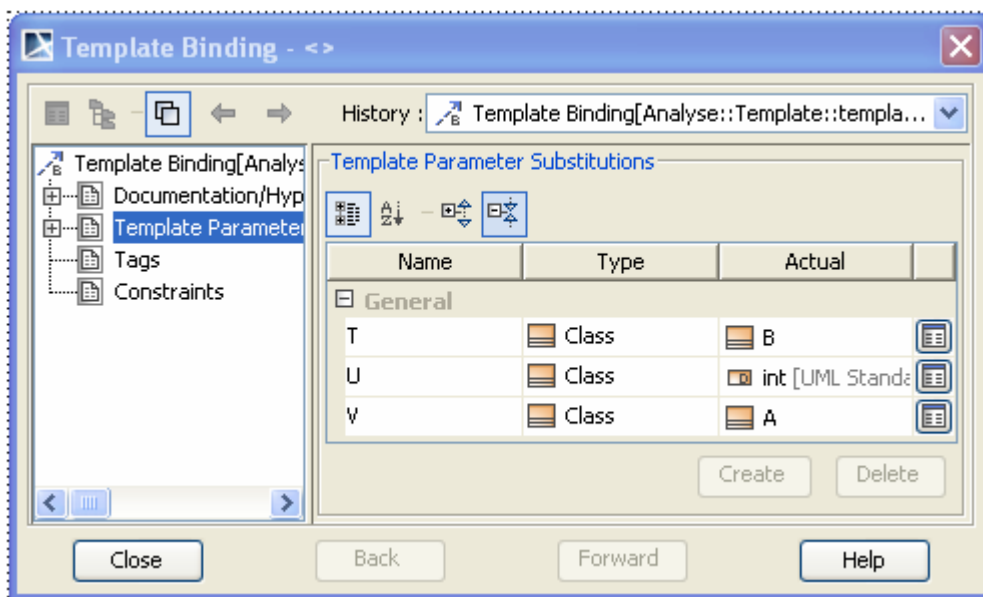
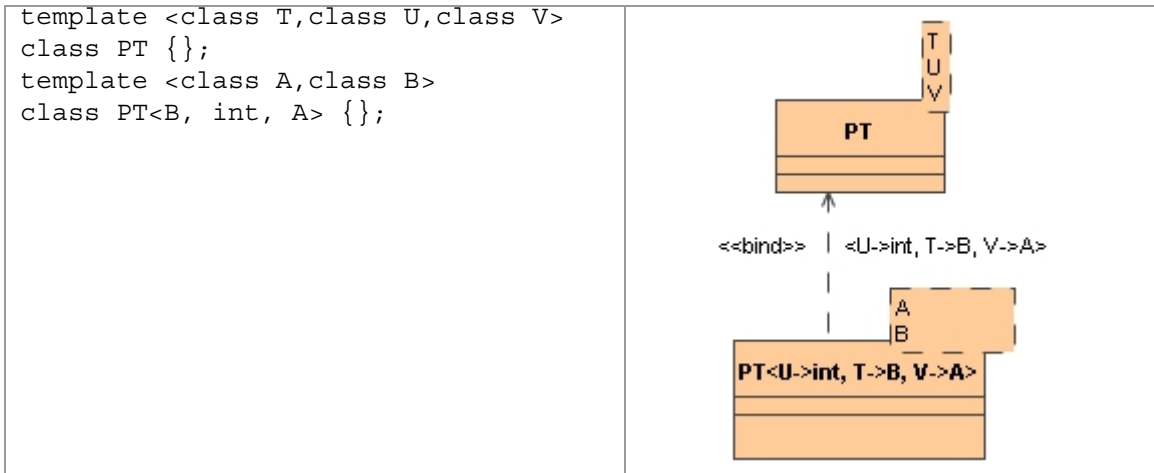
class ScopeTest {
    A<std::B<C<int>::D<float>>> var;
};
    
```



## Partial template instantiation

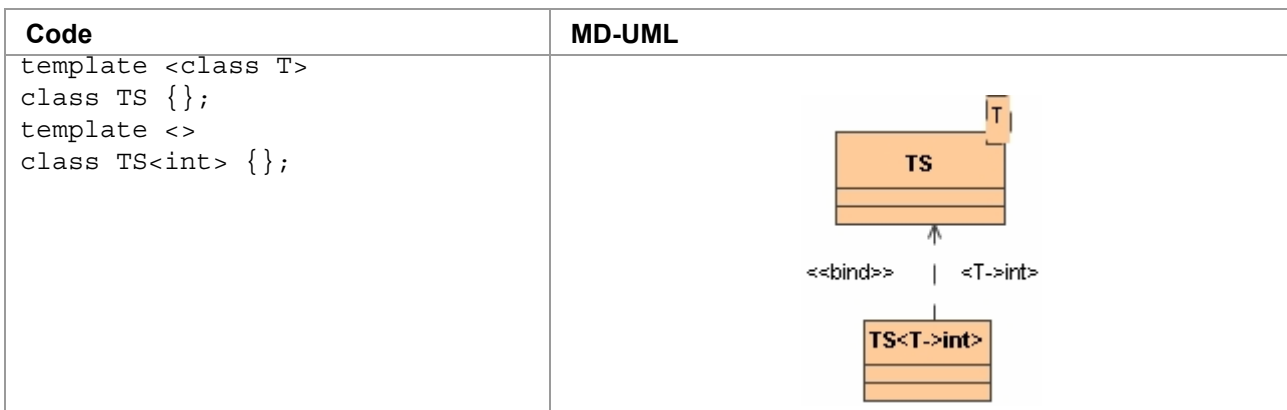
C++ partial template instantiation use the same mapping as Template Instantiation and the unbinded parameter is binded to the specific template parameter class.

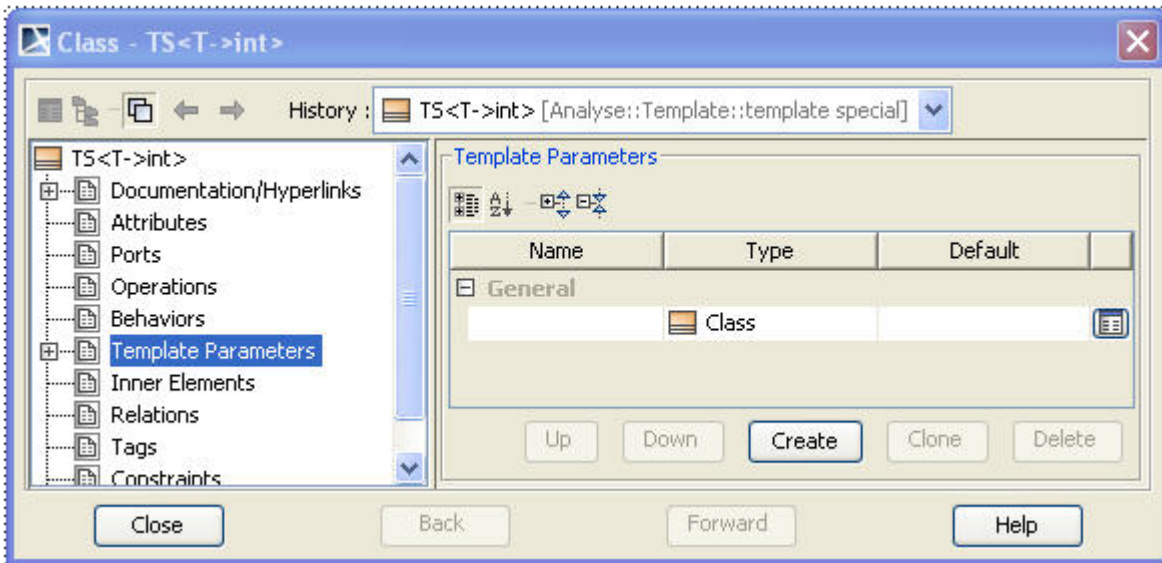
Code	MD-UML
------	--------



## Template specialization

C++ Template specialization uses the same mapping as Template Instantiation.

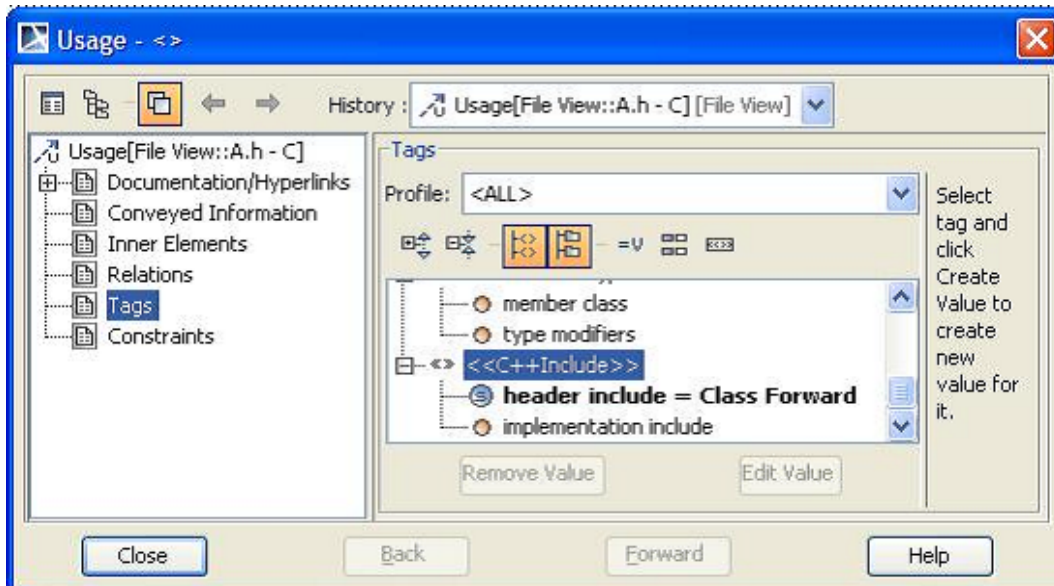
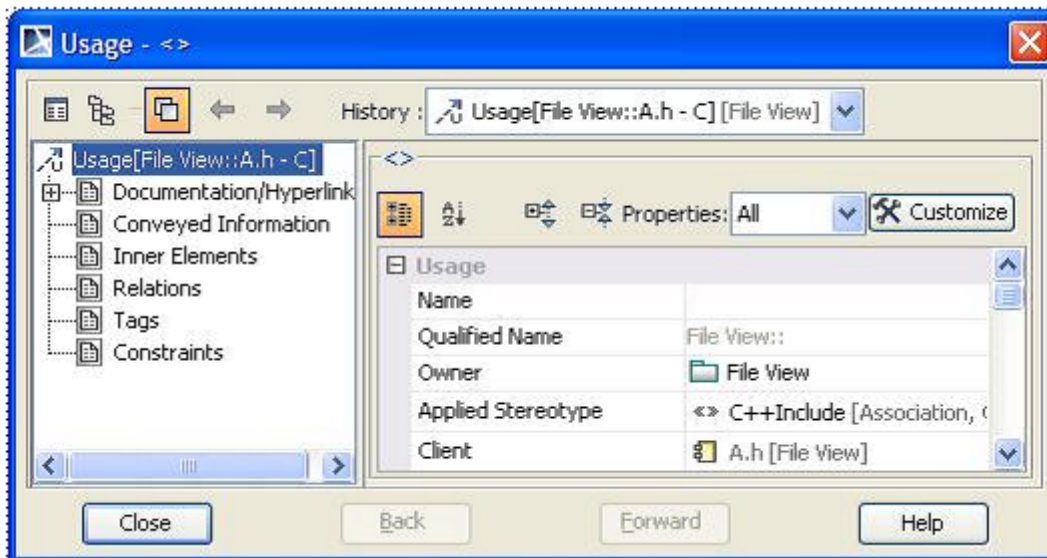




## Forward class declaration

The example code is declared in A.h file. The file component A.h has the <<use>> association applied by <<C++Include>> stereotype with "Class Forward" tag value.

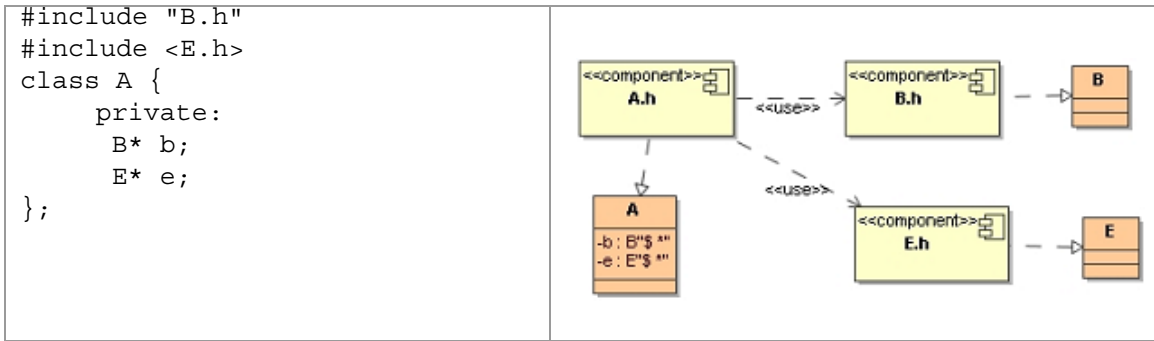
Code	MD-UML
<pre>class C; class A {   private:     C* c; };</pre>	



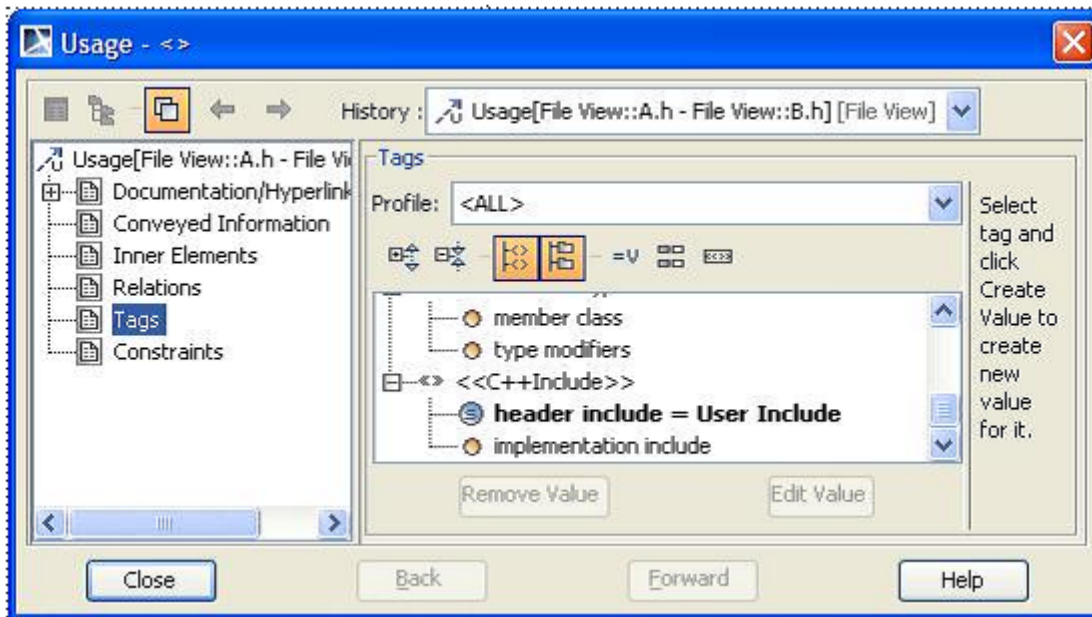
## Include declaration

The example code is declared in A.h file. The <<use>> association is also applied to C++Include stereotype shown in section 3.36 Forward class declaration.

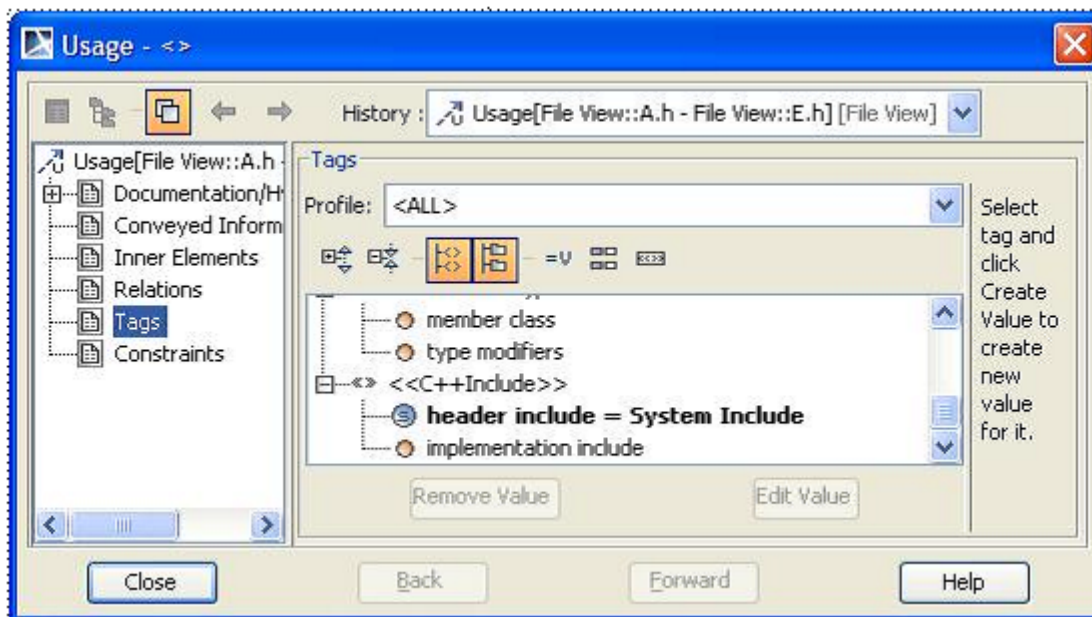
<b>Code</b>	<b>MD-UML</b>
-------------	---------------



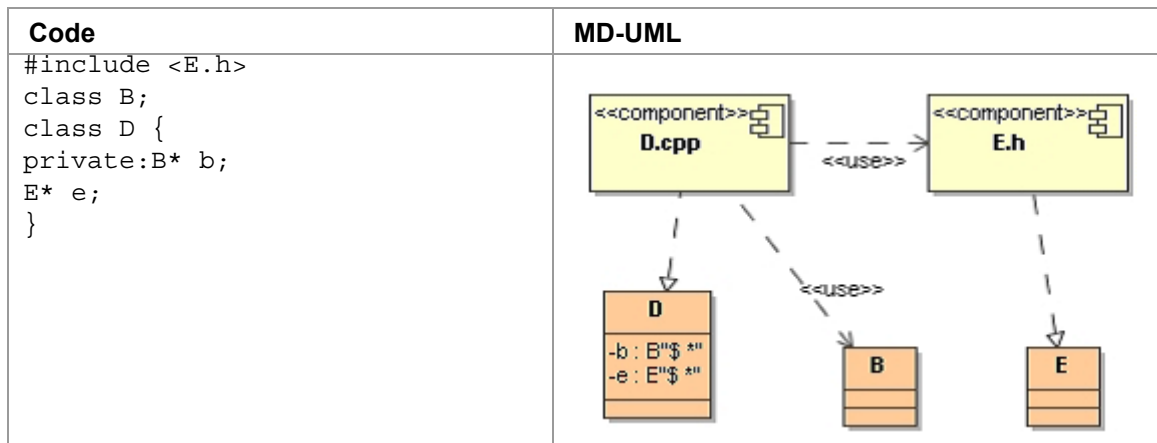
Specification for #include "B.h"



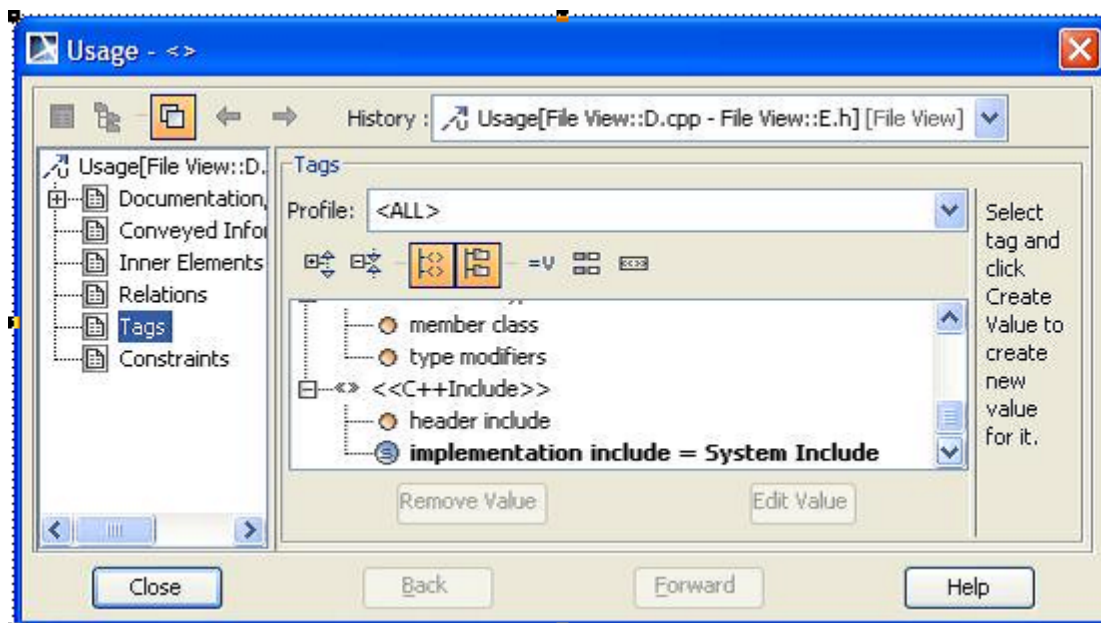
Specification for #include <E.h>



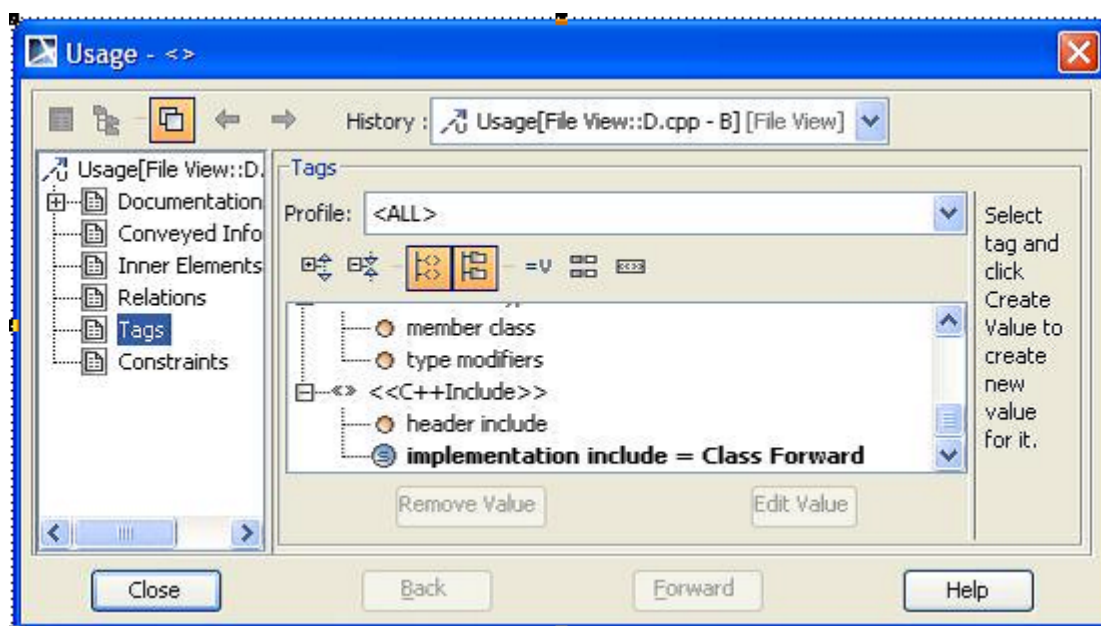
The example code is declared in D.cpp file.



Specification for #include <E.h>



Specification for class B;





## Conversion from old project version

This chapter describes the changes applied when loading a MD project version  $\leq 11.6$ .

### Translation Activity Diagram

There are projects that use C++ language properties or C++ profile or have type modifier, which need to be translated with version of MagicDraw project less than or equal 11.6.

#### Open local project

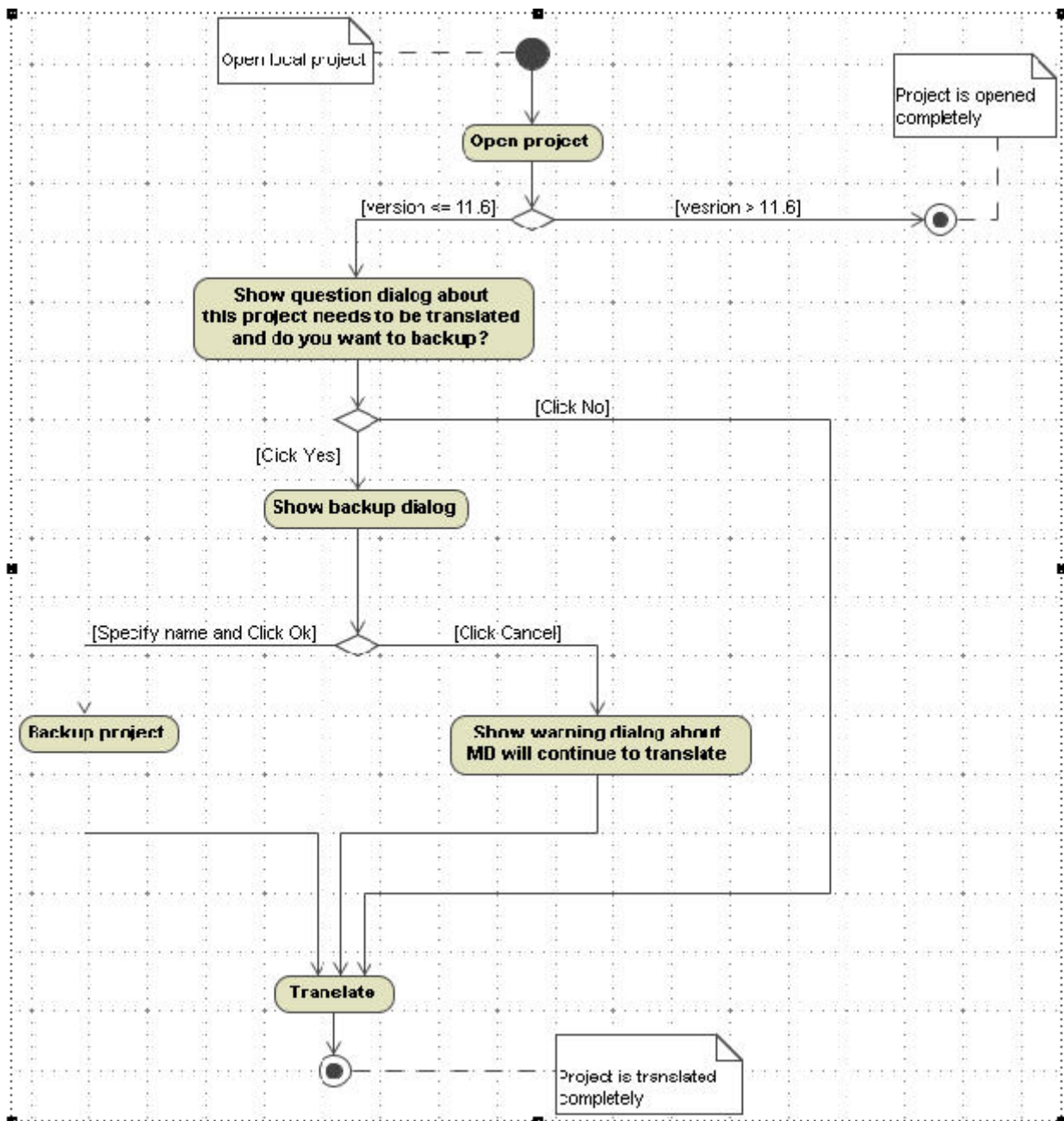


Figure 55 -- Open local project Activity Diagram

### Open teamwork project

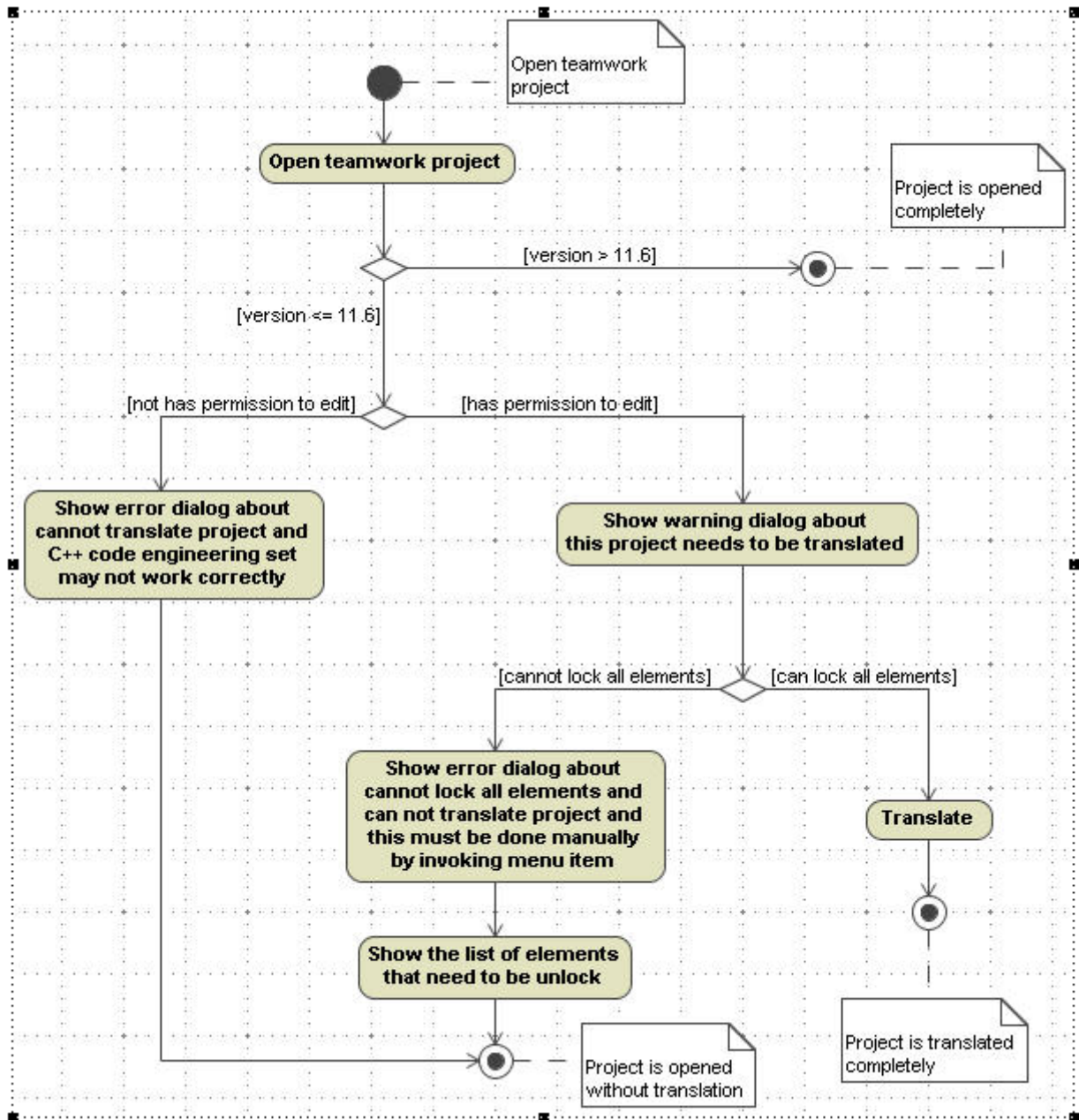


Figure 56 -- Open teamwork project Activity Diagram



## Import MagiDraw project

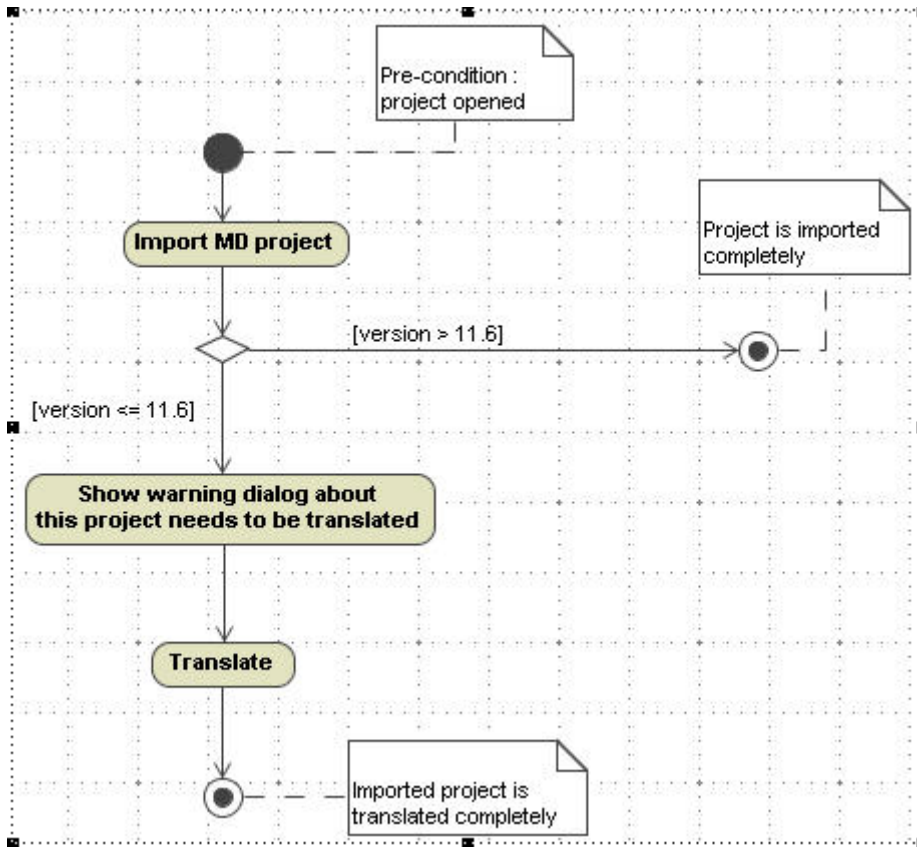


Figure 57 -- Import MagicDraw project Activity Diagram

## Use module

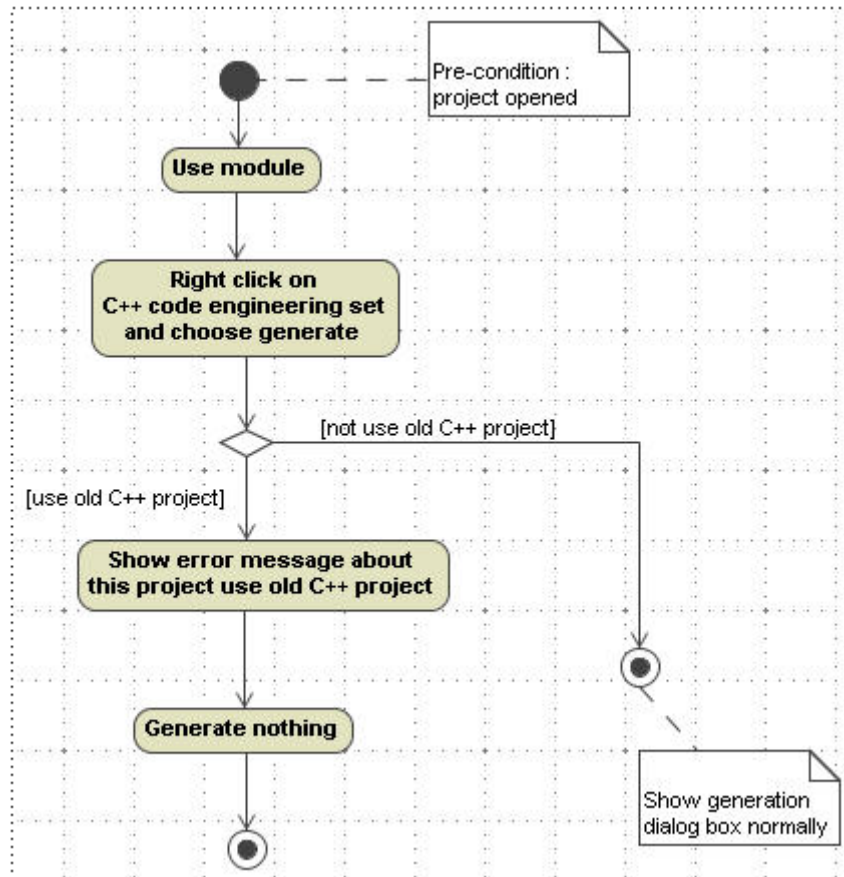


Figure 58 -- Use module Activity Diagram

## Update C++ Language Properties and Profiles

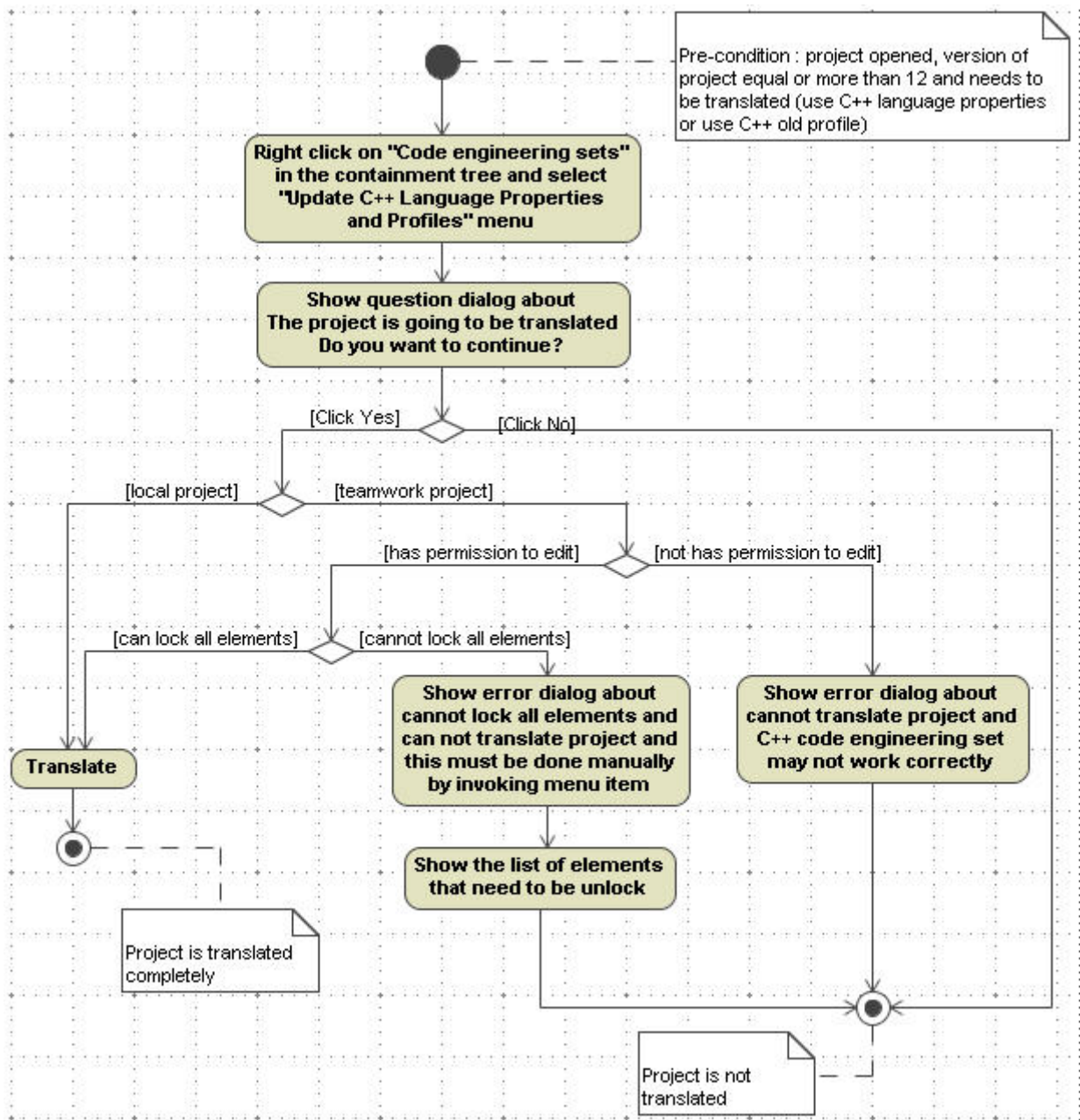


Figure 59 -- Update C++ Language Properties and Profiles Activity Diagram

## Language properties

Until MD version 11.6, language properties are stored in a specific format, since MD version 12 language properties are moved to stereotype's tag value.

## Class

There are **class**, **struct** and **union** class key in Class Language Properties.

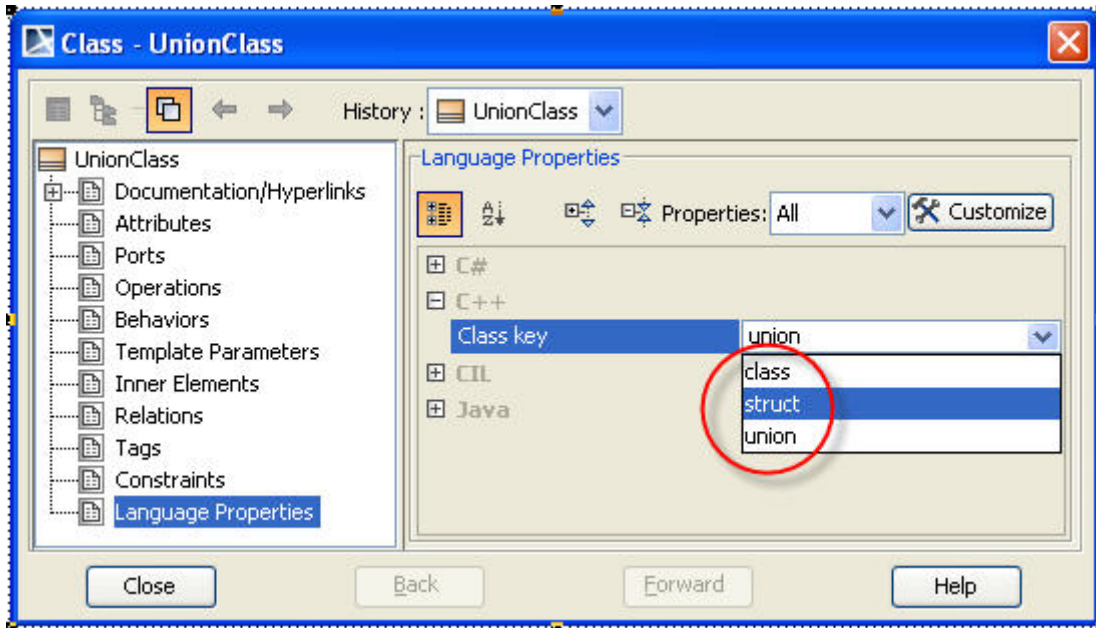
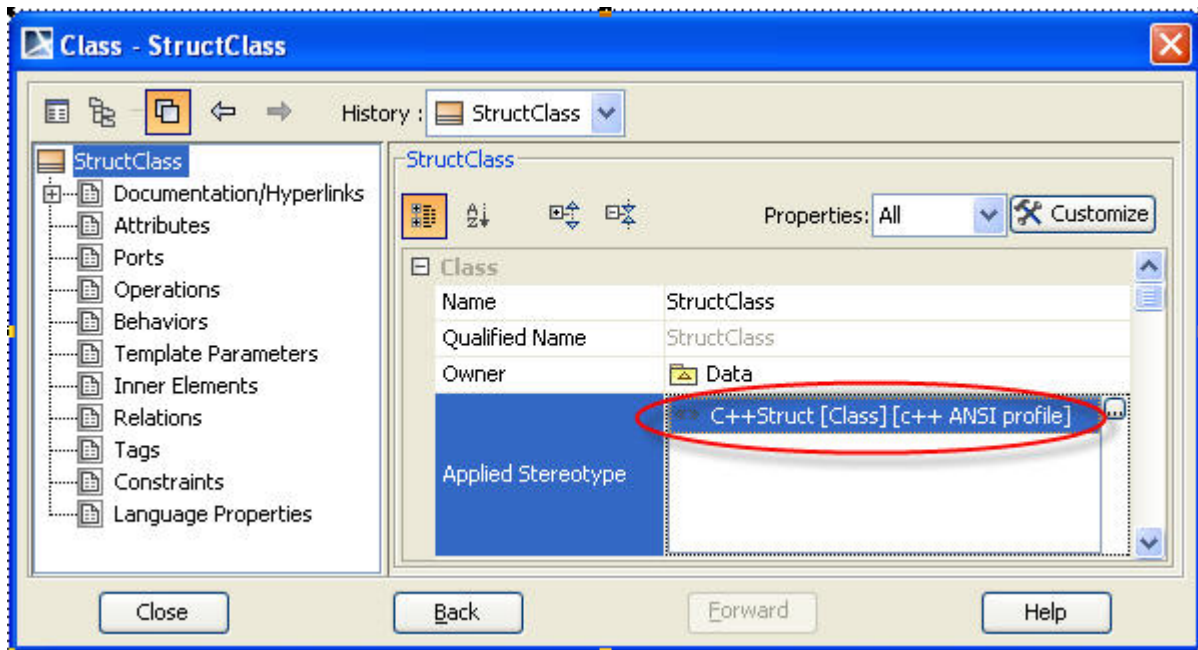


Figure 60 -- Class Language Properties

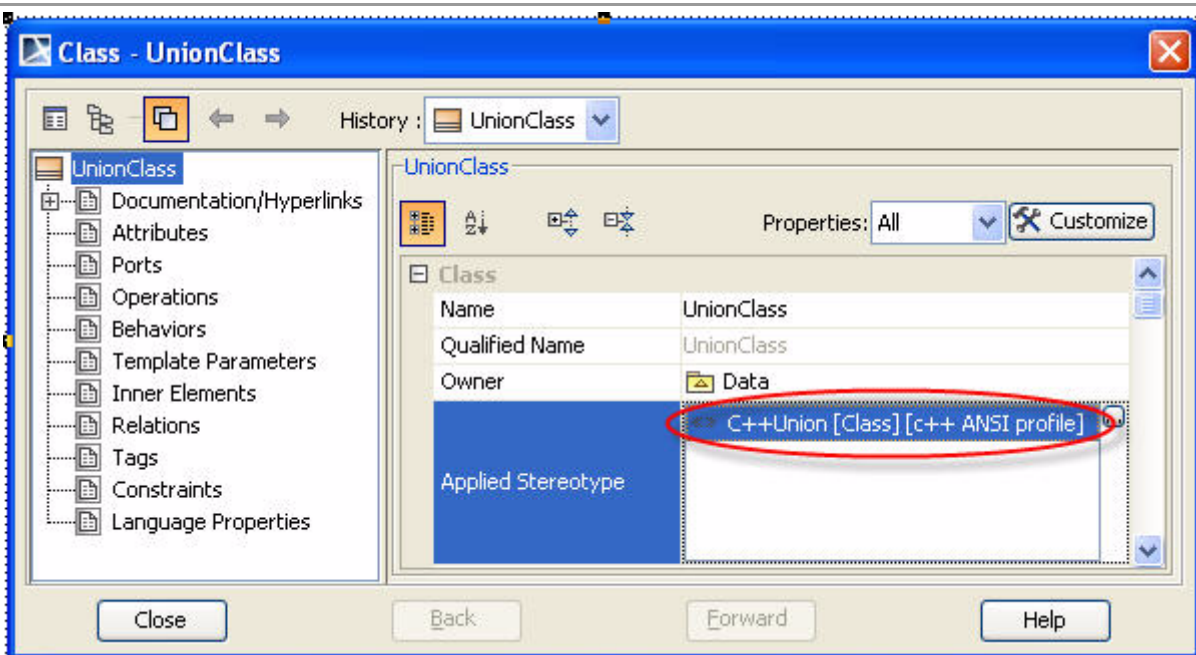
## Class - Class key

Old value	Translation
class	no change.
struct	Apply the <<C++Struct>> stereotype.



union

Apply the <<C++Union>> stereotype.





## Operation

This example is **OperationClass** class that has operation named **OperationClass()** which is constructor and operation named **myOperation**.

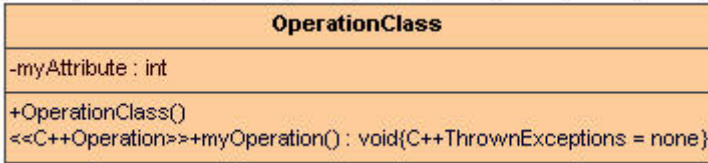


Figure 61 -- Operation Example in Class Diagram

The Model that is being shown in the figure below is a translation.

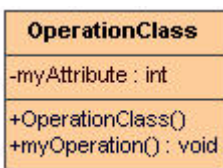


Figure 62 -- Translated Operation in Class Diagram

There are **Inline modifier** and **Virtual modifier** in Operation Language Properties that need to be translated and apply the <<C++Operation>> stereotype.

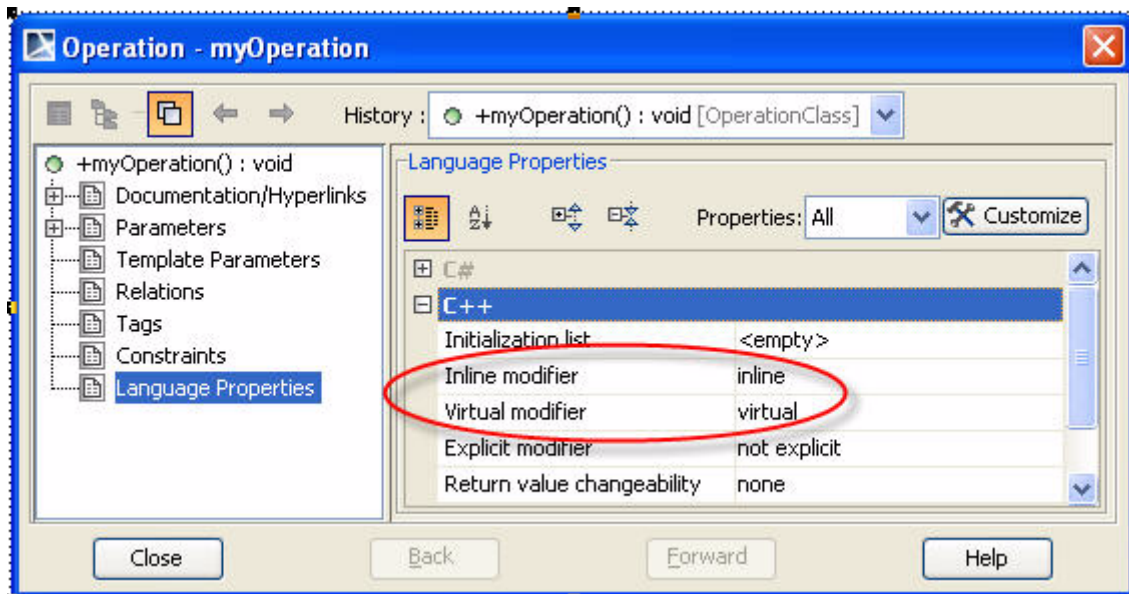


Figure 63 -- Operation Language Properties (Operation)

There are **Initialization list** and **Explicit modifier** in Operation Language Properties that need to be translated and apply the <<C++Constructor>> stereotype.

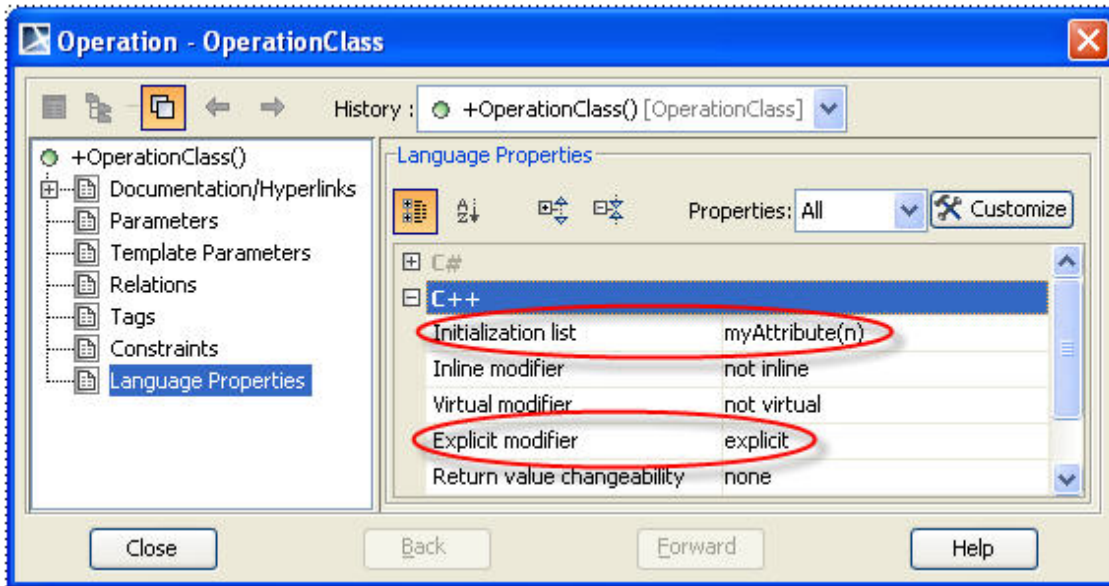


Figure 64 -- Operation Language Properties (Constructor)

Note: Initialization list and Explicit modifier will be translated when it was set in Constructor. The Constructor is an operation that has the same name as its owner or applies the <<constructor>> stereotype in UML Standard Profile.

Operation - Initialization List

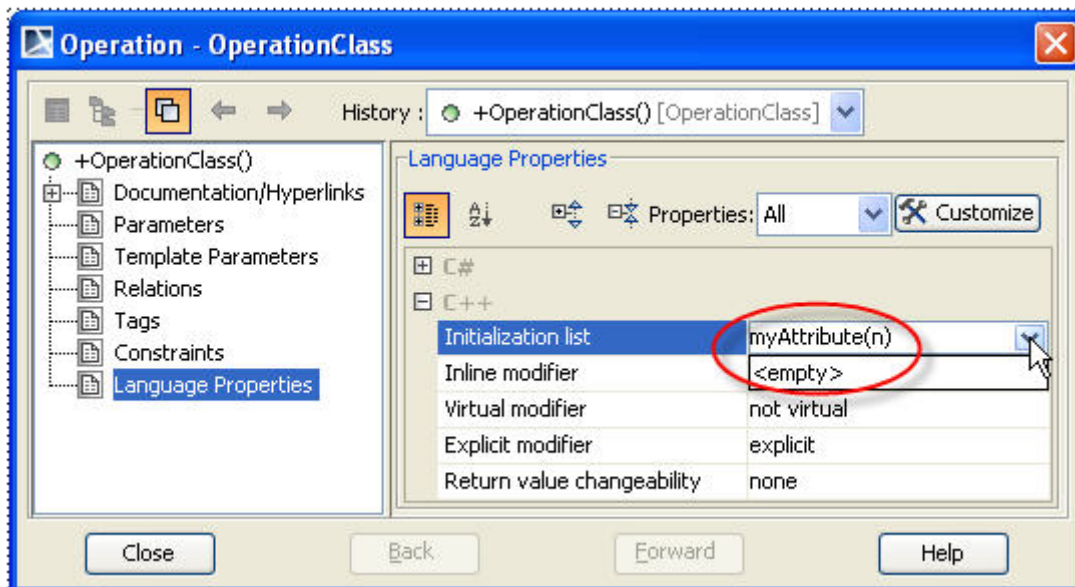
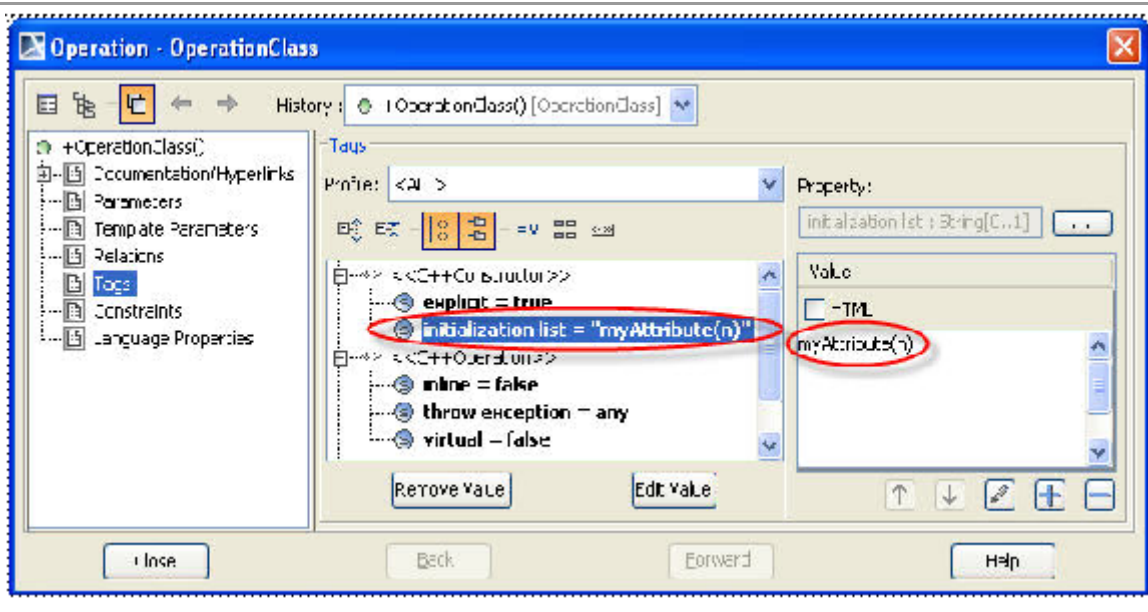


Figure 65 -- Operation Initialization list

Old value	Translation
<empty>	no change.
myAttribute(n)	Apply the <<C++Constructor>> stereotype and set <b>initialization list</b> tag value to <b>myAttribute(n)</b> .



Operation - Inline Modifier

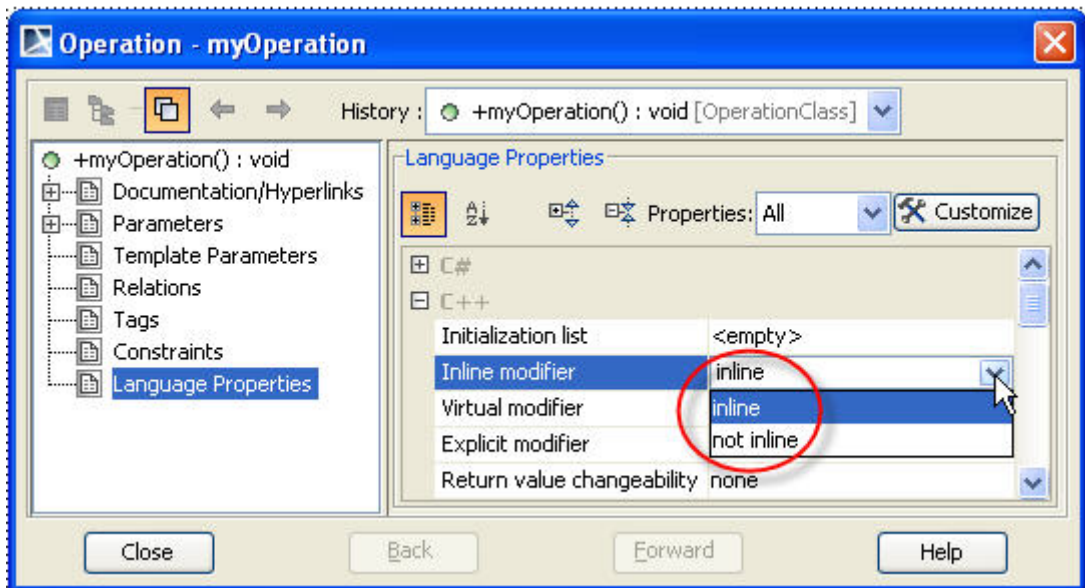
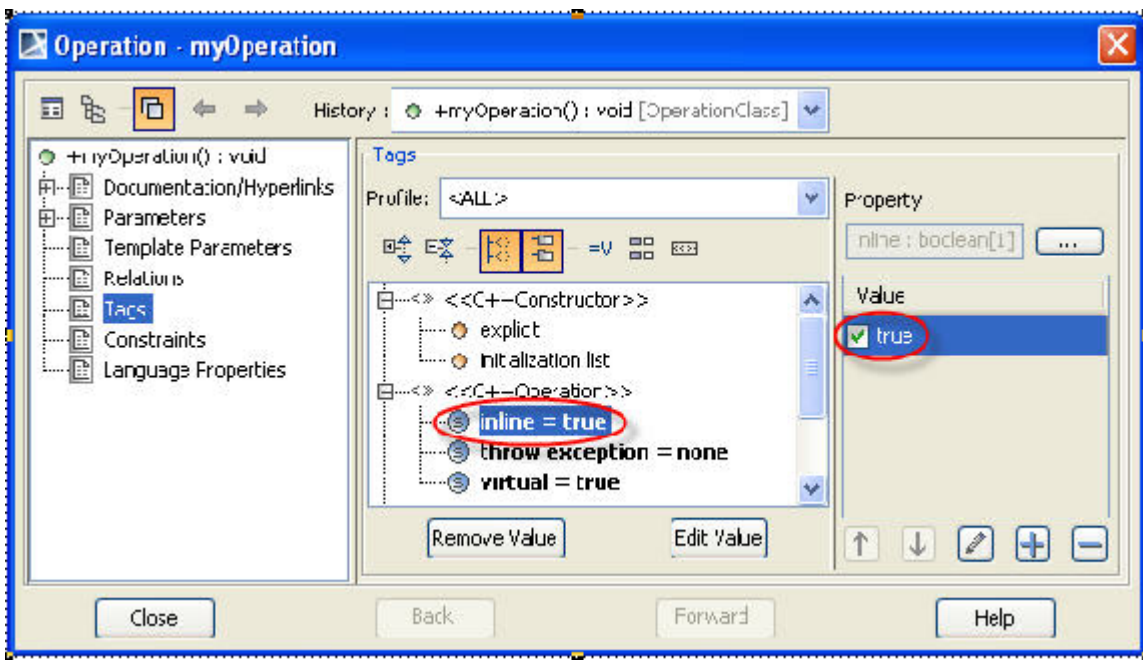


Figure 66 -- Operation Inline modifier

Old value	Translation
not inline	Apply the <<C++Operation>> stereotype and set <b>inline</b> tag value to <b>false</b> .
inline	Apply the <<C++Operation>> stereotype and set <b>inline</b> tag value to <b>true</b> .





Operation - Virtual Modifier

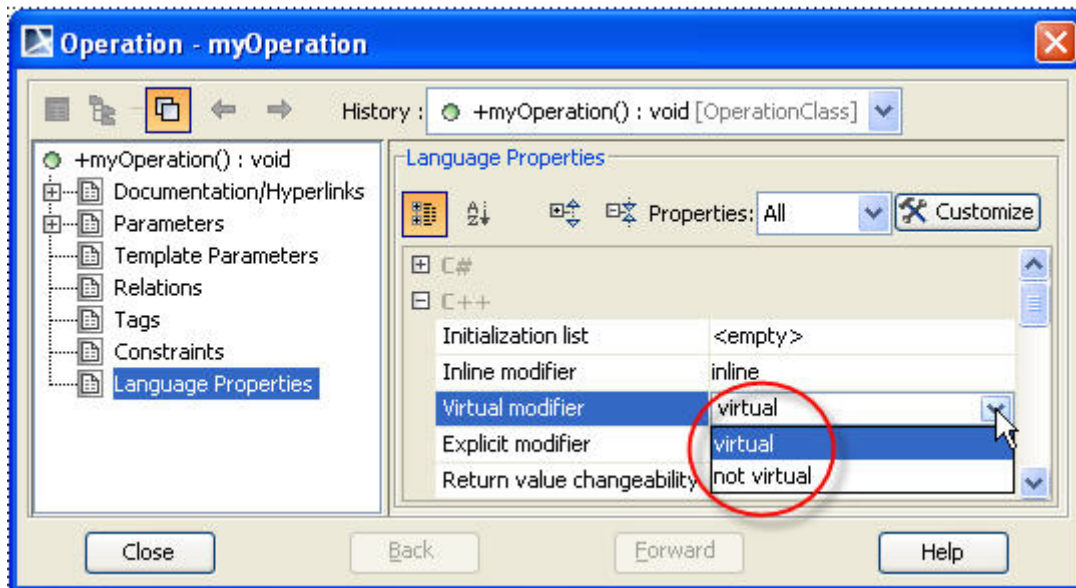
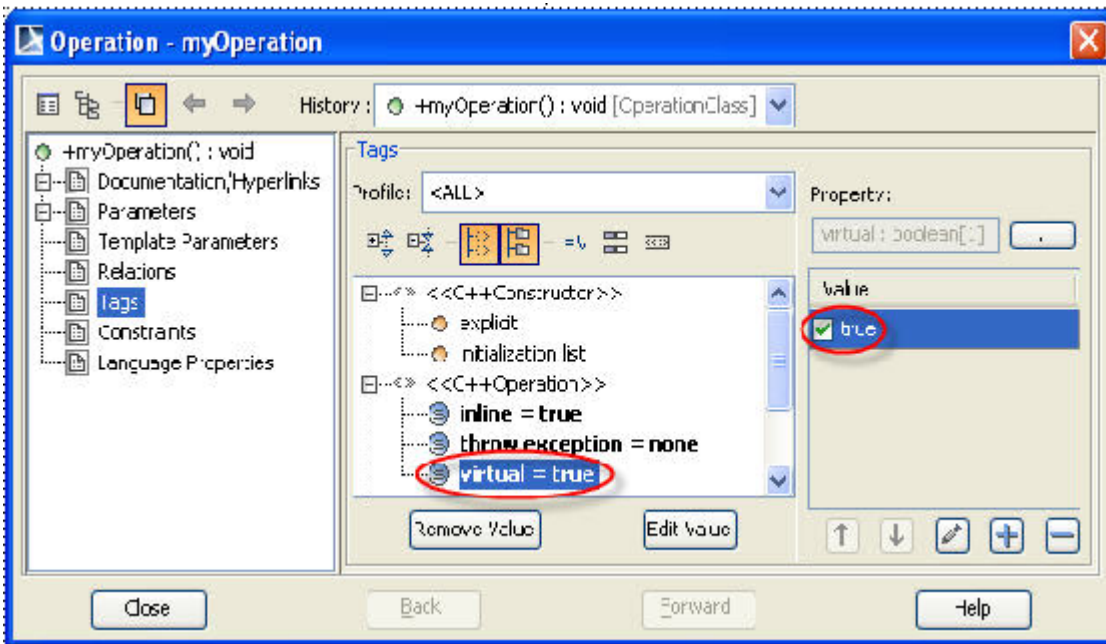


Figure 67 -- Operation Virtual modifier

Old value	Translation
not virtual	Apply the <<C++Operation>> stereotype and set <b>virtual</b> tag value to <b>false</b> .
virtual	Apply the <<C++Operation>> stereotype and set <b>virtual</b> tag value to <b>true</b> .



Operation - Explicit Modifier

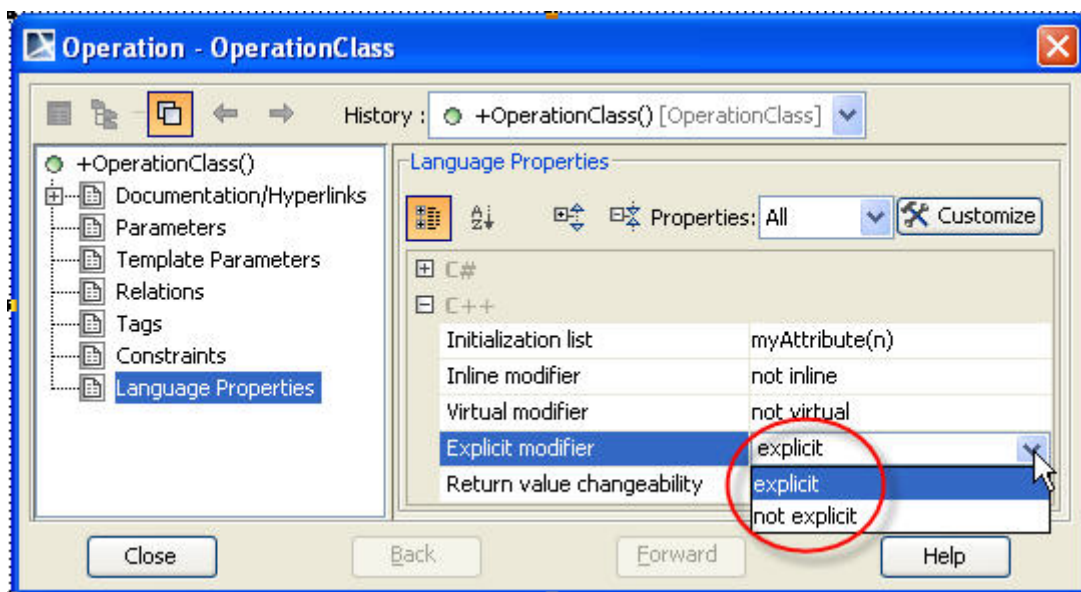
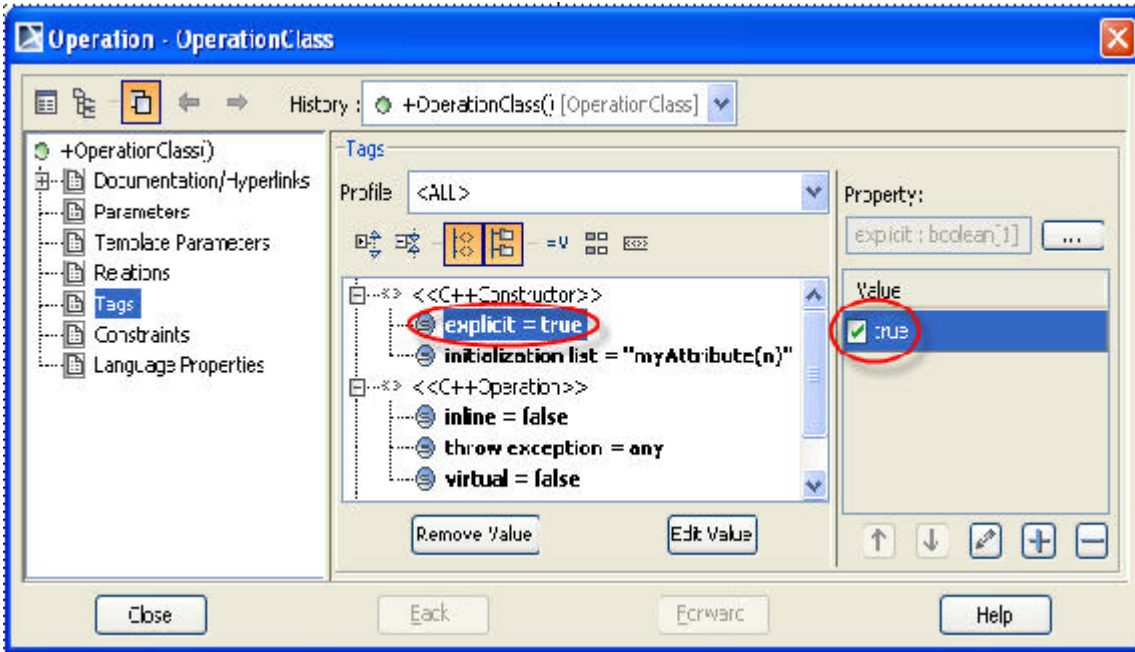


Figure 68 -- Operation Explicit modifier

Old value	Translation
not explicit	Apply the <<C++Constructor>> stereotype and set <b>explicit</b> tag value to <b>false</b> .
explicit	Apply the <<C++Constructor>> stereotype and set <b>explicit</b> tag value to <b>true</b> .



### Operation - Return value changeability

This Example is **ReturnValueChangeabilityClass** class that has three operations and all operations have set **return value changeability** in Language Properties to **const** as below.

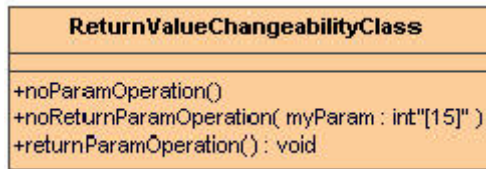


Figure 69 -- Return value changeability Example in Class Diagram

The Model that is being shown in the figure below is a translation.

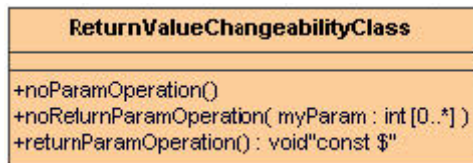


Figure 70 -- Translated Return value changeability in Class Diagram

The Return value changeability Language Properties is being shown in the figure below.

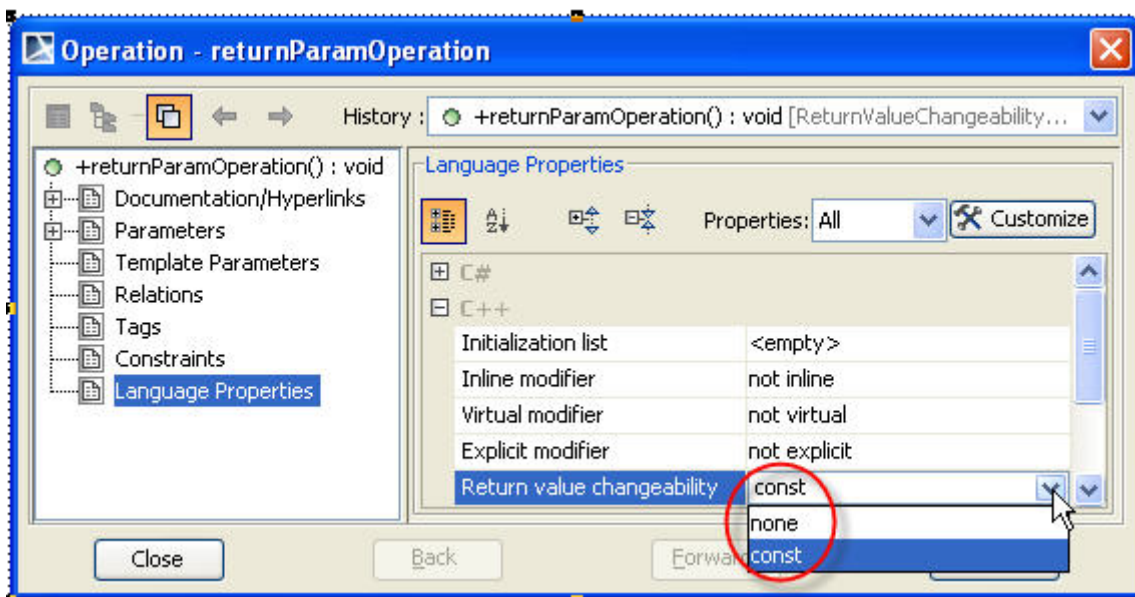
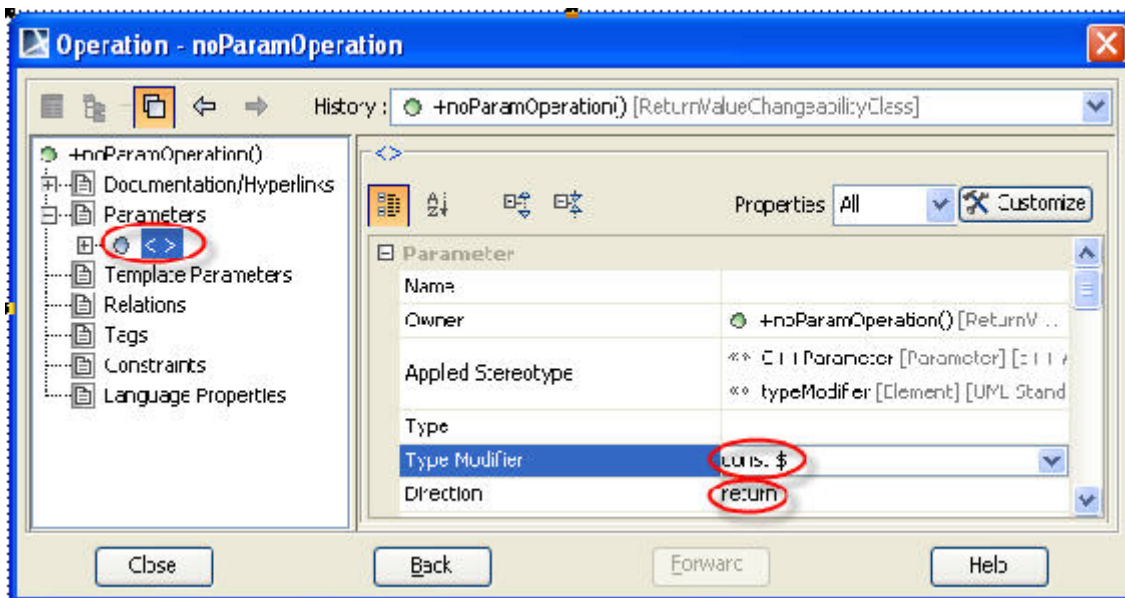


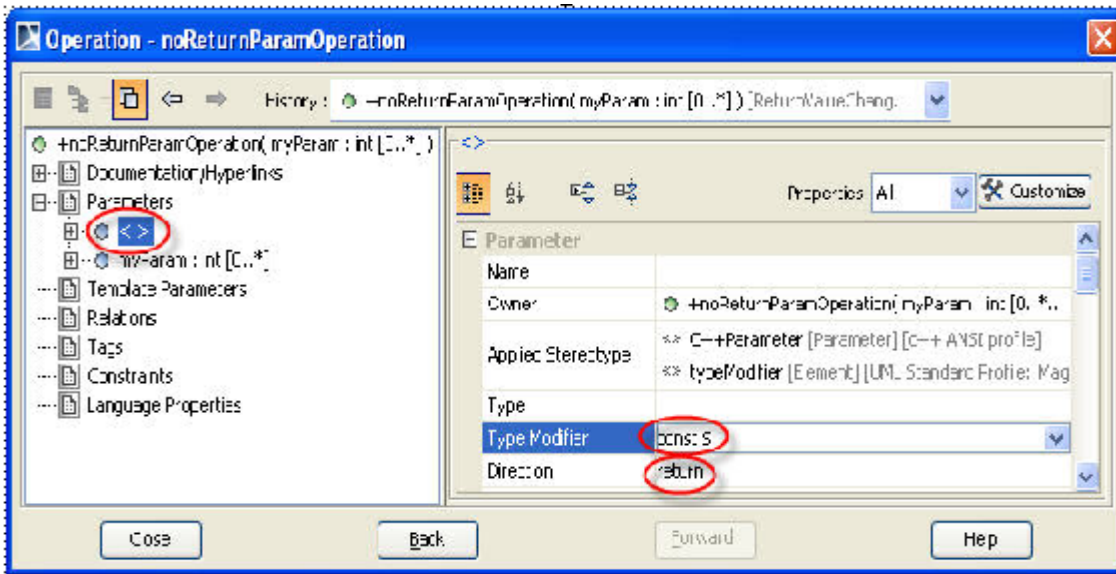
Figure 71 -- Operation Return value changeability

Old value	Translation
none	no change.
<b>const</b> , operation doesn't has parameter.	Create one return type parameter and set <b>Type Modifier to const \$</b> .



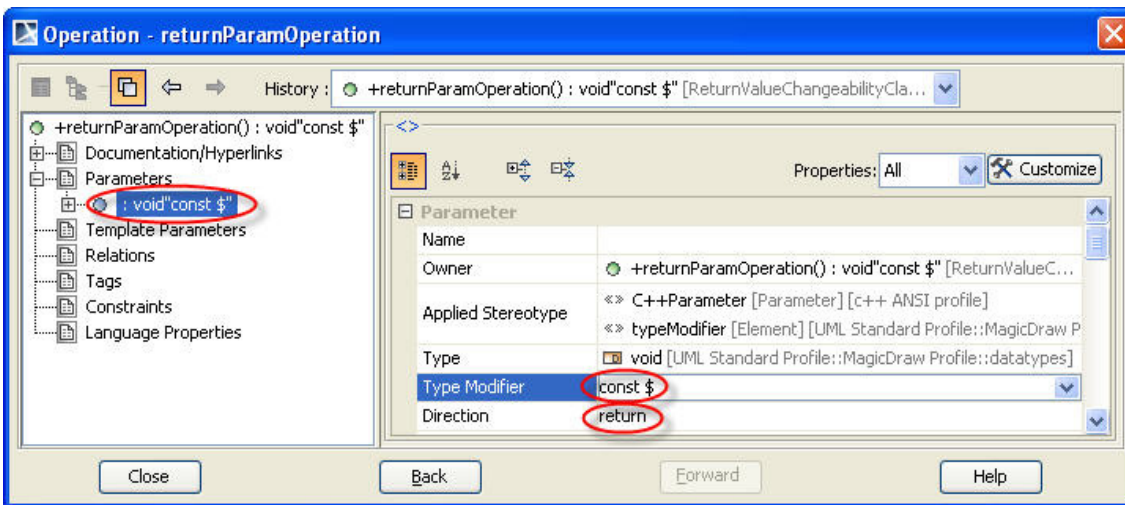
<b>const</b> , operation has parameter but does not have return type parameter.	Create one return type parameter and set <b>Type Modifier to const \$</b> .
---	---





**const**, operation has return type parameter.

Set **Type Modifier** in return type parameter to **const \$**.



### Attribute

This example is **AttributeClass** class that has attribute named **myAttribute**, return type is **int** and type modifier is **[15]**.

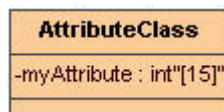


Figure 72 -- Attribute Example in Class Diagram

The Model that is being shown in the figure below is translation.



Figure 73 -- Translated Attribute in Class Diagram

There are **Mutable**, **Bit field**, **Abbreviated Initializer** and **Container** in Attribute Language Properties that need to be translated and apply the <<C++Attribute>> stereotype. A **Volatile** in Attribute Language Properties will move to **Type Modifier**.

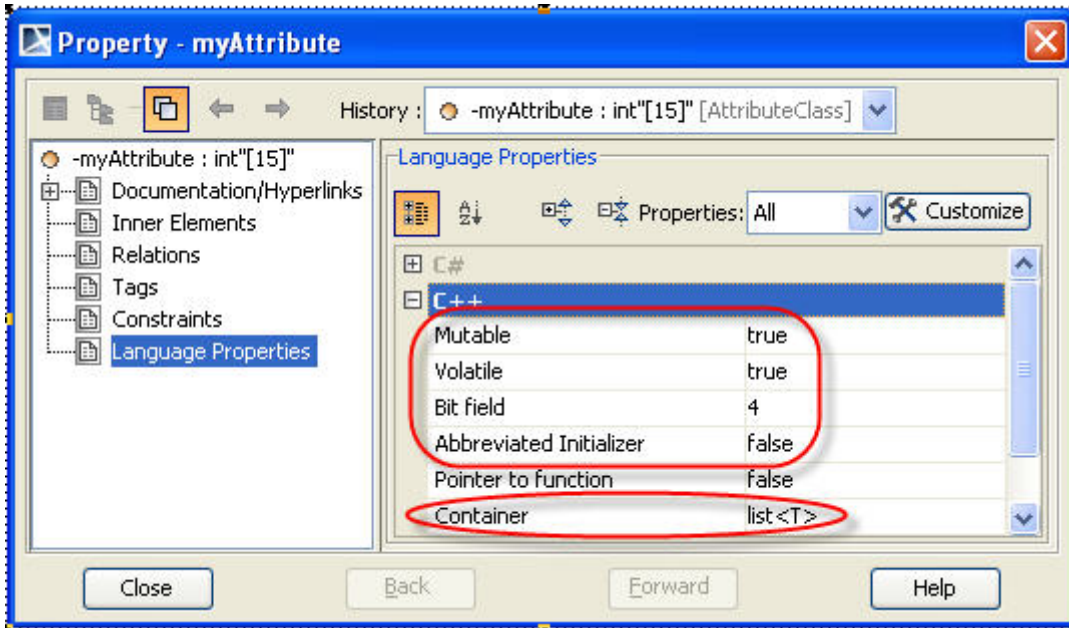


Figure 74 -- Attribute Language Properties

Attribute - Mutable

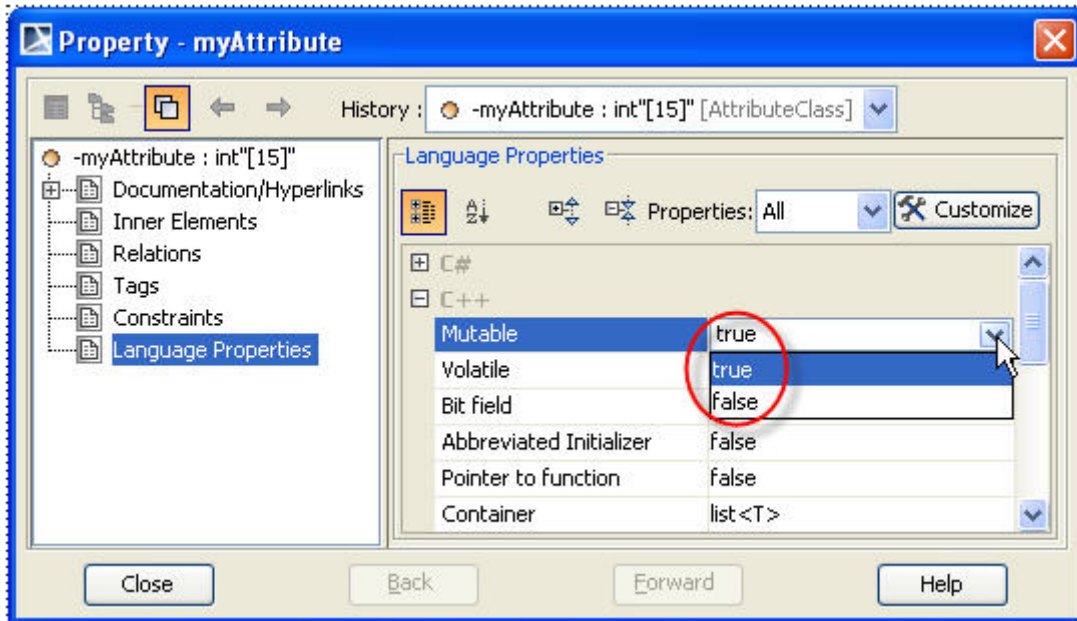
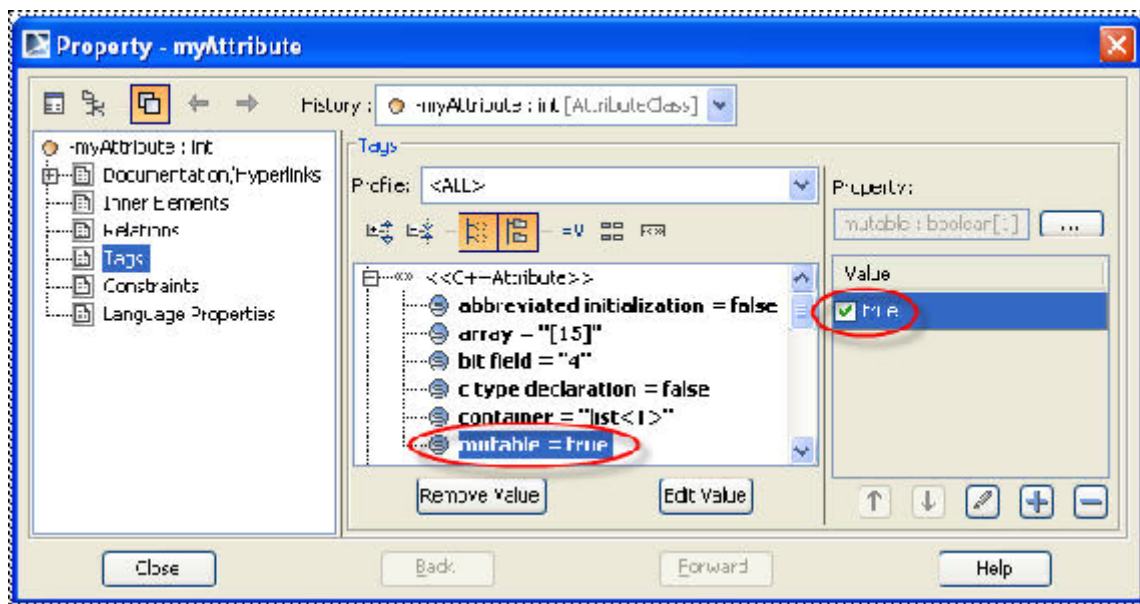


Figure 75 -- Attribute Mutable

Old value	Translation
false	Apply the <<C++Attribute>> stereotype and set <b>mutable</b> tag value to <b>false</b> .
true	Apply the <<C++Attribute>> stereotype and set <b>mutable</b> tag value to <b>true</b> .



Attribute - Volatile

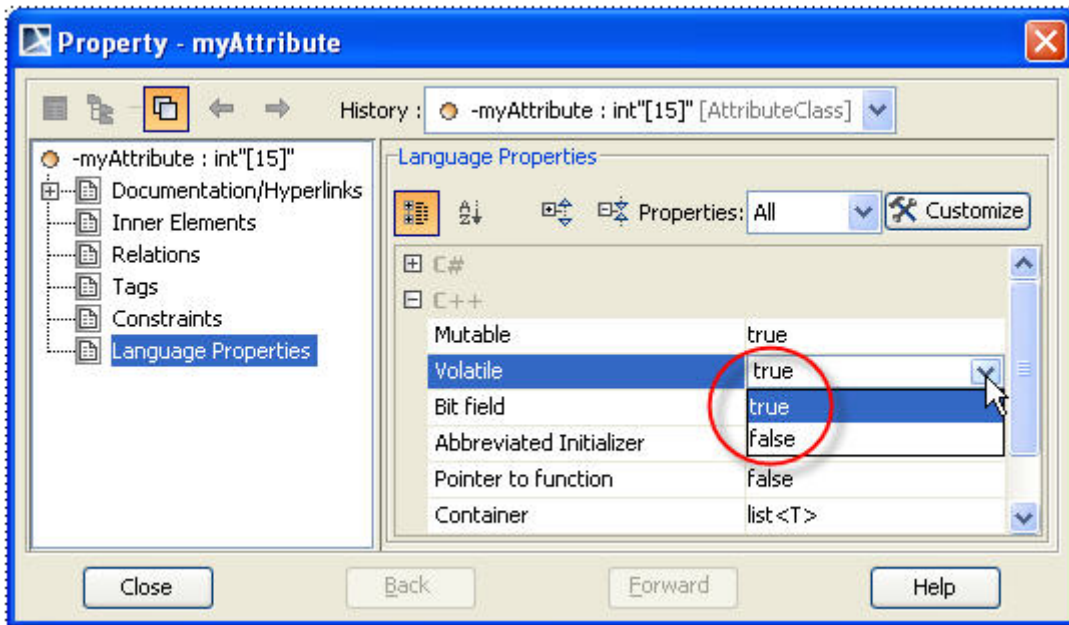
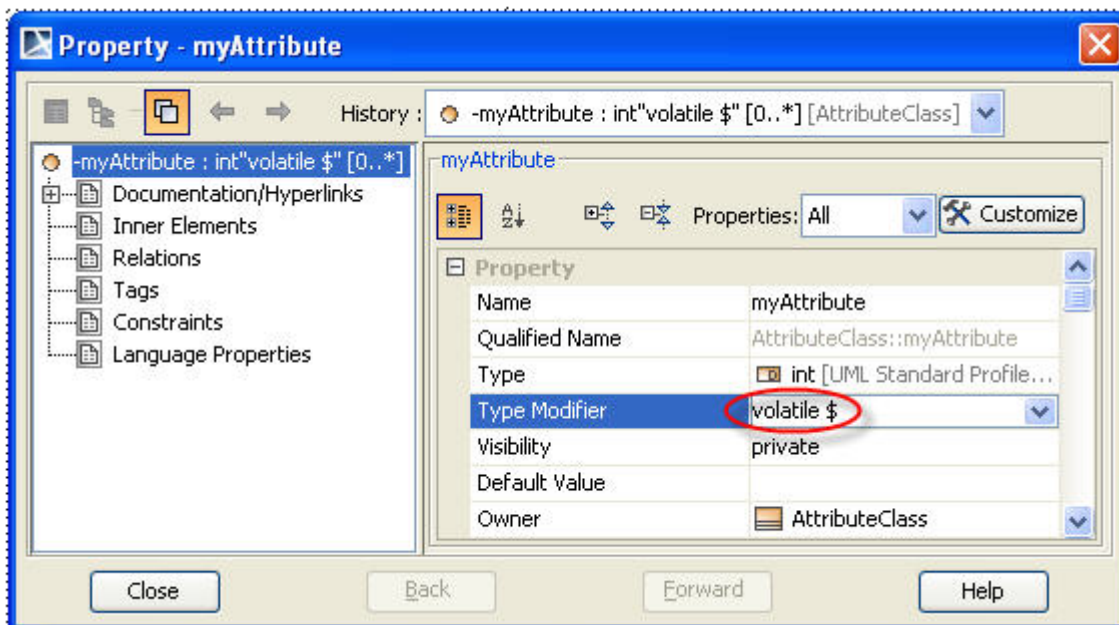


Figure 76 -- Attribute Volatile

Old value	Translation
false	no change.
true	Set Type Modifier to <b>volatile \$</b> .





Attribute -Bit field

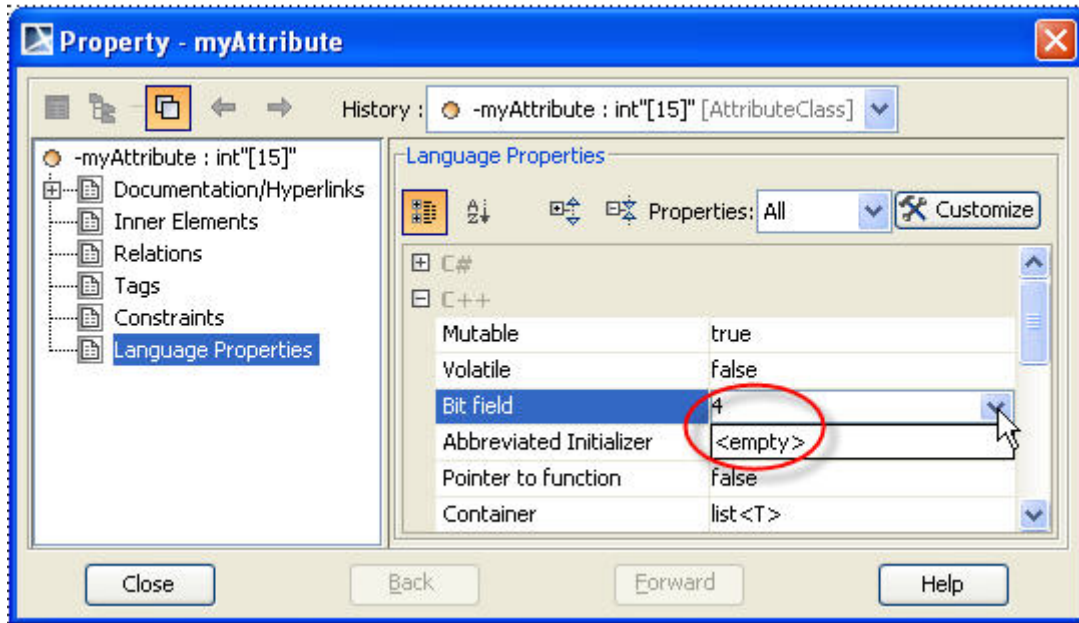
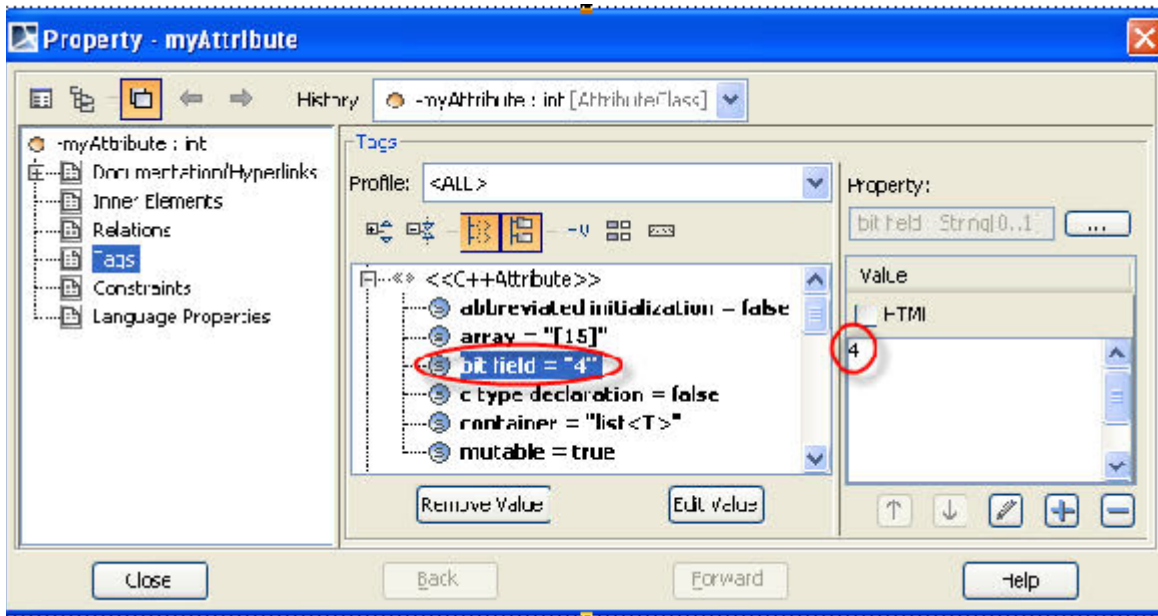


Figure 77 -- Attribute Bit field

Old value	Translation
<empty>	no change.
4	Apply the <<C++Attribute>> stereotype and set <b>bit field</b> tag value to 4.



Attribute -Abbreviated initializer

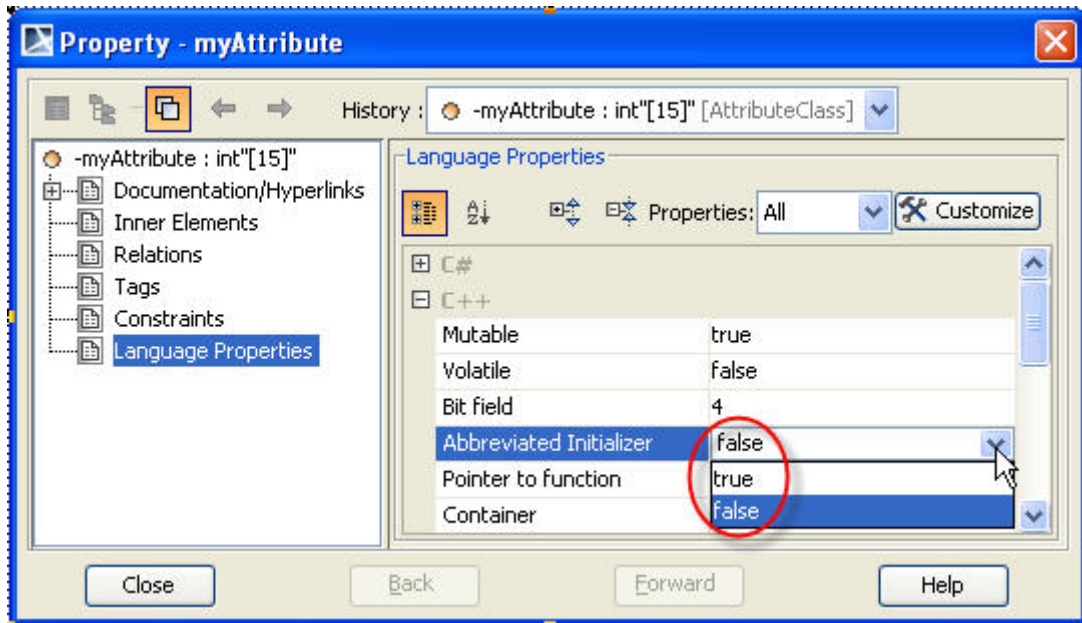
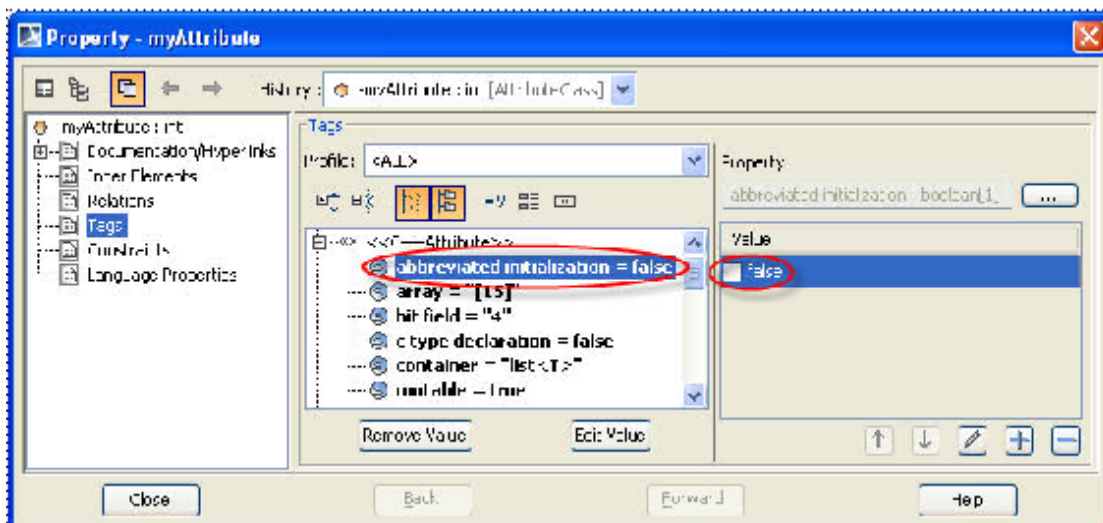


Figure 78 -- Attribute Abbreviated Initializer

Old value	Translation
false	Apply the <<C++Attribute>> stereotype and set <b>abbreviated initialization</b> tag value to <b>false</b> .
true	Apply the <<C++Attribute>> stereotype and set <b>abbreviated initialization</b> tag value to <b>true</b> .



Attribute - Pointer to function

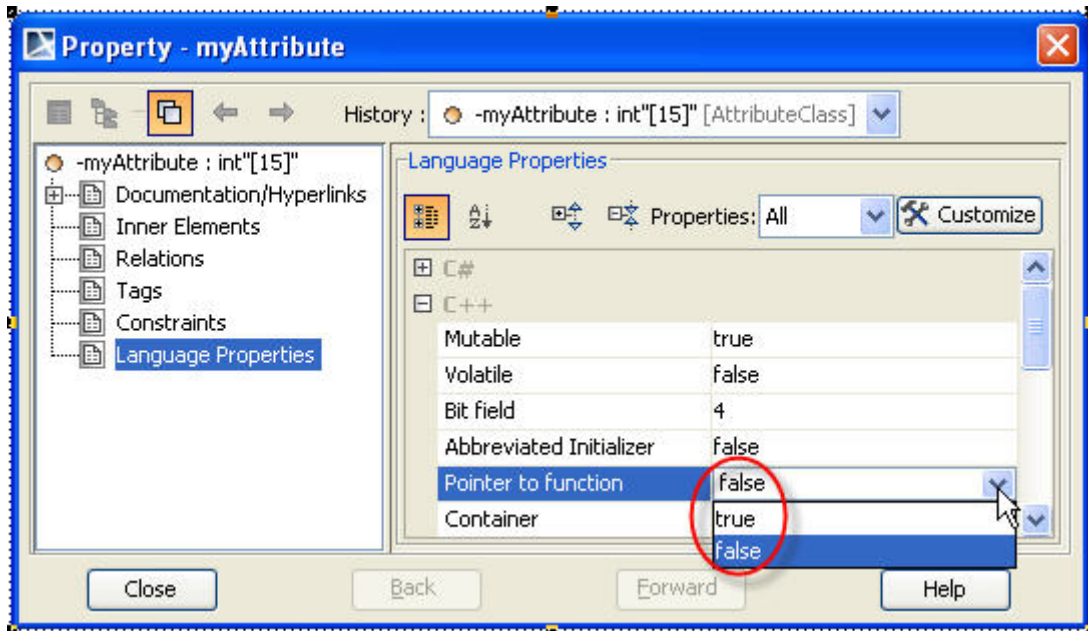


Figure 79 -- Attribute Pointer to function

Old value	Translation
false	no change.
true	no change.

## Attribute - Container

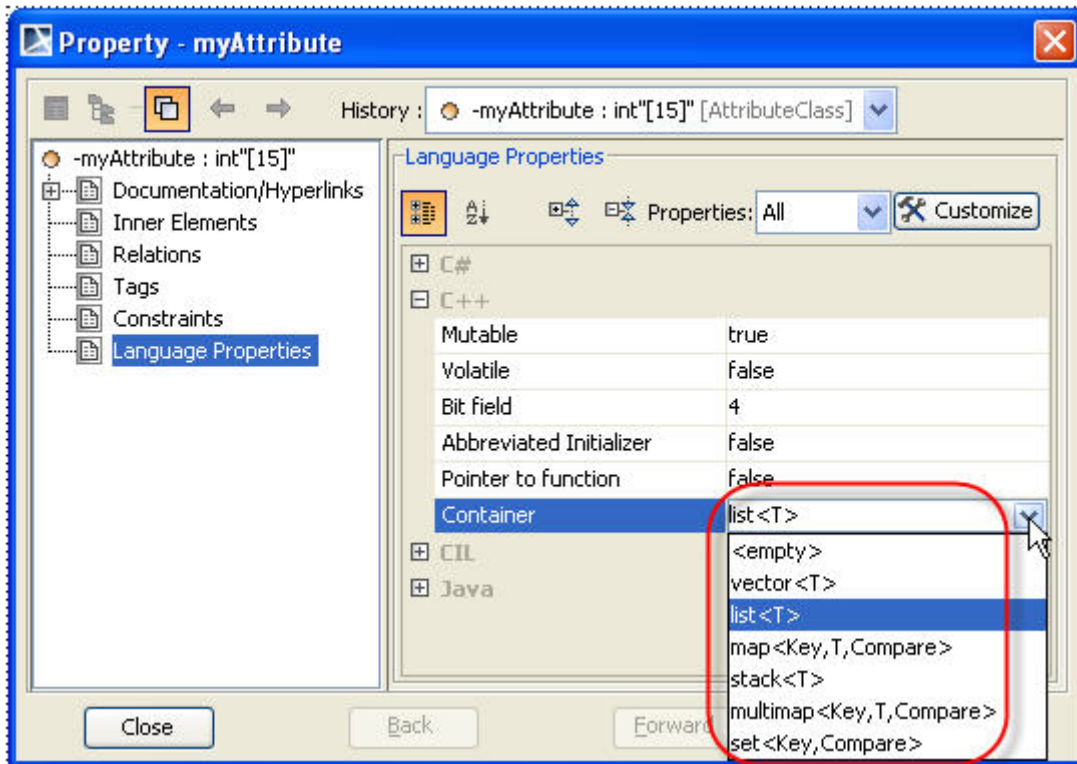
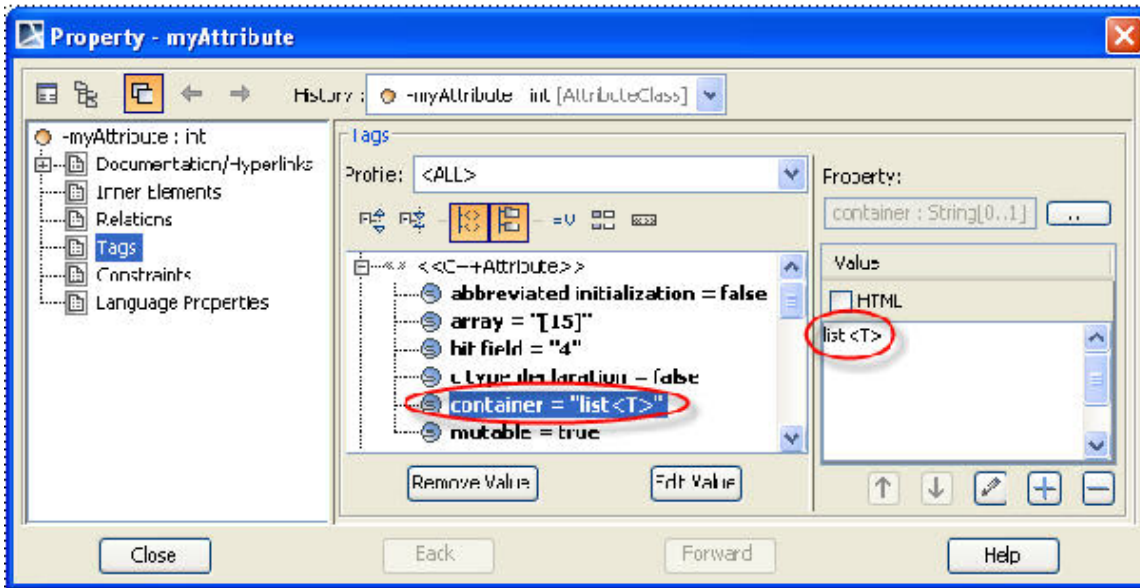


Figure 80 -- Attribute Container

Old value	Translation
<empty>	no change.
vector<T>	Apply the <<C++Attribute>> stereotype and set <b>container</b> tag value to <b>vector&lt;T&gt;</b> .
list<T>	Apply the <<C++Attribute>> stereotype and set <b>container</b> tag value to <b>list&lt;T&gt;</b> .
map<Key, T, Compare>	Apply the <<C++Attribute>> stereotype and set <b>container</b> tag value to <b>map&lt;Key, T, Compare&gt;</b> .
stack<T>	Apply the <<C++Attribute>> stereotype and set <b>container</b> tag value to <b>stack&lt;T&gt;</b> .
multimap<Key, T, Compare>	Apply the <<C++Attribute>> stereotype and set <b>container</b> tag value to <b>multimap&lt;Key, T, Compare&gt;</b> .
set<Key, Compare>	Apply the <<C++Attribute>> stereotype and set <b>container</b> tag value to <b>set&lt;Key, Compare&gt;</b> .



## Generalization

These examples are **ParentClass** class and **childClass** class that extends from **ParentClass** class.

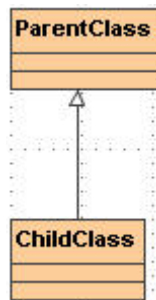


Figure 81 -- Generalization Example in Class Diagram

There are **Inheritance type** and **Virtual modifier** in Generalization Language Properties that need to be translated and apply the **<<C++Generalization>>** stereotype.

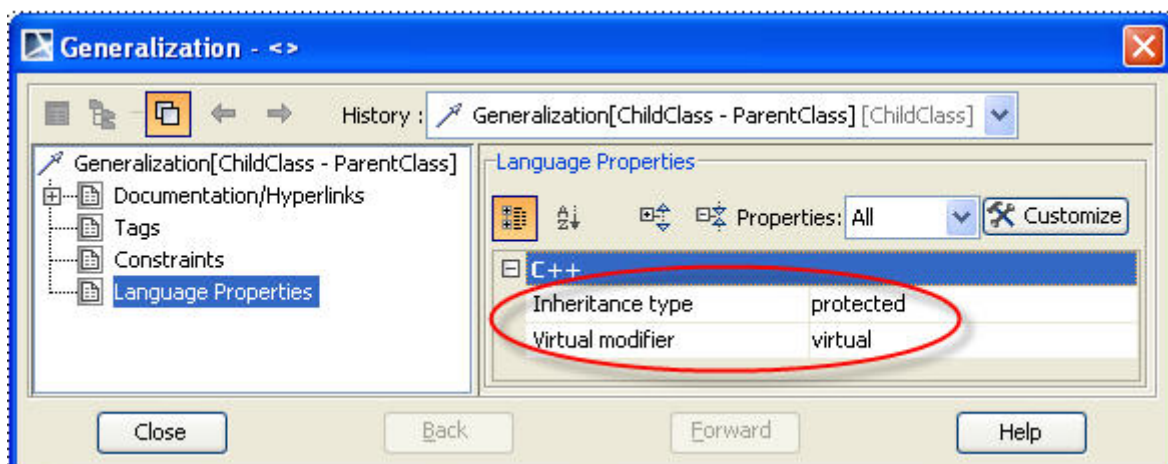


Figure 82 -- Generalization Language Properties



Generalization - Inheritance type

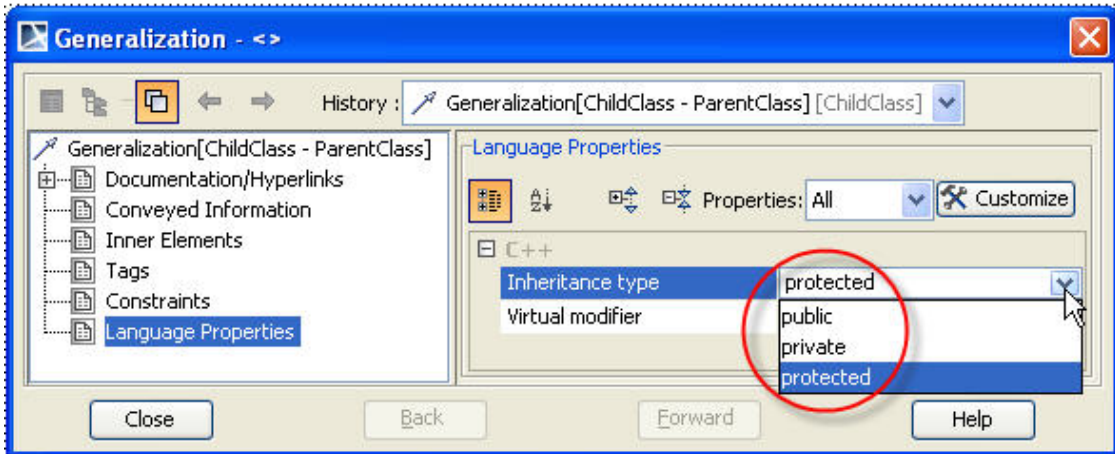
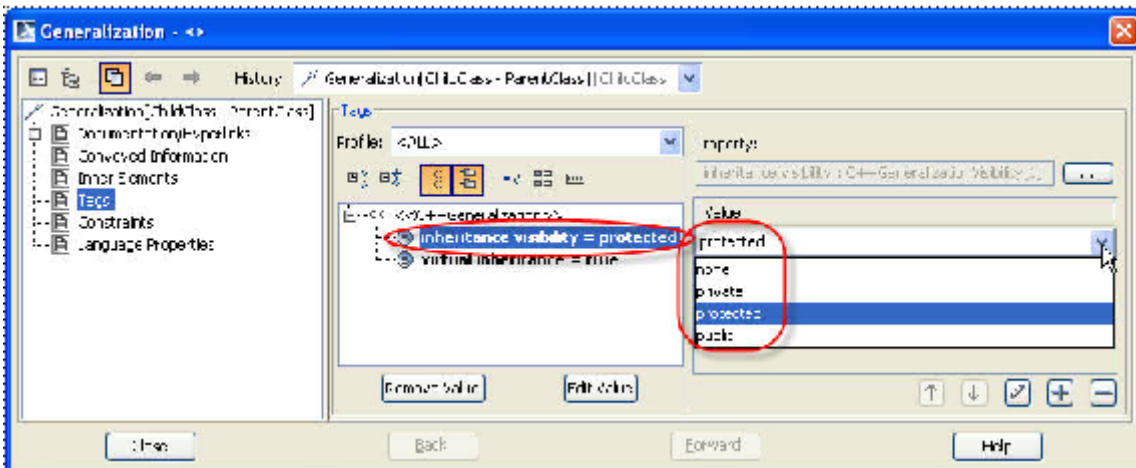


Figure 83 -- Generalization Inheritance type

Old value	Translation
public	Apply the <<C++Generalization>> stereotype and set <b>inheritance visibility</b> tag value to <b>public</b> .
protected	Apply the <<C++Generalization>> stereotype and set <b>inheritance visibility</b> tag value to <b>protected</b> .
private	Apply the <<C++Generalization>> stereotype and set <b>inheritance visibility</b> tag value to <b>private</b> .



## Generalization - Virtual modifier

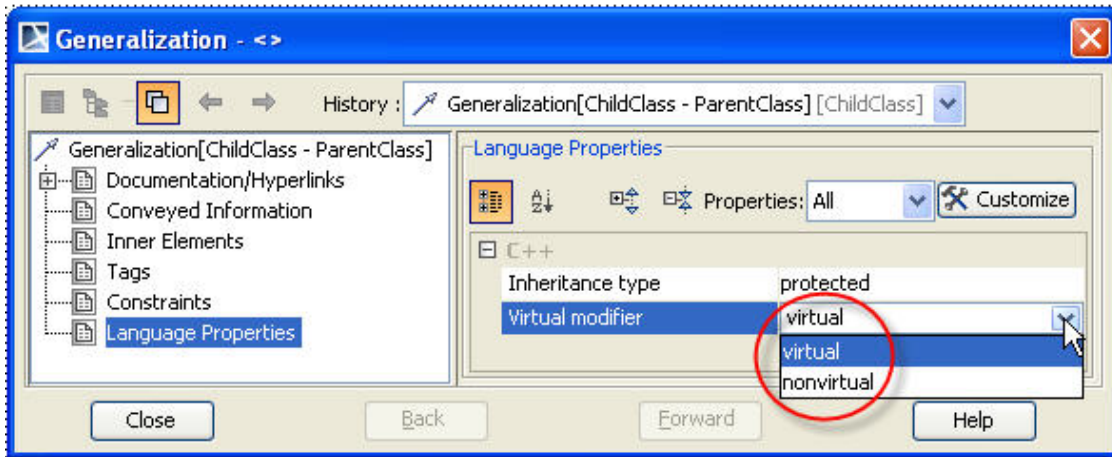
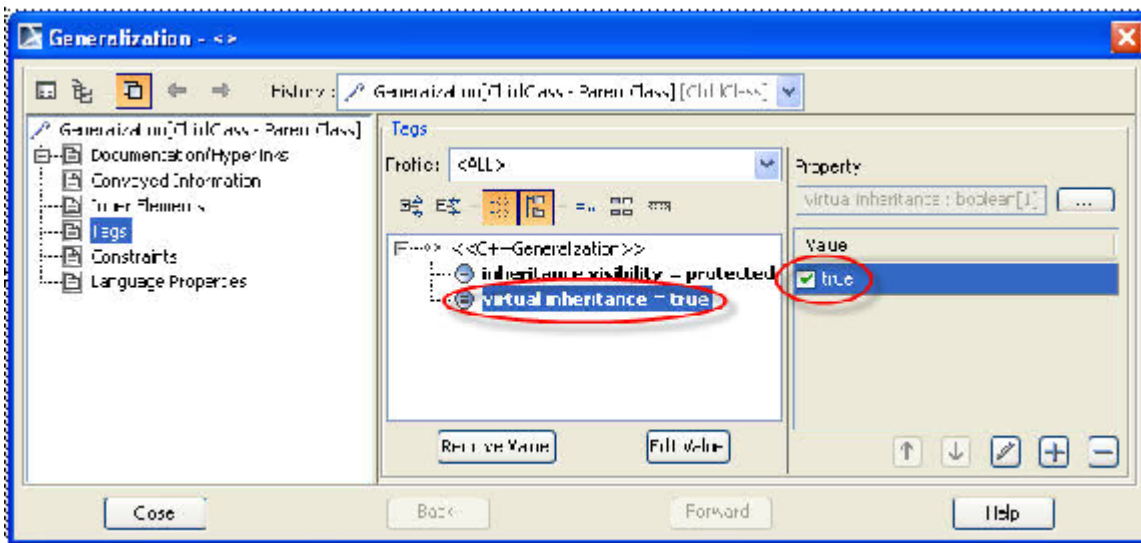


Figure 84 -- Generalization Virtual modifier

Old value	Translation
nonvirtual	Apply the <<C++Generalization>> stereotype and set <b>virtual inheritance</b> tag value to <b>false</b> .
virtual	Apply the <<C++Generalization>> stereotype and set <b>virtual inheritance</b> tag value to <b>true</b> .



## Type Modifiers

In version 12, type modifiers use the \$ character to specify the type modifiers construct. This allows mapping of complex type modifiers. Eg. `const int* const` is mapped to `const $* const`.

### Array

The Attribute Type Modifier is being shown in the figure below.

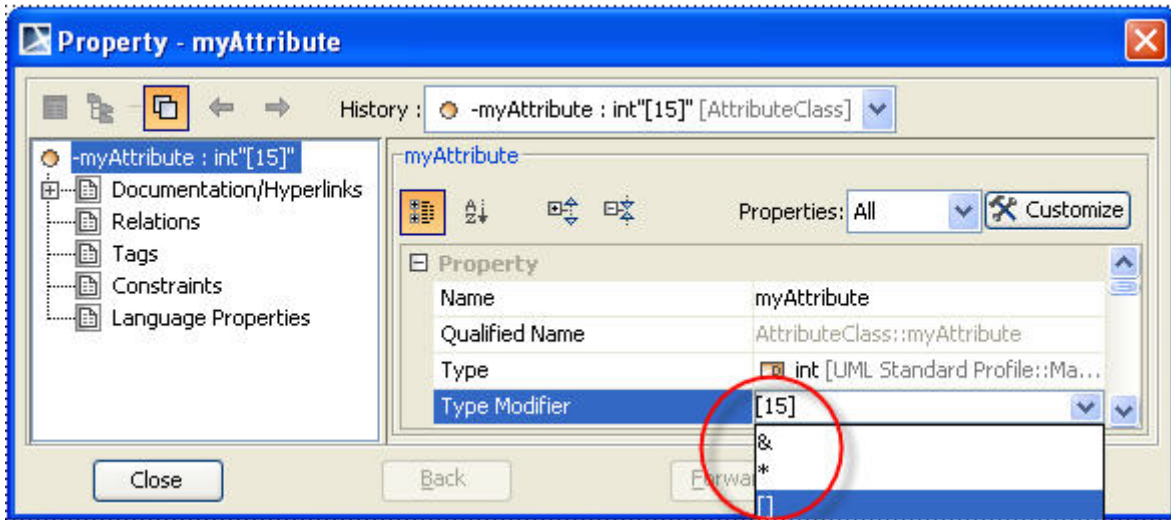


Figure 85 -- Array type modifier in Attribute



The Parameter Type Modifier is being shown in the figure below.

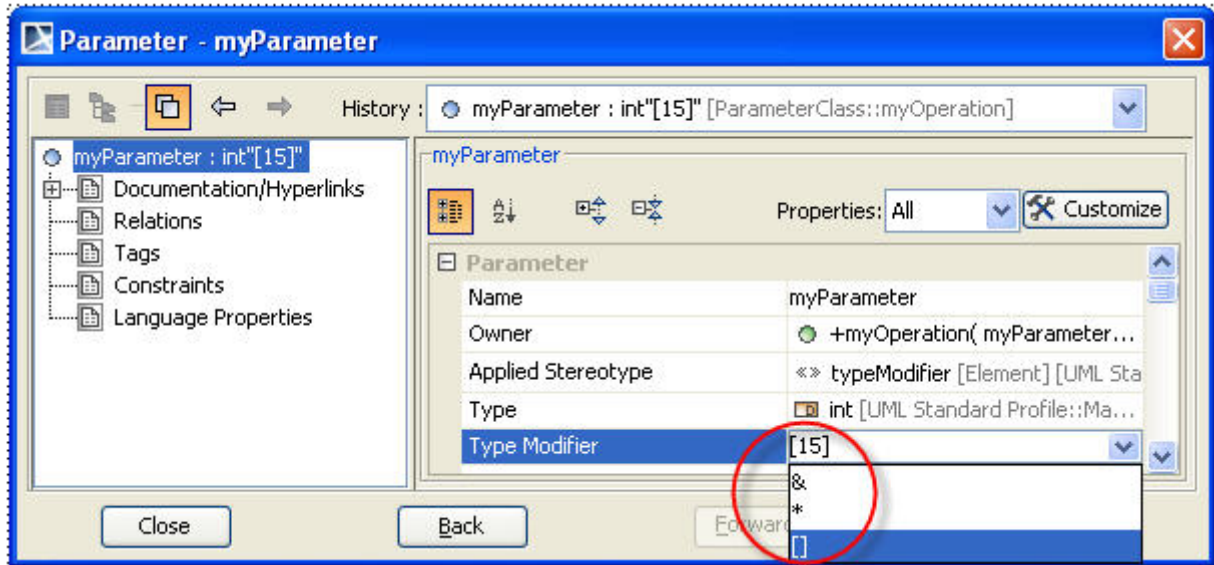
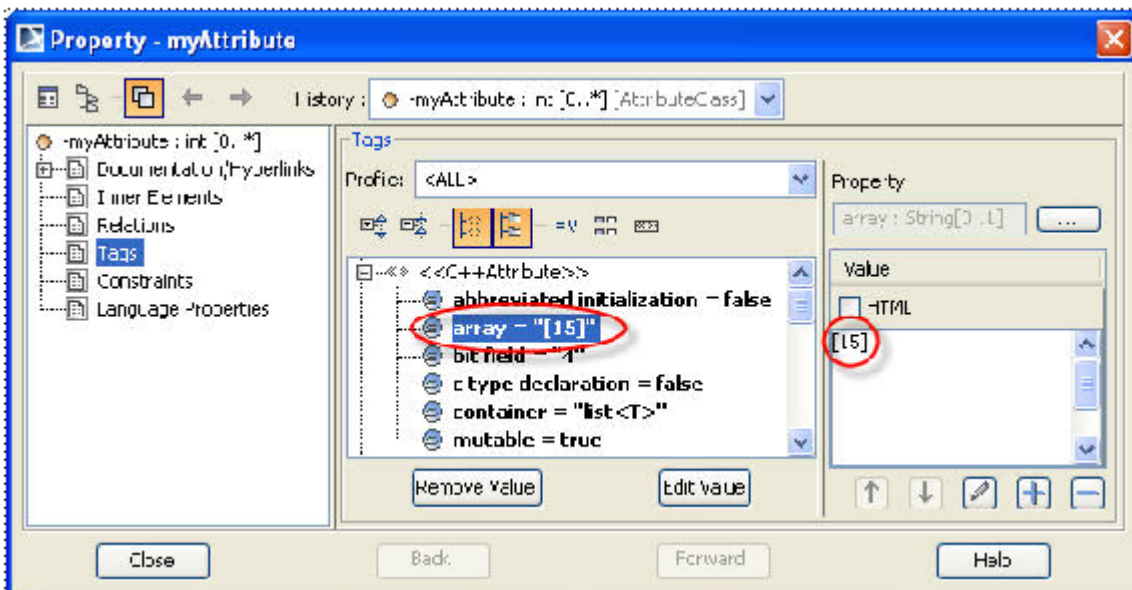
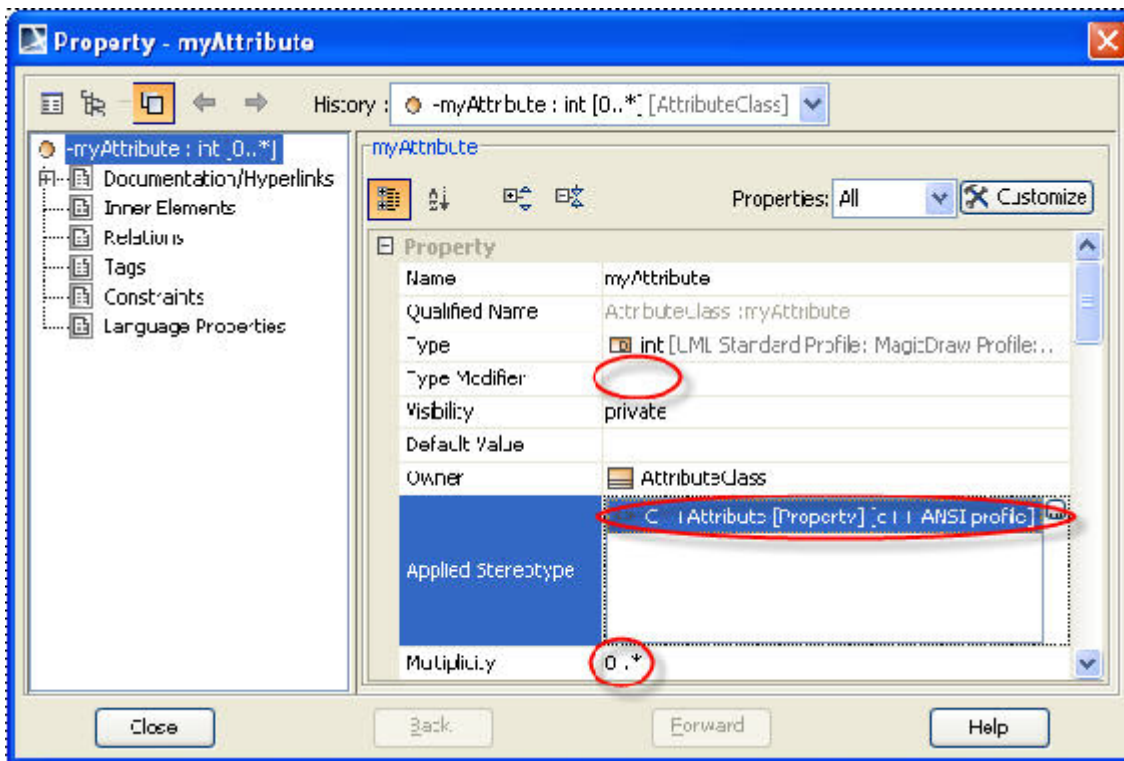


Figure 86 -- Array type modifier in Parameter

Old value	Translation
[15]	Apply the <<C++Attribute>> stereotype and set <b>array</b> tag value to [15]. Set <b>Multiplicity</b> field to 0..*. Remove [15] from <b>type modifier</b> field.

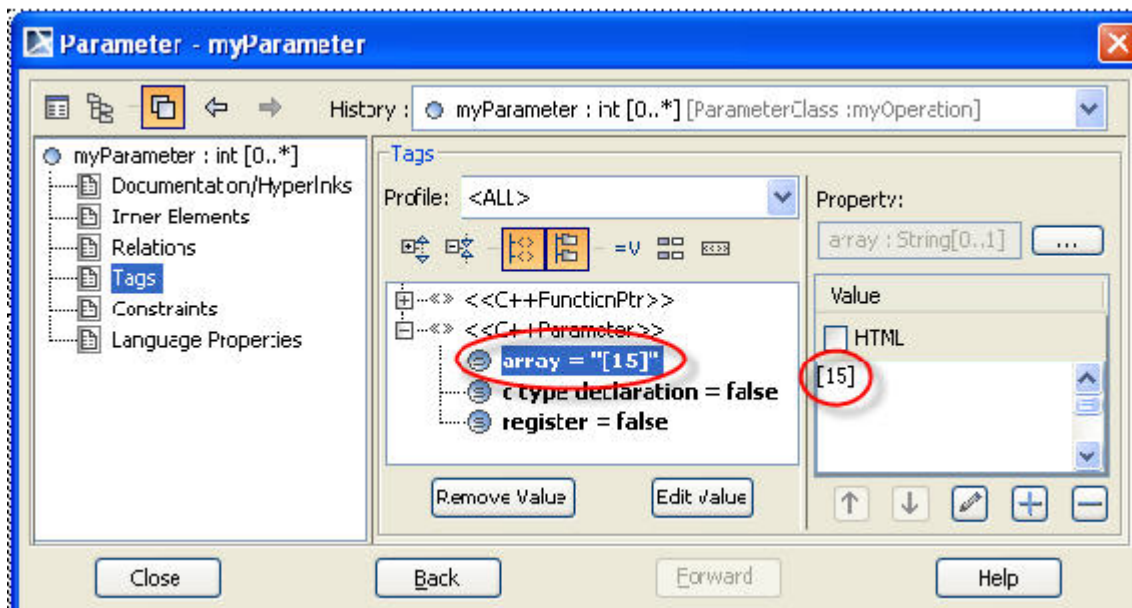
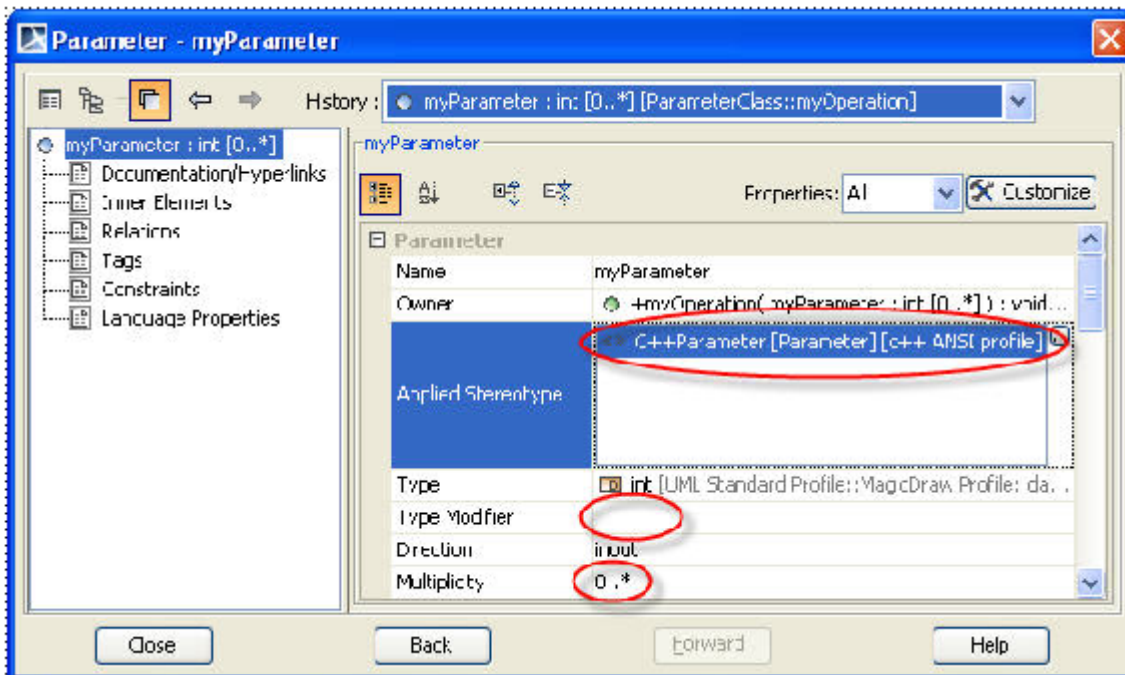


ParameterClass
+myOperation( myParameter : int"[15]" ) : void

ParameterClass
+myOperation( myParameter : int [0..*] ) : void

[15]

Apply the <<C++Parameter>> stereotype and set **array** tag value to [15].  
 Set **Multiplicity** field to 0..\*.  
 Remove [15] from **type modifier** field.



## Stereotypes

### <<C++EnumerationLiteral>>

This example is **EnumClass** class that has literal value named **literal**. This literal value applies the **<<C++EnumerationLiteral>>** stereotype and sets **C++Initializer** tag value to **0**.

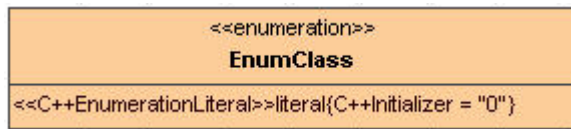


Figure 87 -- <<C++EnumerationLiteral>> stereotype Example in Class Diagram

The <<C++EnumerationLiteral>> stereotype is being shown in the figure below.

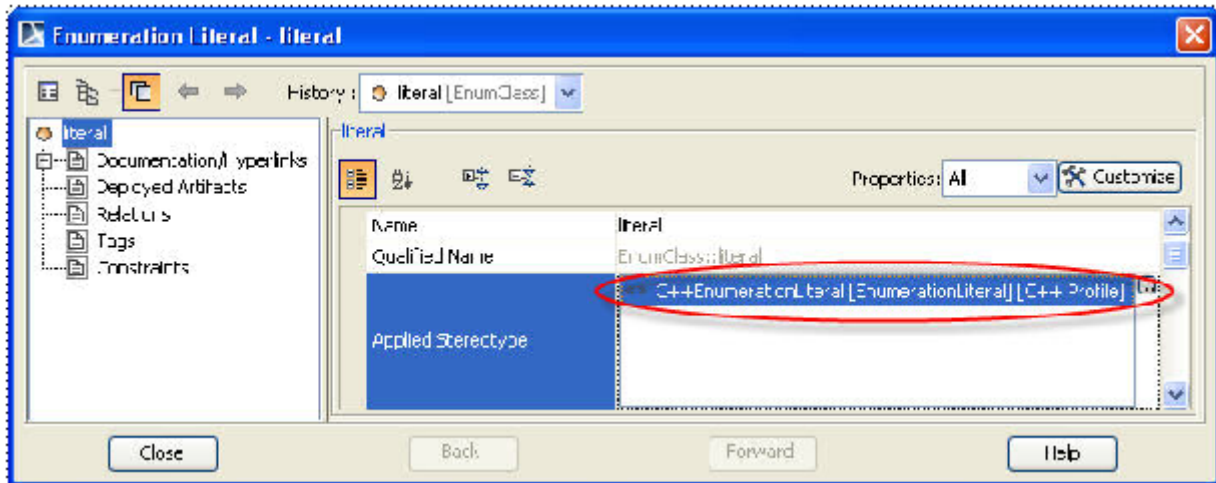


Figure 88 -- <<C++EnumerationLiteral>> stereotype

The C++Initializer tag value is being shown in the figure below.

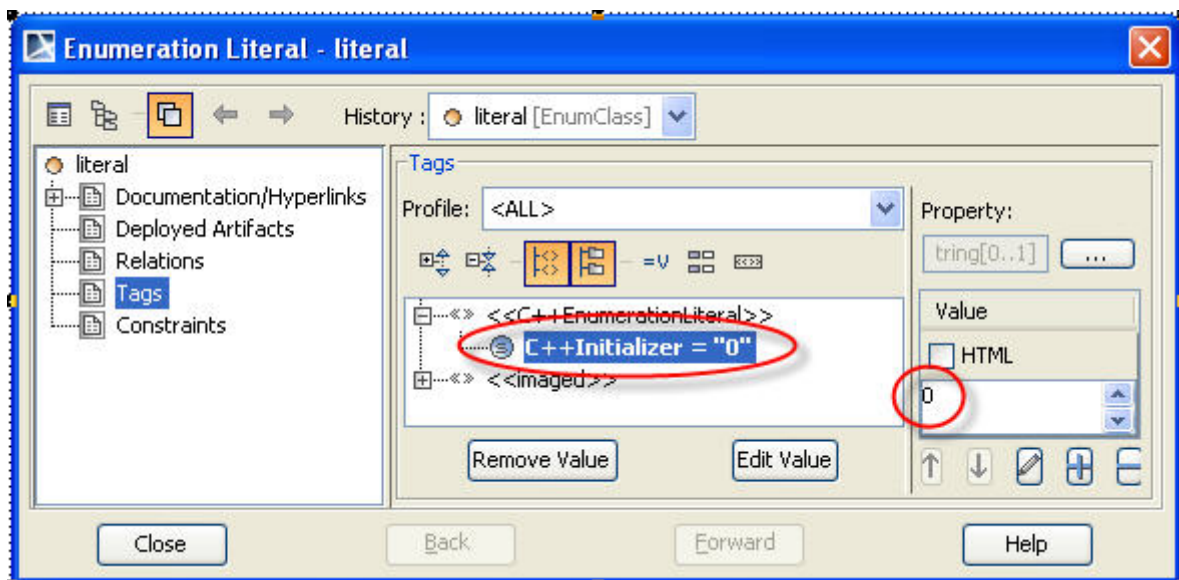
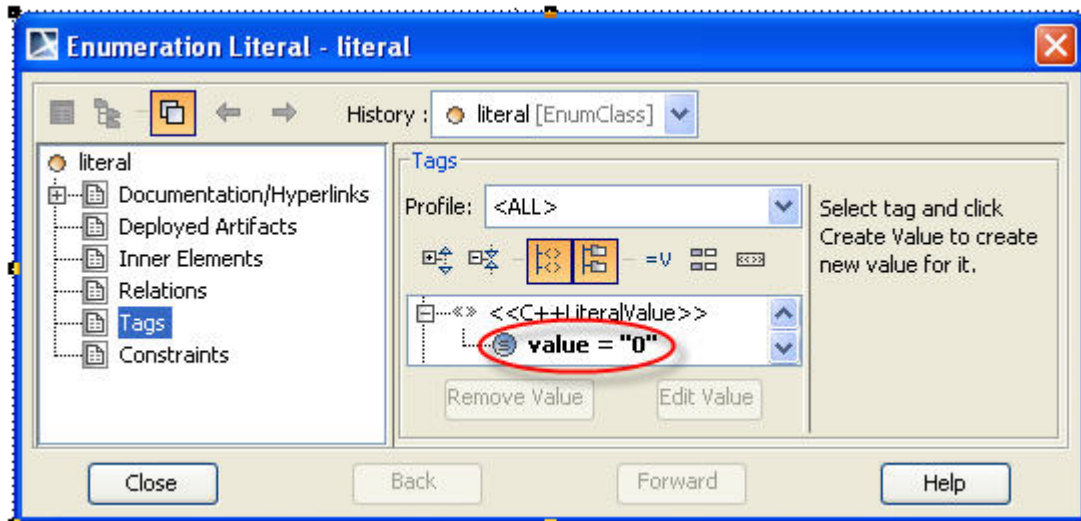
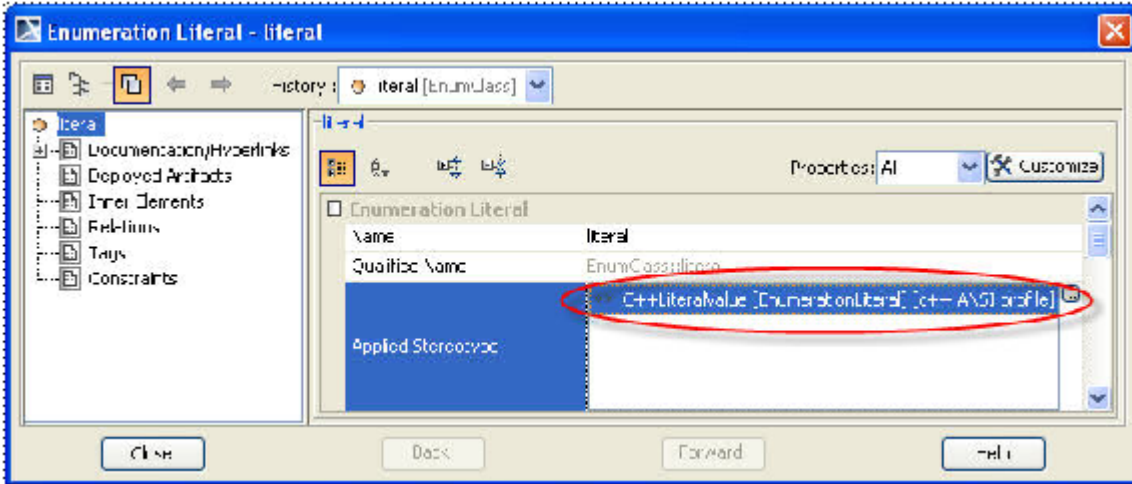


Figure 89 -- Enumeration Literal Tag Value

Old value	Translation
-----------	-------------



<p>Enumeration Literal applies the <code>&lt;&lt;C++EnumerationLiteral&gt;&gt;</code> stereotype and sets <b>C++Initializer</b> tag value to 0.</p>	<p>Apply the <code>&lt;&lt;C++LiteralValue&gt;&gt;</code> and set <b>value</b> tag value to 0. Remove the <code>&lt;&lt;C++EnumerationLiteral&gt;&gt;</code> stereotype and <b>C++Initializer</b> tag value from Enumeration Literal.</p>
---	---



The Model that is being shown in the figure below is a translation.

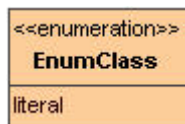


Figure 90 -- Translated Enumeration Literal in Class Diagram

## <<C++Namespace>>

This example is **MyPackage** package that applies the `<<C++Namespace>>` stereotype in old profile and sets **unique namespace name** tag value to **myNamespace**.

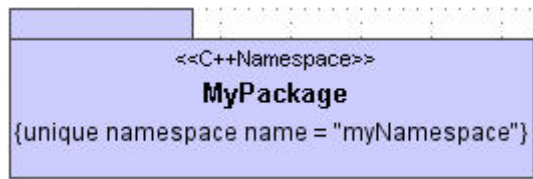


Figure 91 -- <<C++Namespace>> stereotype Example in Class Diagram

The <<C++Namespace>> stereotype is being shown in the figure below.

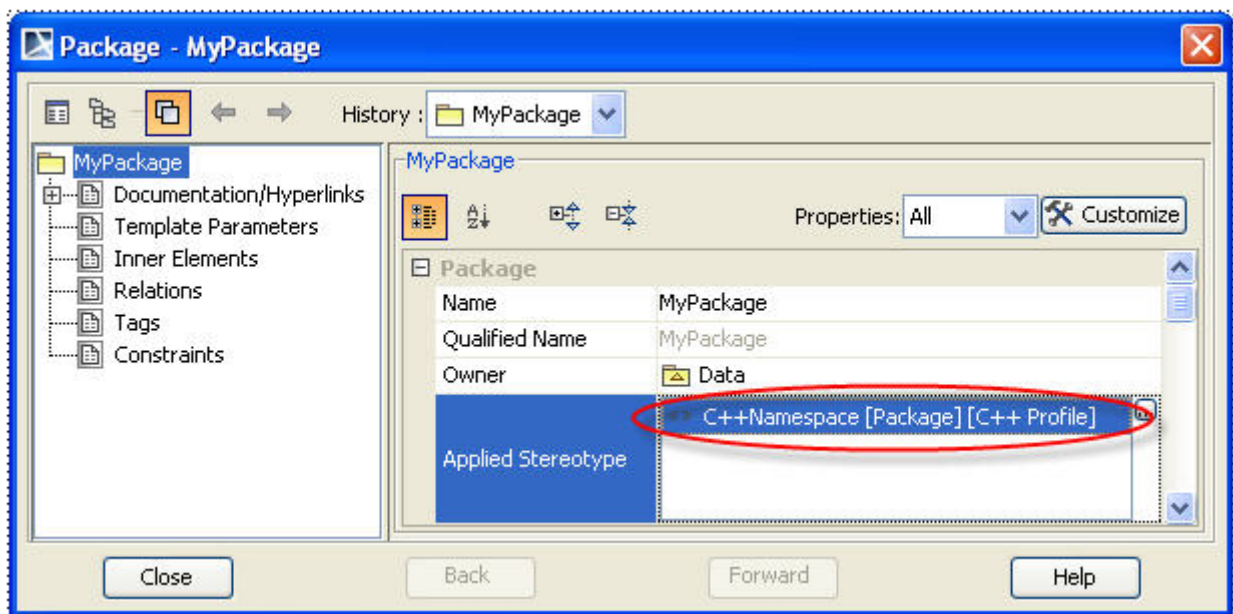


Figure 92 -- <<C++Namespace>> stereotype

The **unique namespace name** tag value is being shown in the figure below.

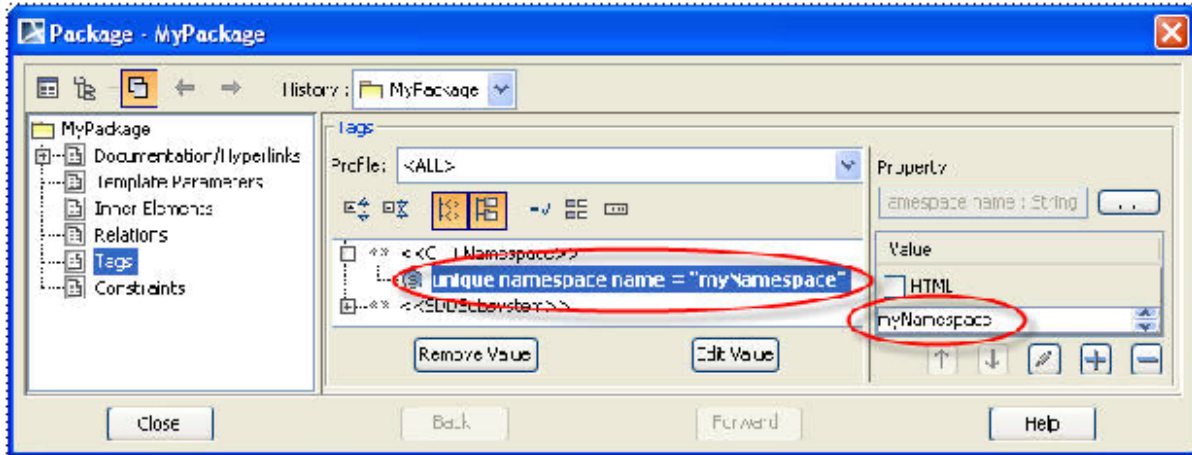
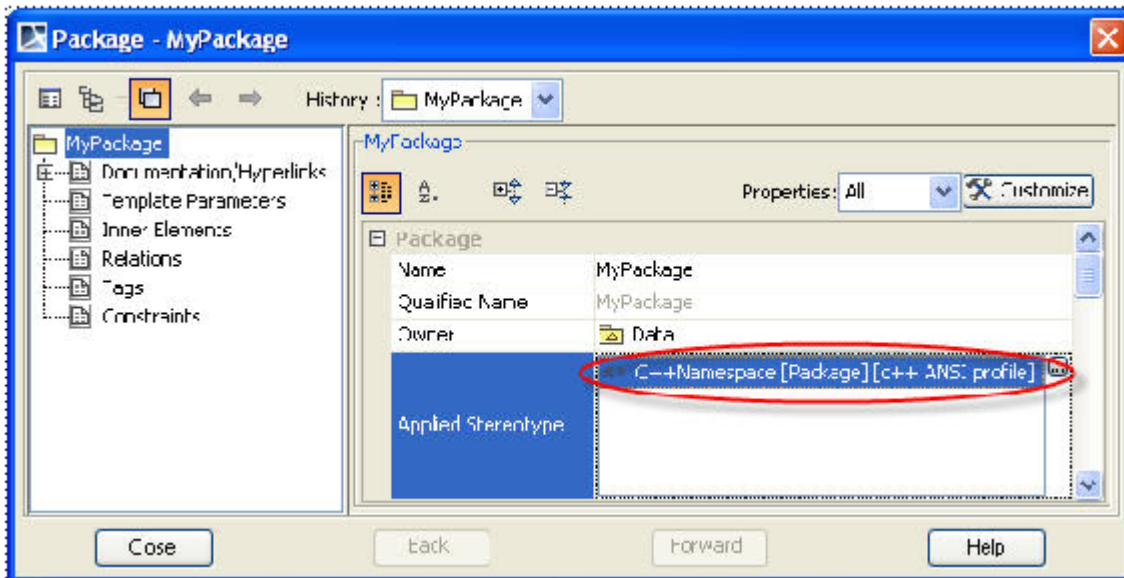
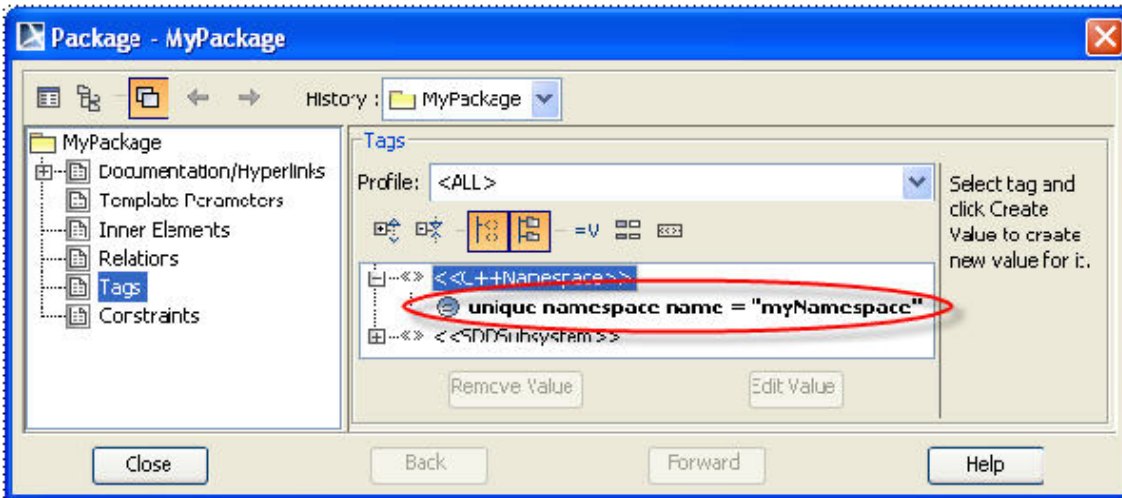


Figure 93 -- unique namespace name tag value

Old value	Translation
<p>Package applies the <b>&lt;&lt;C++Namespace&gt;&gt;</b> stereotype in old profile (C++ Profile) and sets <b>unique namespace name</b> tag value to <b>myNamespace</b>.</p>	<p>Apply the <b>&lt;&lt;C++Namespace&gt;&gt;</b> stereotype in new profile (c++ ANSI profile) and set <b>unique namespace name</b> tag value to <b>myNamespace</b>. Remove the <b>&lt;&lt;C++Namespace&gt;&gt;</b> stereotype (C++ Profile) and <b>unique namespace name</b> tag value.</p>





## <<C++Typename>>

The <<C++Typename>> stereotype can apply to Template Parameters. Therefore, Elements that have template parameters could apply this stereotype.

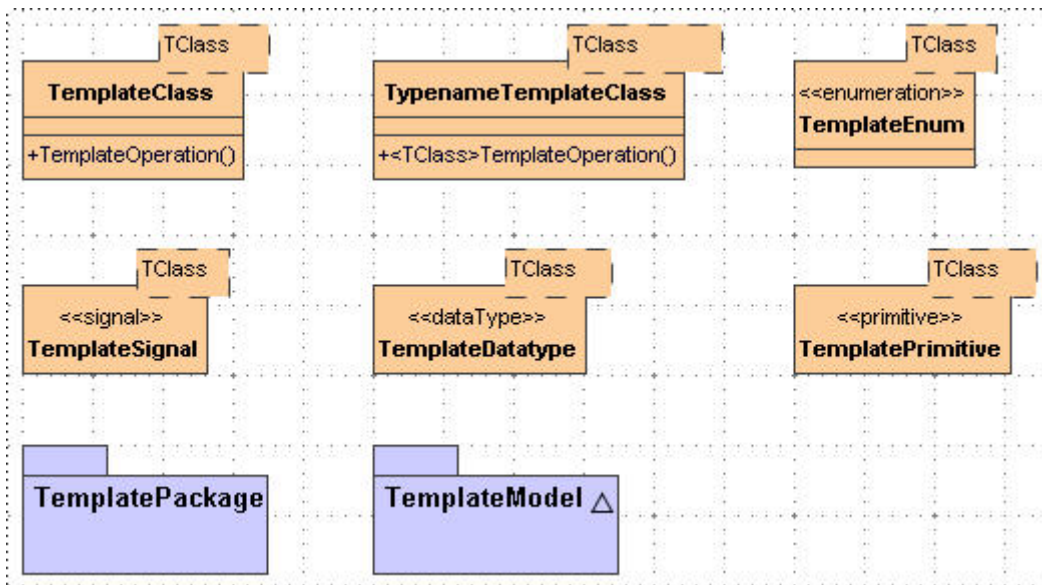


Figure 94 -- Elements that can have template parameters

In version 12.0, all elements that have template parameters must apply the <<C++TemplateParameter>> stereotype.

If elements from old version apply the <<C++Typename>> stereotype, the translation will apply the <<C++TemplateParameter>> stereotype and set **type keyword** tag value to **typename**. If not, the translation will apply the <<C++TemplateParameter>> stereotype and set **type keyword** tag value to **class**.



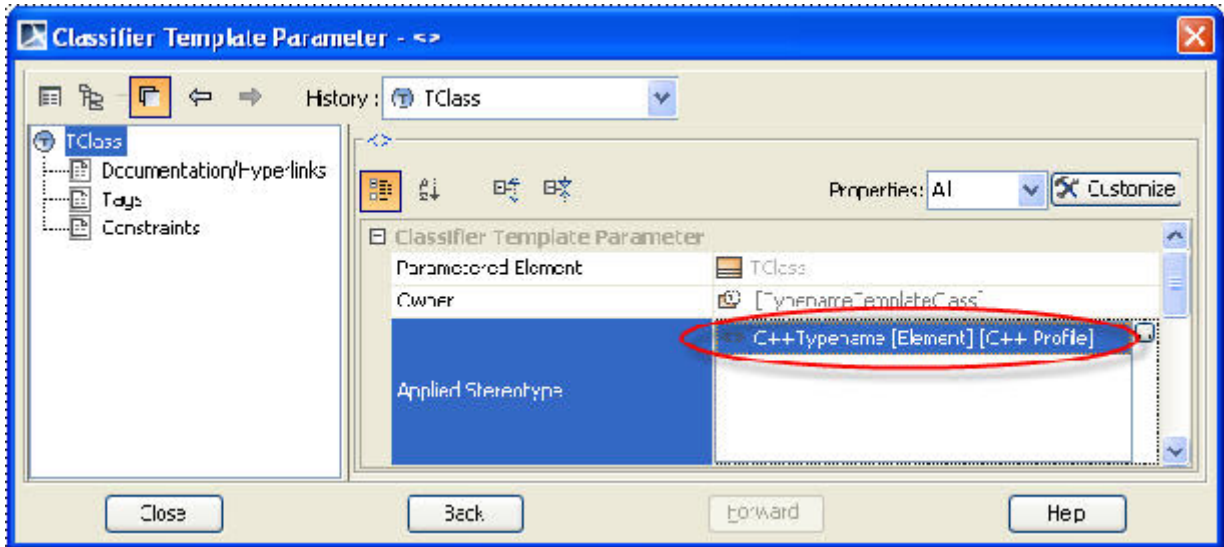


Figure 95 -- <<C++Typename>> stereotype

The <<C++TemplateParameter>> stereotype is being shown in the figure below.

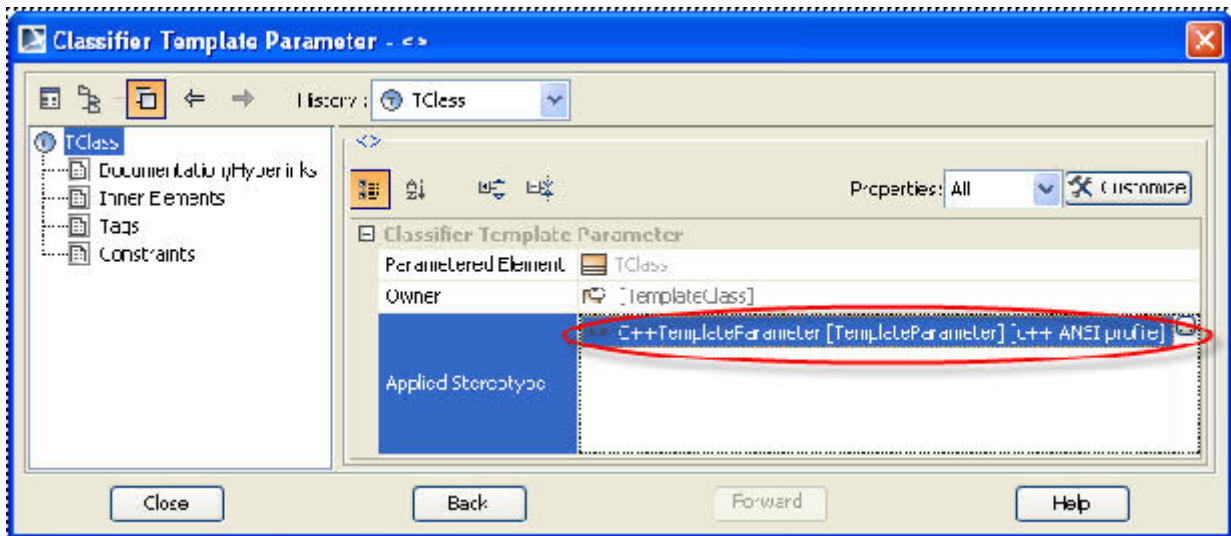
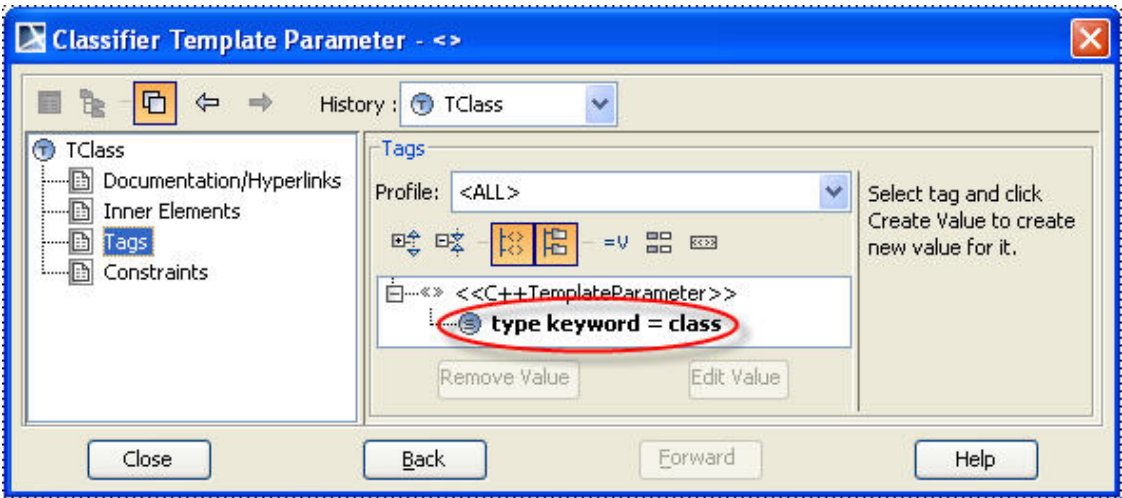


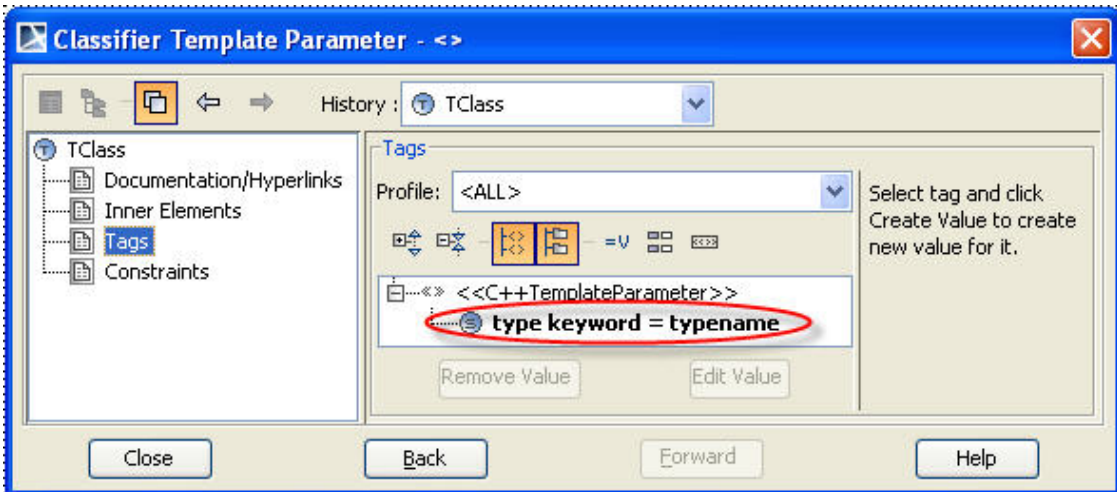
Figure 96 -- <<C++TemplateParameter>> stereotype

Old value	Translation
Template Parameter <b>does not</b> apply the <<C++Typename>> stereotype.	Apply the <<C++TemplateParameter>> stereotype and set <b>type keyword</b> tag value to <b>class</b> .



Template Parameter applies the <<C++Typename>> stereotype.

Apply the <<C++TemplateParameter>> stereotype and set **type keyword** tag value to **typename**.  
 Remove the <<C++Typename>> stereotype.



### <<constructor>> and <<destructor>> in UML Standard Profile

These examples are UMLStandardConstructorClass class that has an operation named myOperation() which applies the <<constructor>> stereotype in UML Standard Profile and UMLStandardDestructorClass class that has an operation named myOperation() which applies the <<destructor>> stereotype in UML Standard Profile.



Figure 97 -- <<constructor>> and <<destructor>> stereotype Example in Class Diagram

<<constructor>>

The <<constructor >> stereotype is being shown in the figure below.

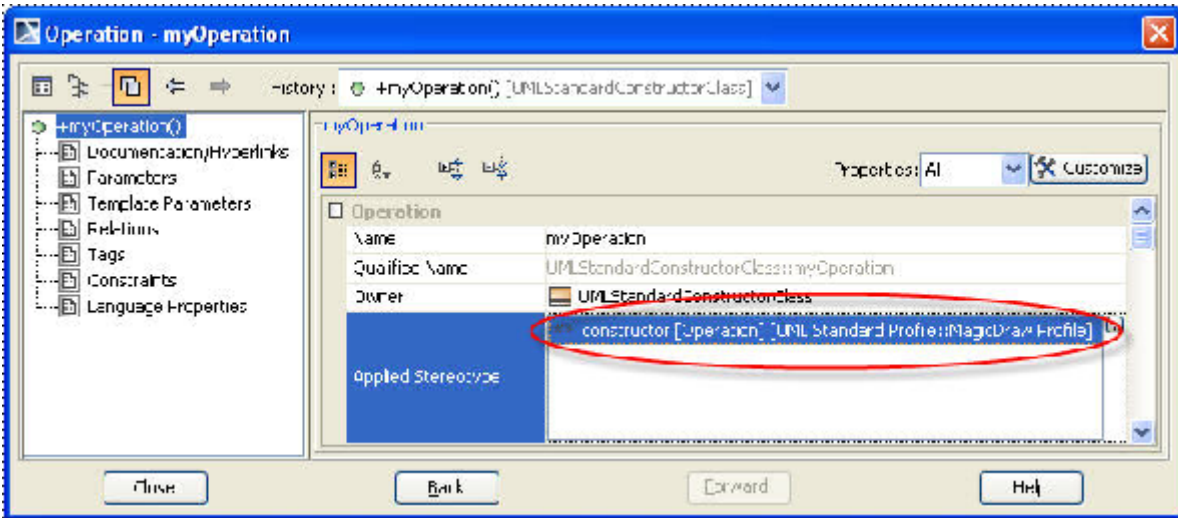
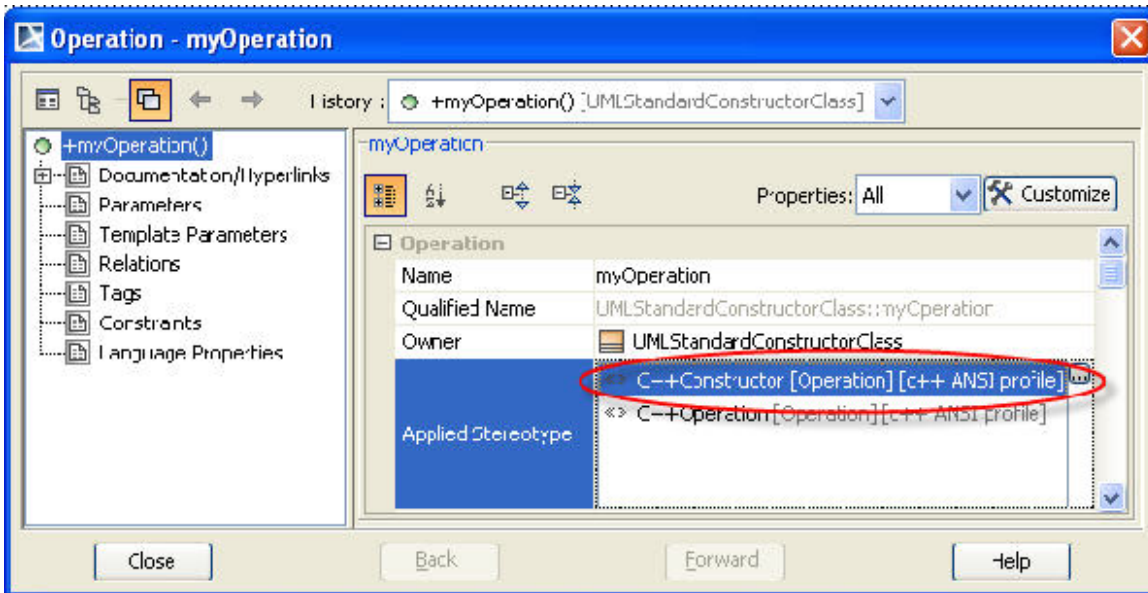


Figure 98 -- <<constructor>> stereotype

Old value	Translation
Operation applies the <<constructor>> stereotype in UML Standard Profile.	Apply the <<C++Constructor>> stereotype (c++ ANSI profile). Remove the <<constructor>> stereotype (UML Standard Profile).



<<destructor>>

The <<destructor >> stereotype is being shown in the figure below.

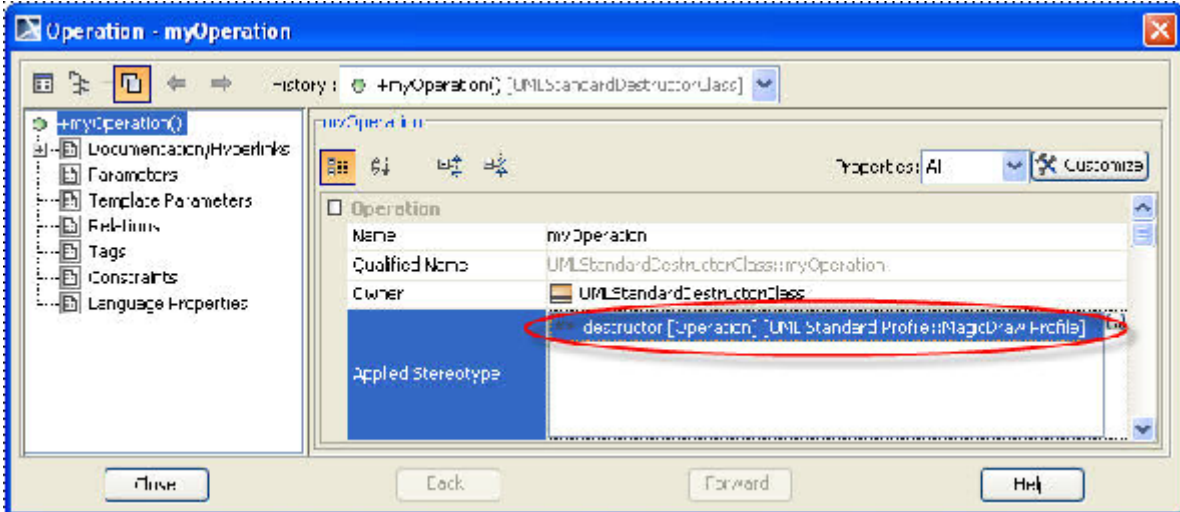
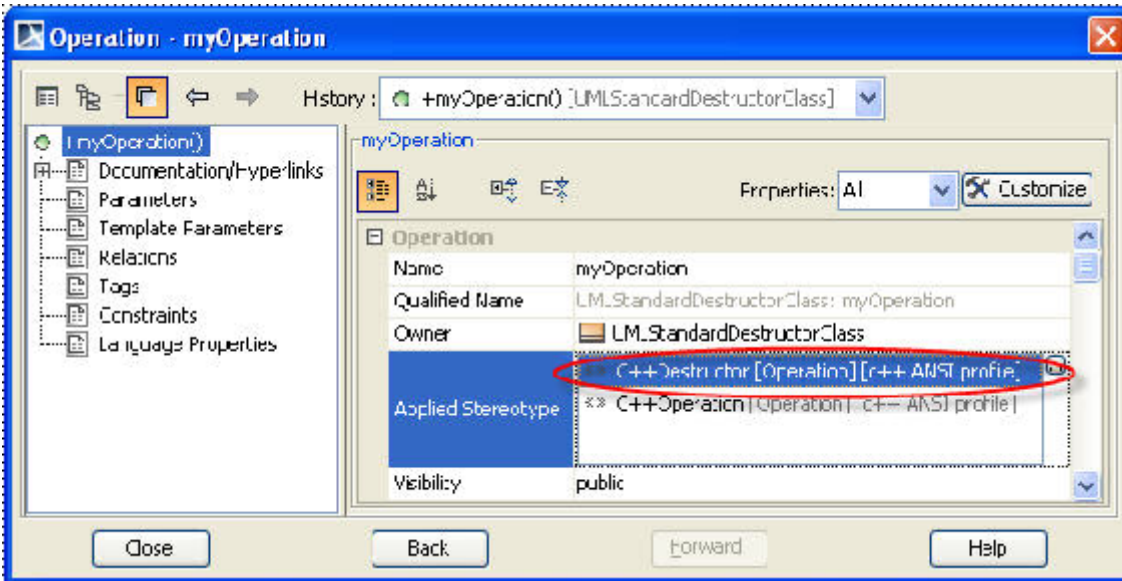


Figure 99 -- <<destructor>> stereotype

Old value	Translation
Operation applies the <<destructor>> stereotype in UML Standard Profile.	Apply the <<C++Destructor>> stereotype (C++ ANSI profile). Remove the <<destructor>> stereotype (UML Standard Profile).





## Tag Value

### C++ThrownExceptions

The <<C++Operation>> stereotype can apply to an operation.

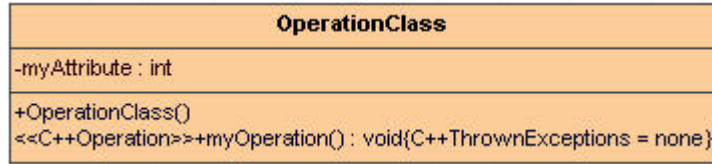


Figure 100 -- C++ThrownExceptions Example in Class Diagram

In version 12.0, all operations must apply <<C++Operation >> stereotype and default **throw exception** tag value is **any**.

The **C++ThrownExceptions** tag value is being shown in the figure below.

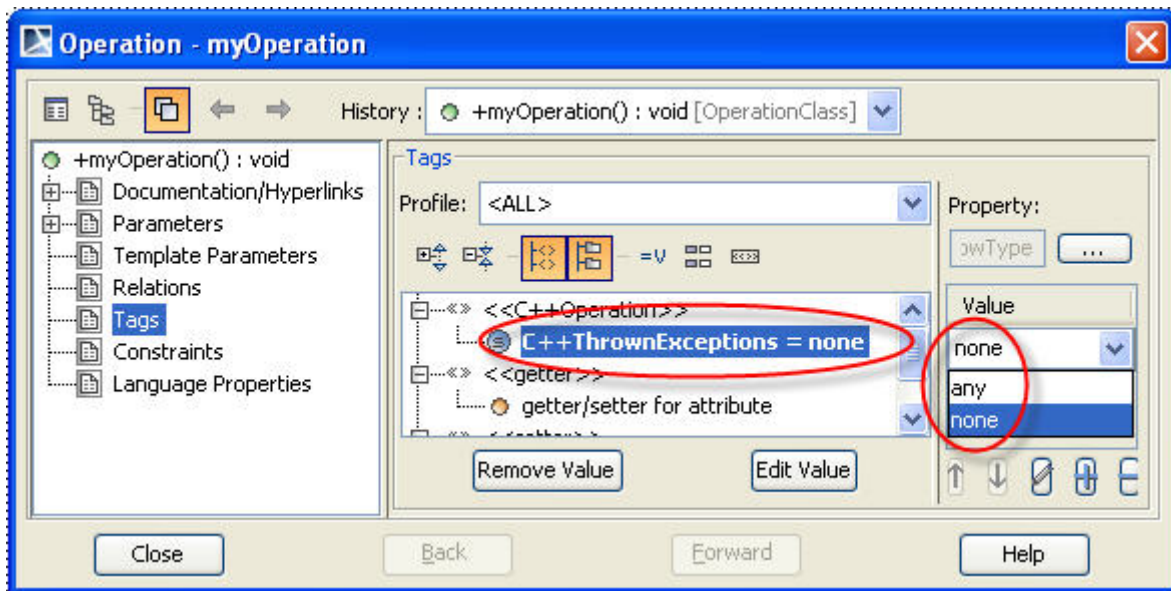


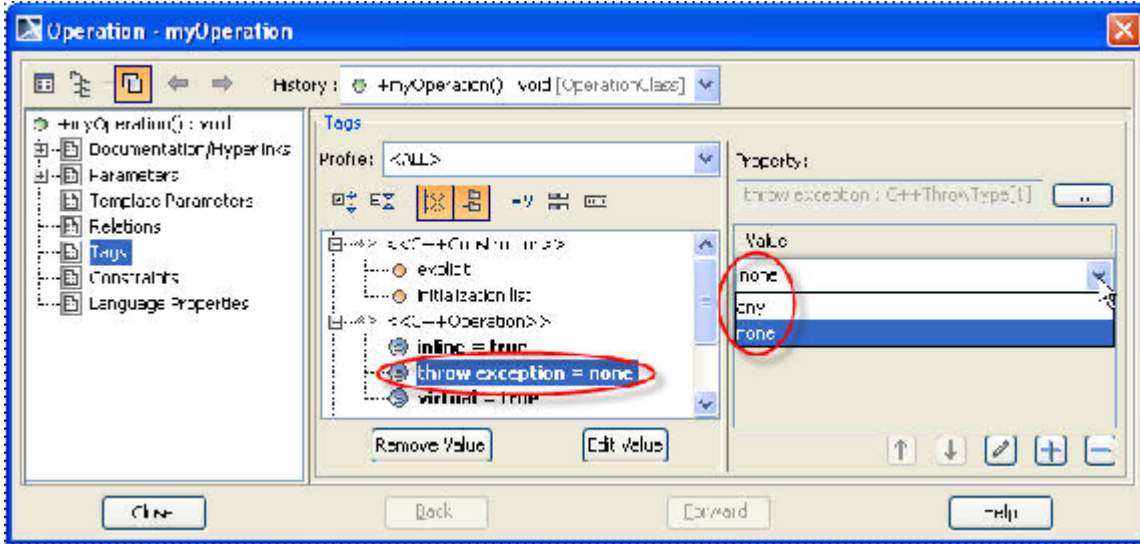
Figure 101 -- C++ThrownExceptions Tag Value

Old value	Translation
Operation <b>does not</b> apply the <<C++Operation>> stereotype in old profile (C++ Profile).	Apply the <<C++Operation>> stereotype in new profile (c++ ANSI profile) and set <b>throw exception</b> tag value to <b>any</b> .
Operation applies the <<C++Operation>> stereotype in old profile (C++ Profile) and set <b>C++ThrownExceptions</b> tag value to <b>any</b> .	Apply the <<C++Operation>> stereotype in new profile (c++ ANSI profile) and set <b>throw exception</b> tag value to <b>any</b> . Remove the <<C++Operation>> stereotype (old profile) and <b>C++ThrownExceptions</b> tag value.

Operation applies the <<C++Operation>> stereotype in old profile (C++ Profile) and set **C++ThrownExceptions** tag value to **none**.

Apply the <<C++Operation>> stereotype in new profile (c++ ANSI profile) and set **throw exception** tag value to **none**.

Remove the <<C++Operation>> stereotype (old profile) and **C++ThrownExceptions** tag value.



## Constructor and Destructor name

This example is **myClass** class that has operations named **myClass()** which is the constructor and **~myClass()** which is the destructor.

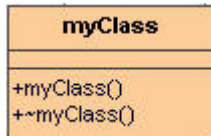


Figure 102 -- Constructor and Destructor name Example in Class Diagram

The Constructor and Destructor name are being shown in the figure below.

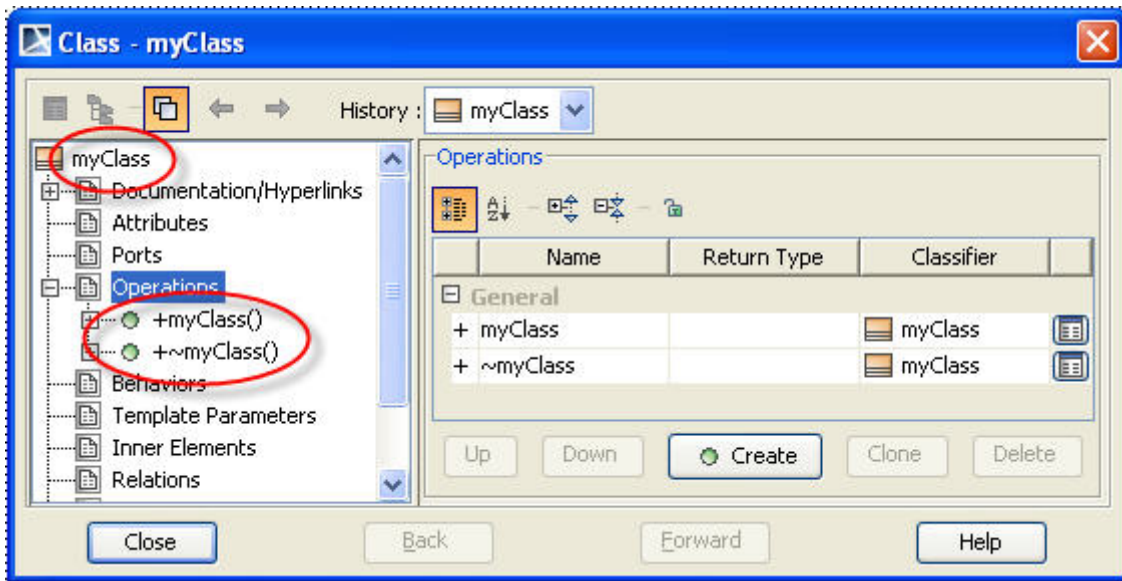
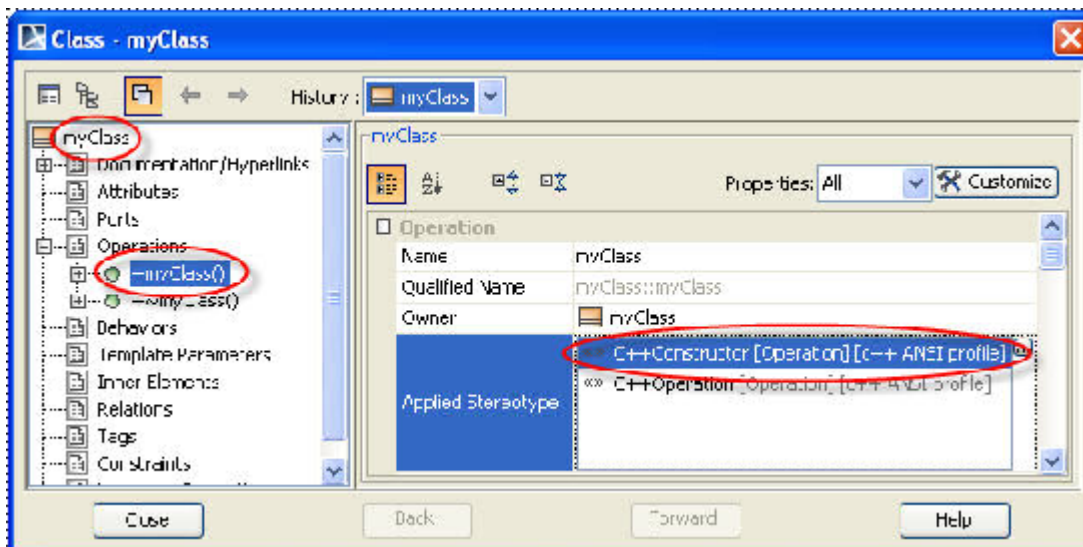


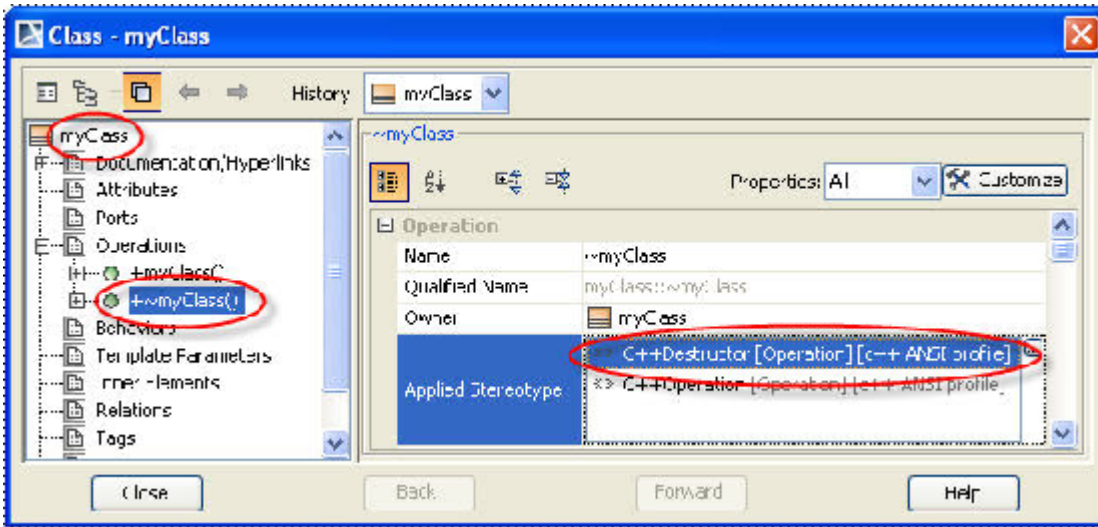
Figure 103 -- Constructor and Destructor name

Old value	Translation
Operation name is the same as the owner's name.	Apply the <<C++Constructor>> stereotype.



Operation name is '~' + the same name as the owner's name. Apply the <<C++Destructor>> stereotype.





## Data type

### bool

There are five cases of using bool data type such as Attribute type, Parameter type, Return type, Actual in Template Parameter Substitution and Default in Classifier Template Parameter. MagicDraw will change old bool data type to new bool data type in new profile.

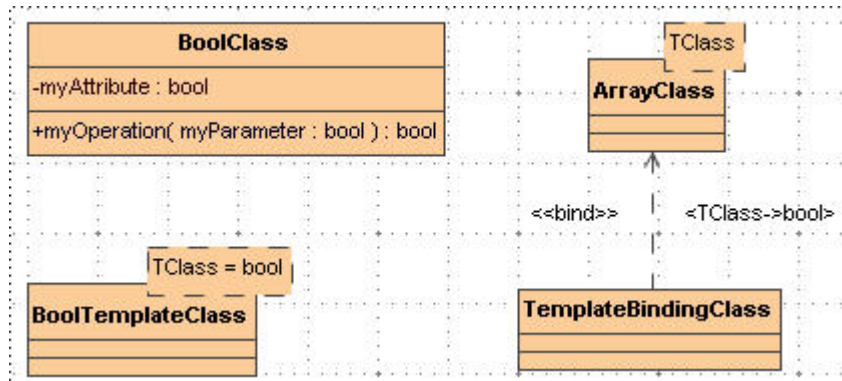


Figure 104 -- bool data type Example in Class Diagram

## Attribute type

The old bool data type in Attribute type is being shown in the figure below.

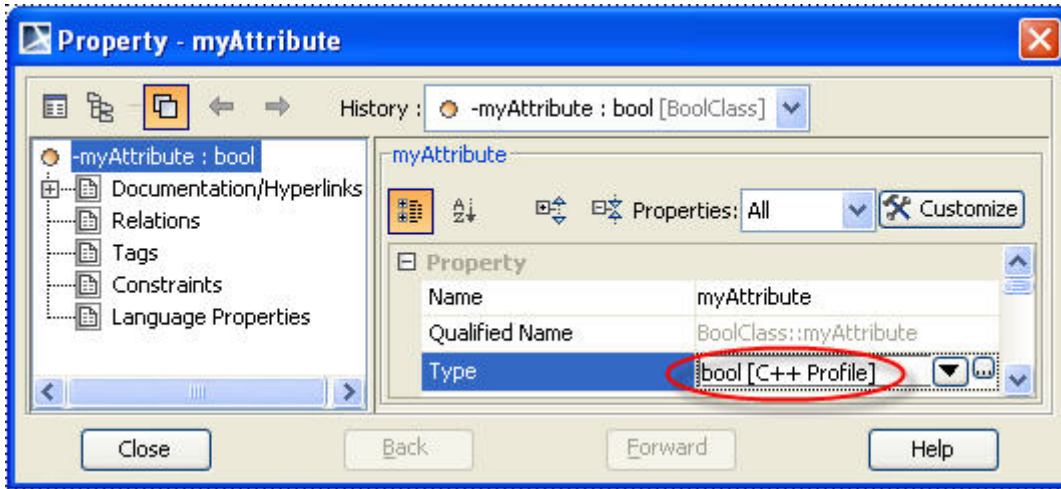
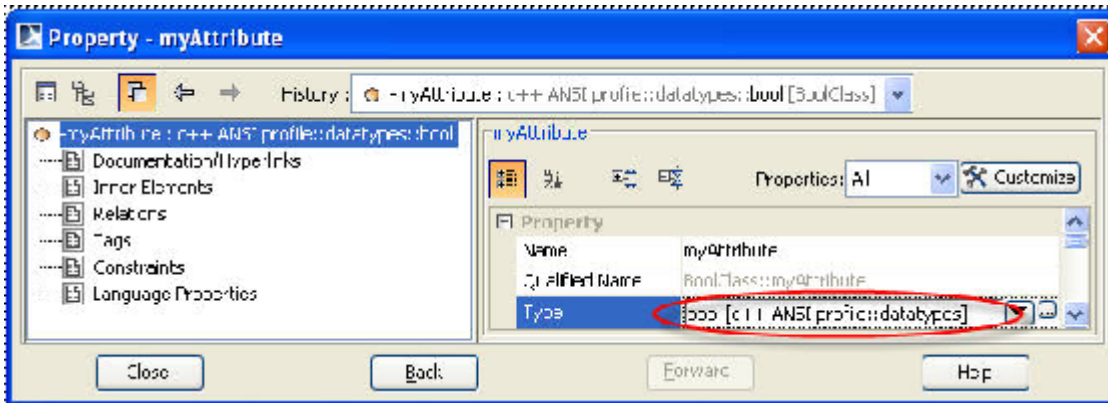


Figure 105 -- old bool data type in Attribute type

Old value	Translation
Old bool data type in old profile (C++ Profile).	Change to the new bool data type in new profile (c++ ANSI profile).



Parameter type

The old bool data type in Parameter type is being shown in the figure below.

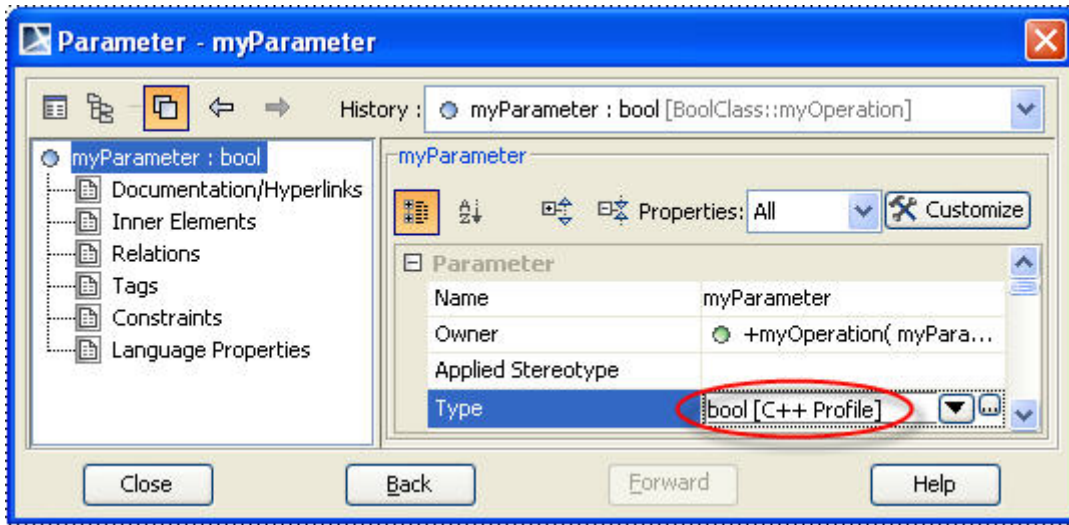
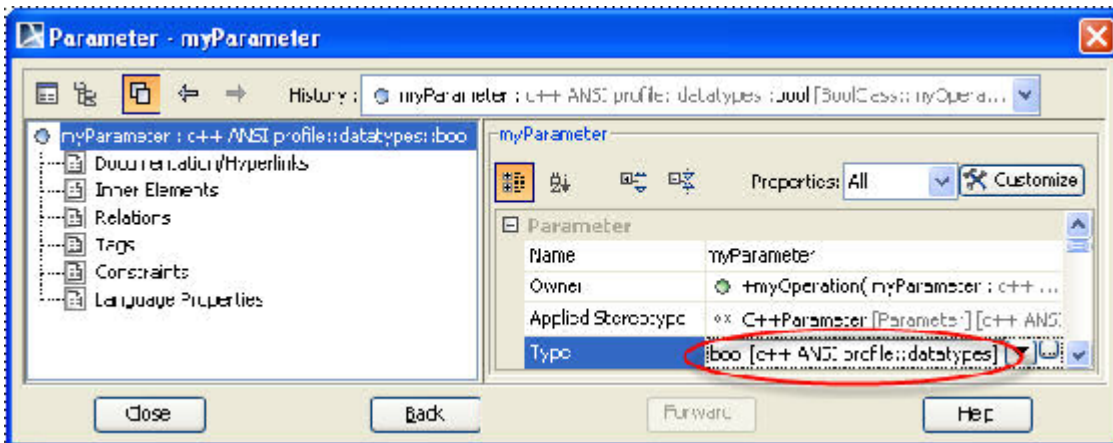


Figure 106 -- old bool data type in Parameter type

Old value	Translation
Old bool data type in old profile (C++ Profile).	Change to the new bool data type in new profile (c++ ANSI profile).



Return type

The old bool data type in Return type is being shown in the figure below.

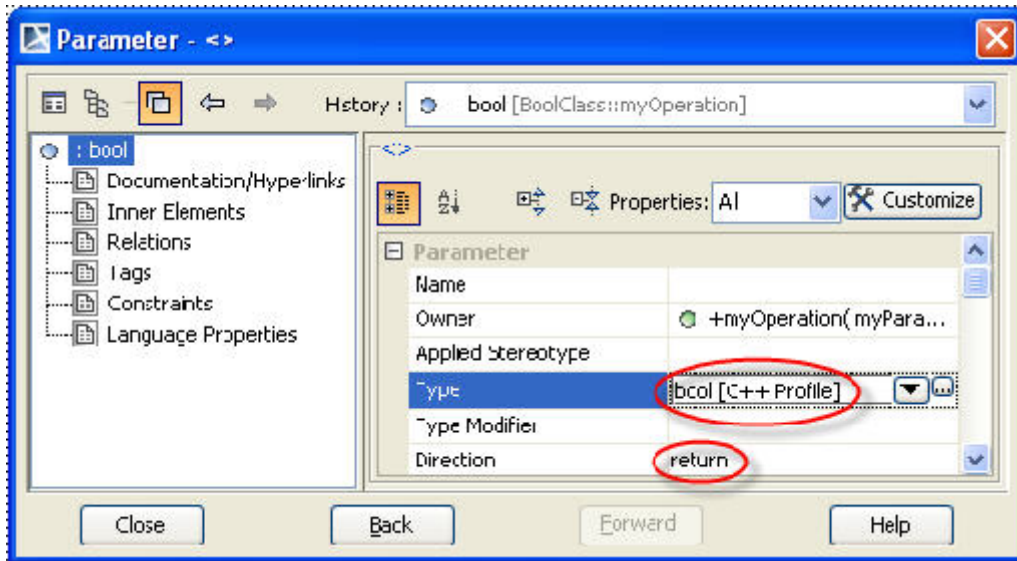
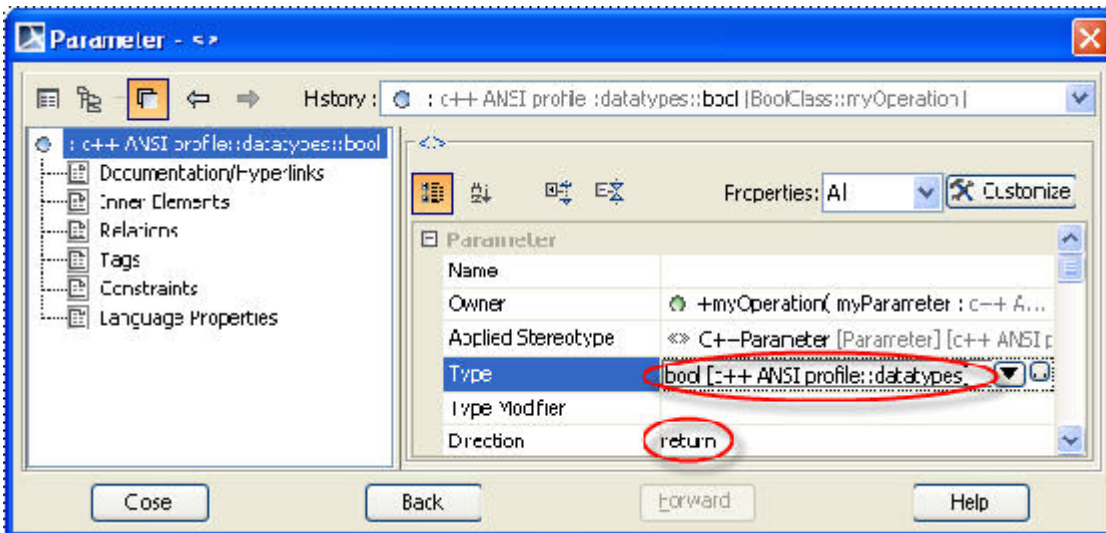


Figure 107 -- Figure 57 -- old bool type in Return type

Old value	Translation
Old bool data type in old profile (C++ Profile).	Change to the new bool data type in new profile (c++ ANSI profile).



Actual in Template Parameter Substitution

The old bool data type in Actual in Template Parameter Substitution is being shown in the figure below.

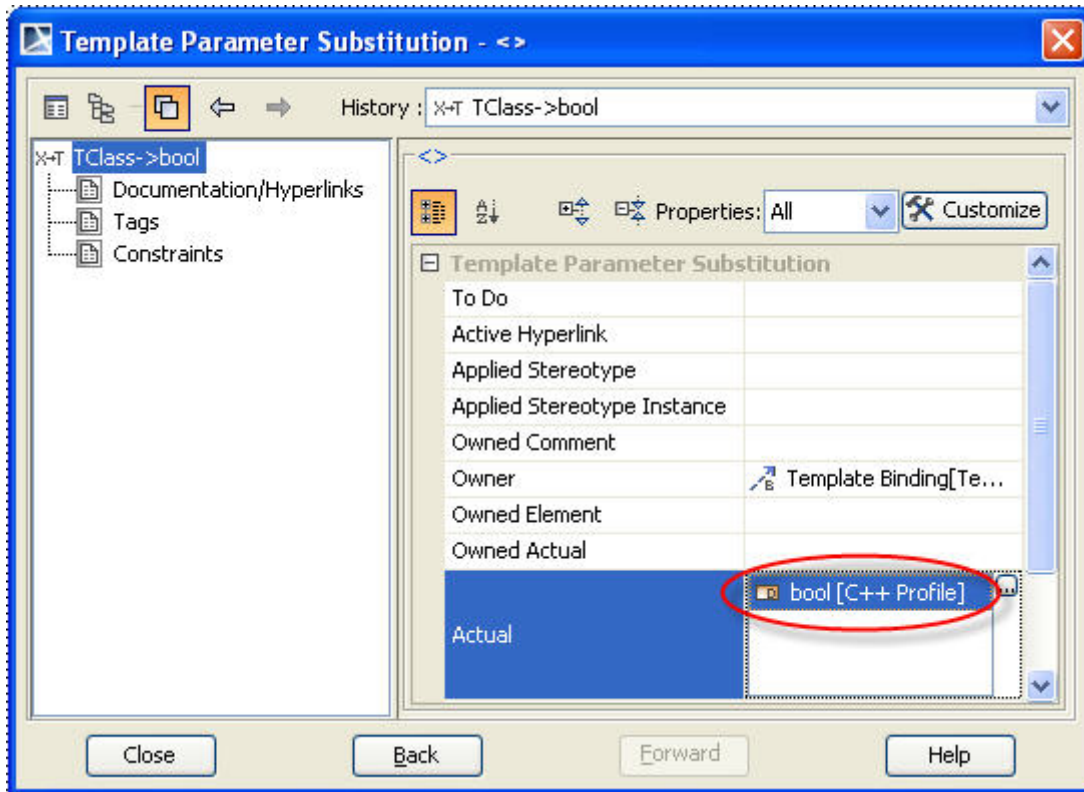
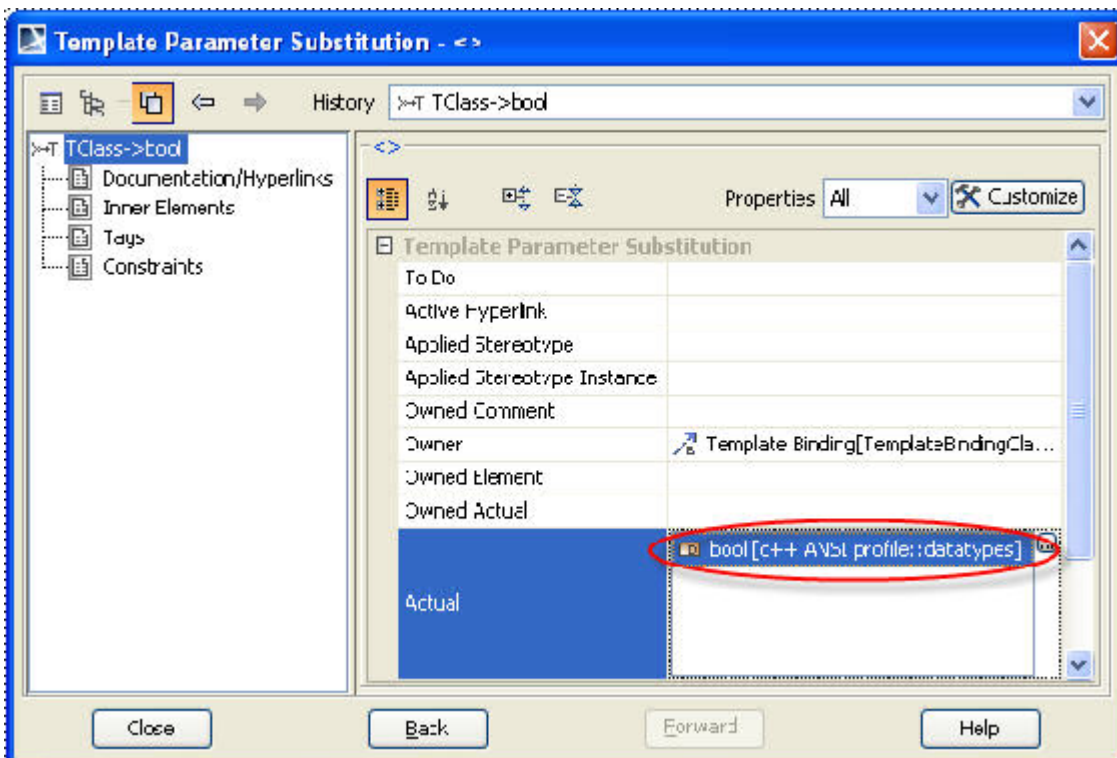


Figure 108 -- old bool data type in Actual in Template Parameter Substitution

Old value	Translation
Old bool data type in old profile (C++ Profile).	Change to the new bool data type in new profile (c++ ANSI profile).





Default in Classifier Template Parameter

The old bool data type in Default in Classifier Template Parameter is being shown in the figure below.

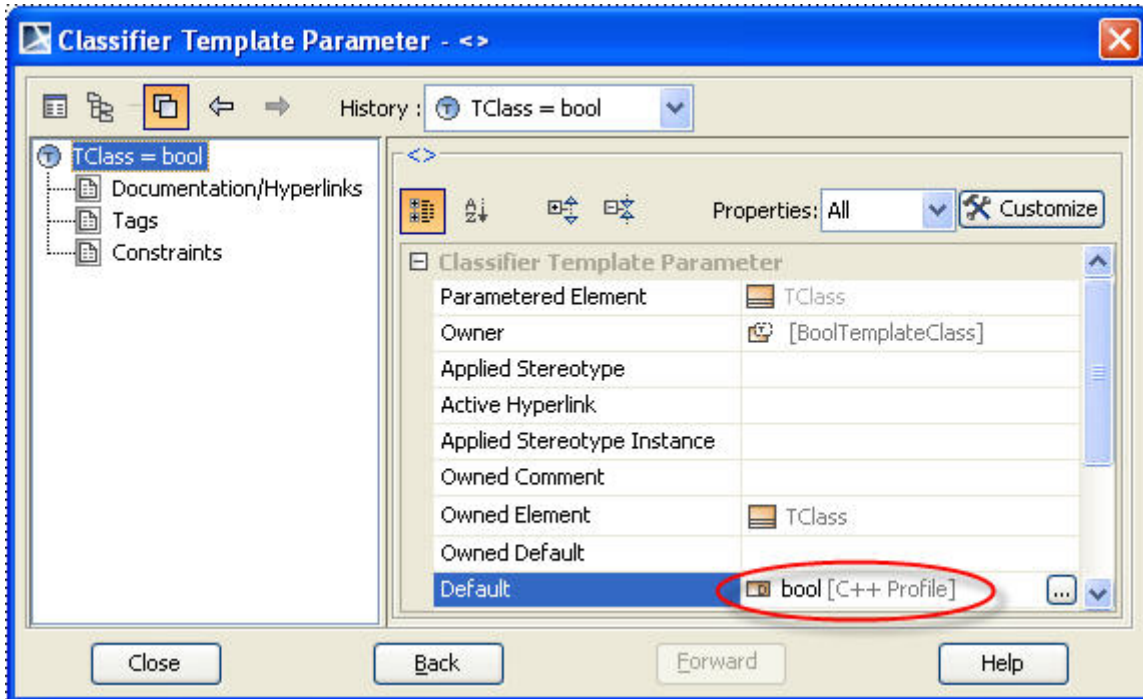
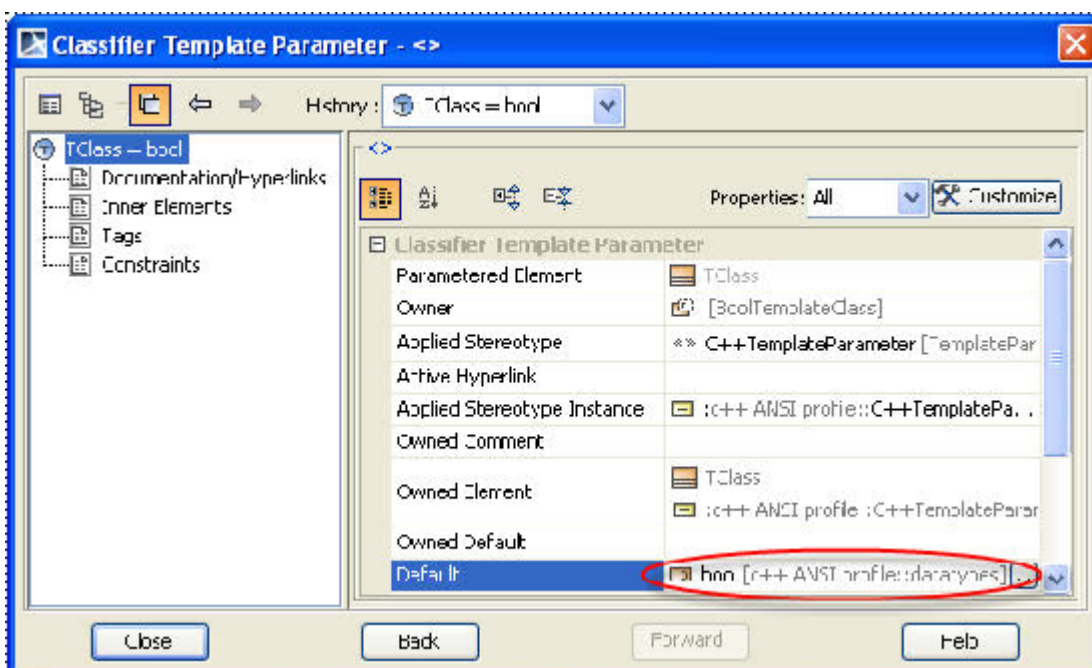


Figure 109 -- old bool data type in Default in Classifier Template Parameter

Old value	Translation
Old bool data type in old profile (C++ Profile).	Change to the new bool data type in new profile (c++ ANSI profile).



## DSL customization

This chapter describes how to customize tag values that transform from stereotypes in old C++ profile and language properties.

### Operation and Constructor

There are **Inline** and **Virtual** in Operation Language Properties that translate to Operation tag values.

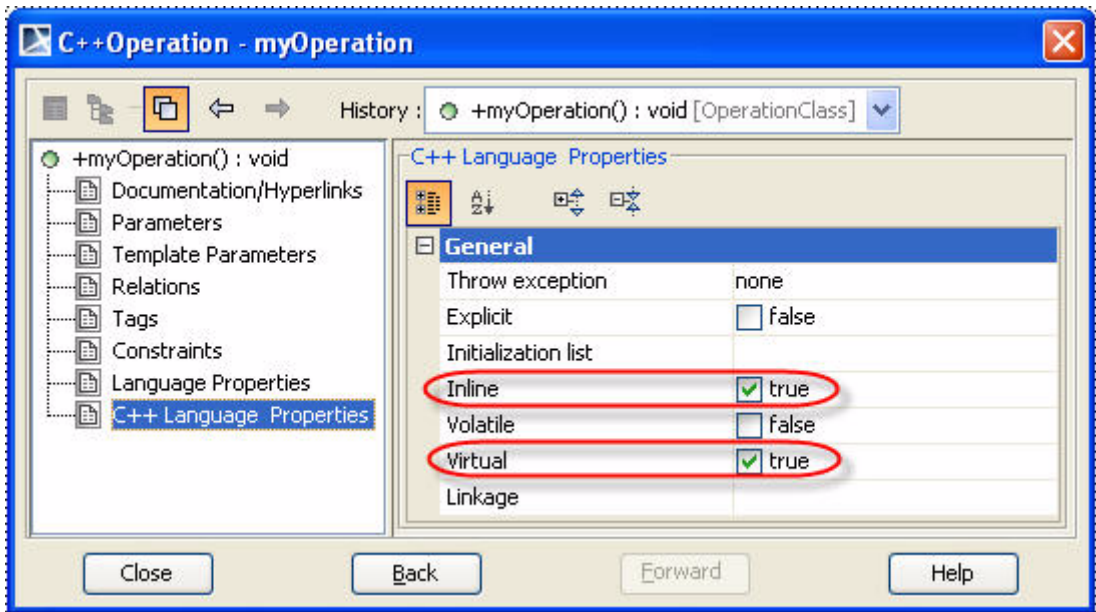


Figure 110 -- tag values in Operation

There are **Explicit** and **Initialization list** in Operation Language Properties that translate to Constructor tag values.

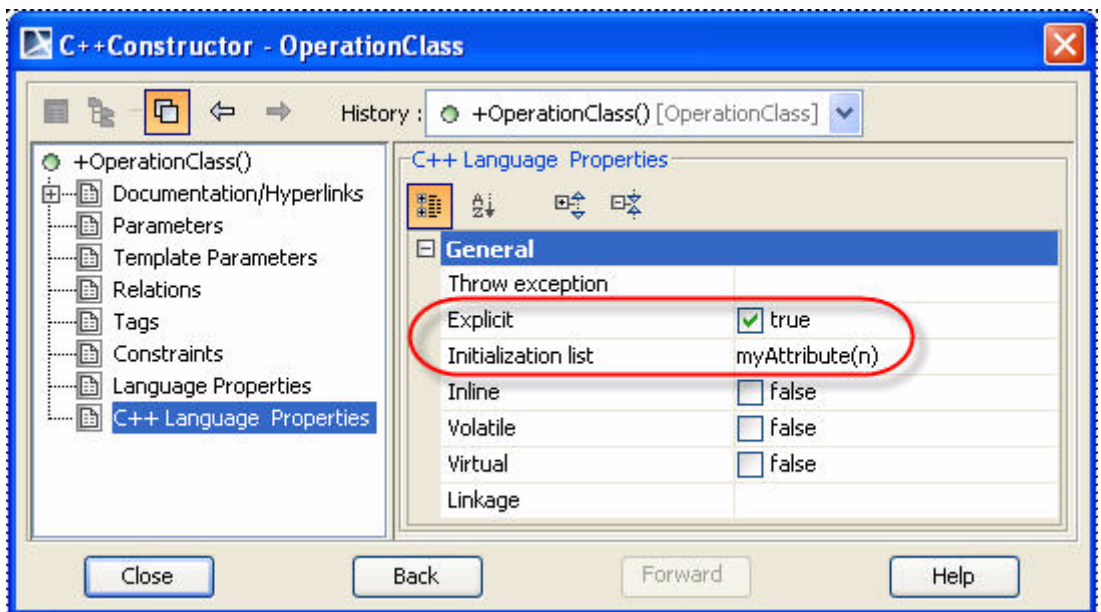


Figure 111 -- tag values in Constructor



There is **C++ThrownExceptions** tag value that translate to Operation tag value.

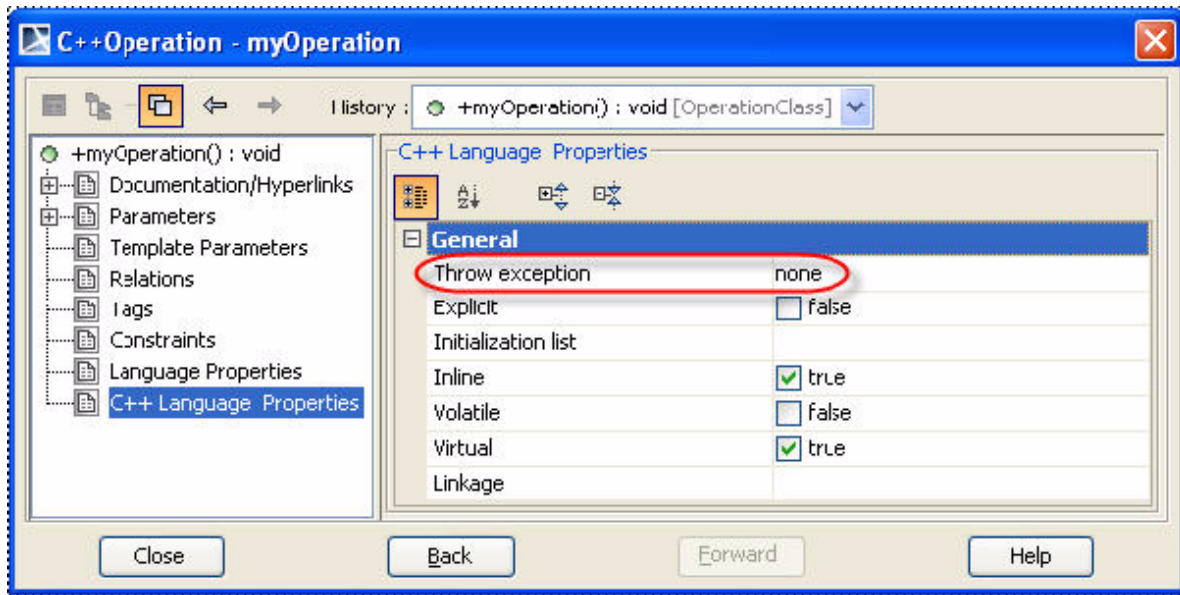


Figure 112 -- Throw exception tag value in Operation

## Attribute

There are **Abbreviated initialization**, **Bit field**, **Mutable** and **Container** in Attribute Language Properties and **Array** from **Type Modifier** that translate to Attribute tag values.

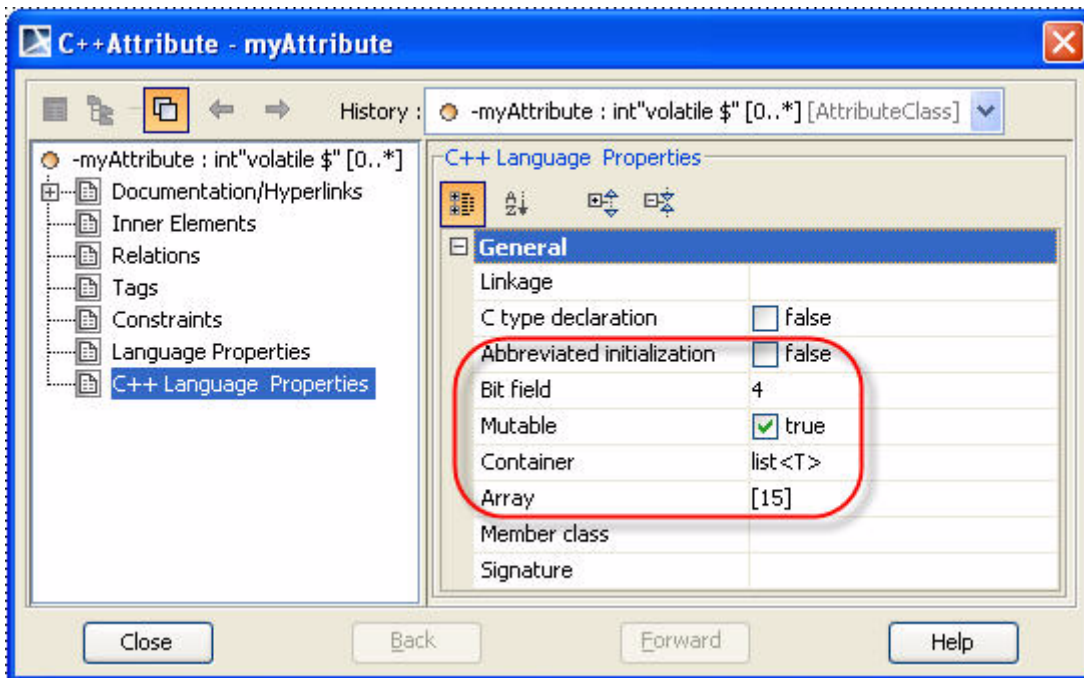


Figure 113 -- tag values in Attribute

## Generalization

There are **Virtual inheritance** and **Inheritance visibility** in Generalization Language Properties that translate to Generalization tag values.

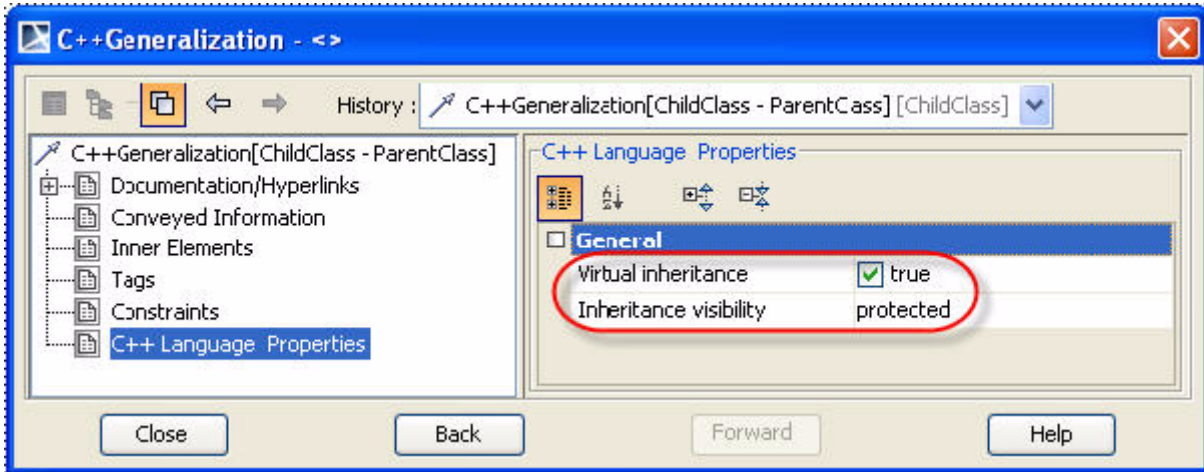


Figure 114 -- tag values in Generalization

## Enumeration literal

There is value from <<C++EnumerationLiteral>> stereotype that translate to Enumeration literal tag value.

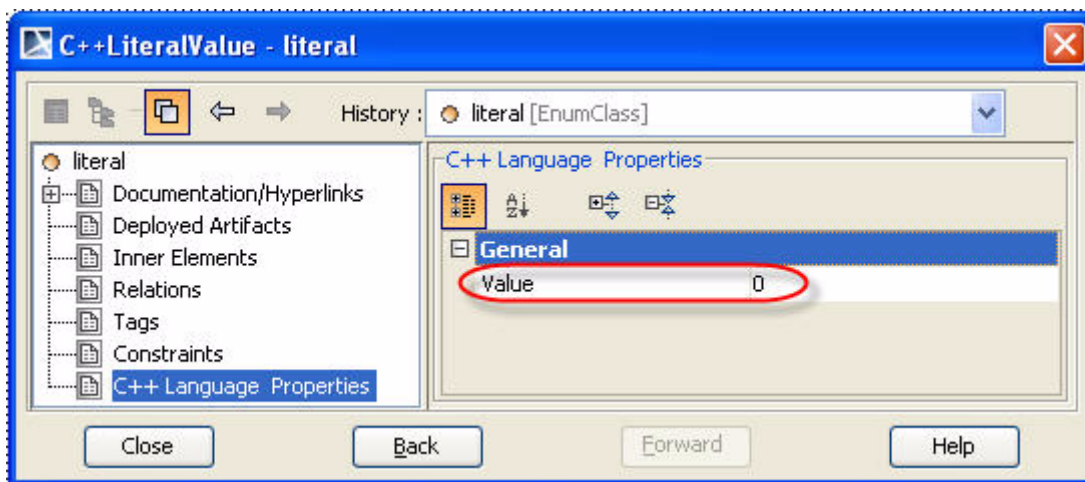


Figure 115 -- tag value in Enumeration literal

## Namespace

There is value from <<C++Namespace>> stereotype that translate to Namespace tag value.

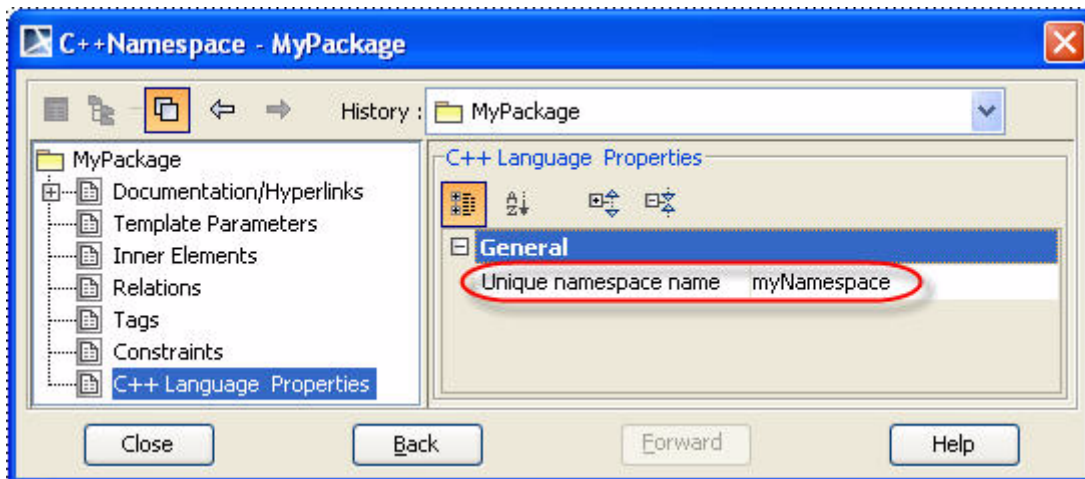


Figure 116 -- tag value in Namespace

## Template parameter

There is value from `<<C++Typename>>` stereotype that translate to Template parameter tag value.

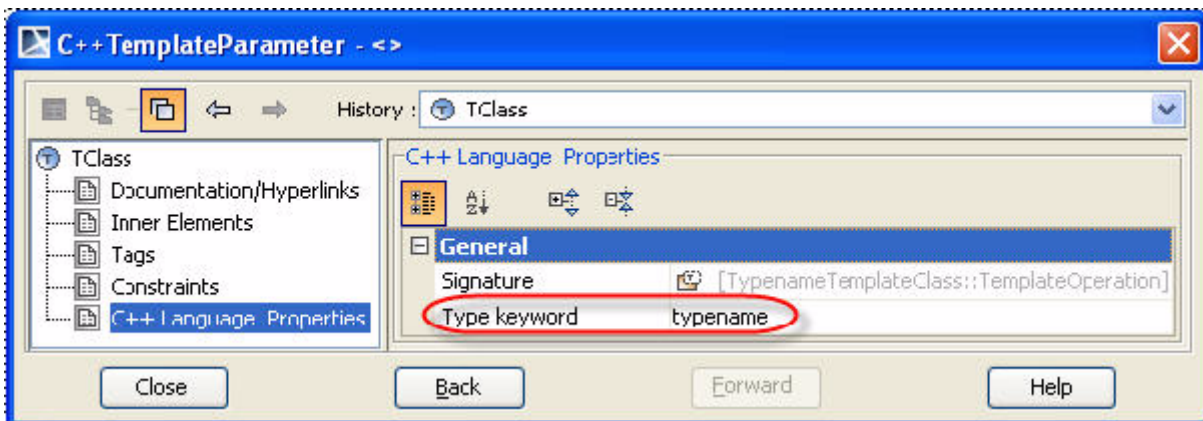


Figure 117 -- tag value in Template parameter

## Profile constraints

This chapter describes all constraints added by the ANSI profile.

### Operation

- **isQuery** set to true can only apply to member operation (non global)
- **virtual** tag set to true can only apply to non static member operation.

### Constructor

operation with `<<C++Constructor>>` stereotype

- Name should be the same as the owner's name.

## Destructor

operation with <<C++Destructor>> stereotype

- Name should be '~' + the same name as the owner's name.

## Global

Class with <<C++Global>> stereotype :

- Name length should be 0.
- All operations and attributes should be public.
- Only one global class by namespace.
- Owner of global class can only be package

## Typedef

Class with <<C++Typedef>> stereotype :

- One and only one <<C++BaseType>> dependency from a <<C++Typedef>> class.
- <<C++BaseType>> supplier dependency can only be Classifier or Operation.
- <<C++Typedef>> class can not contain attribute and operation.
- Only one inner Class or inner Enumeration is valid.

## Friend

Dependency with <<C++Friend>> stereotype.

- <<C++Friend>> client dependency can only be Class or Operation.
- <<C++Friend>> supplier dependency can only be Class.

## New in MagicDraw 12.1

### CG Properties Editor

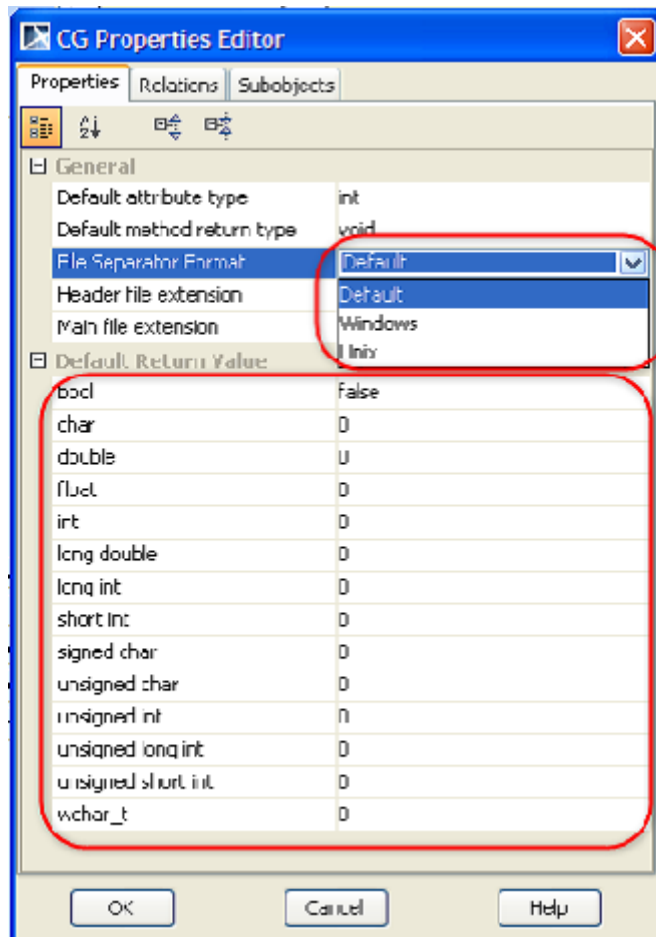


Figure 118 -- New properties

### File Separator Format

This property is used to indicate the format of file separator used in code generation.

### Default Return Value

Now you can set the default return value, which will be used for generating function body for new function. For example, if the default return value of bool is set to false and generated the function shown below will be created in .cpp file.

```
bool func(); in .h file,  
bool func()  
{  
return false;  
}
```

## Roundtrip on #include statement and forward class declaration

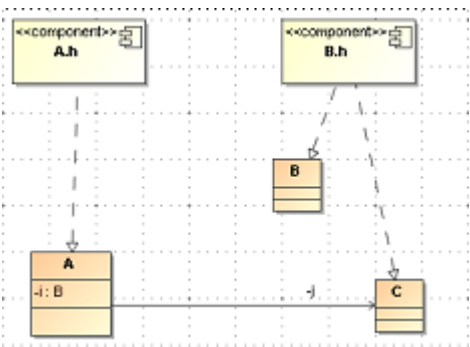
From the previous version of MagicDraw, the #include statement will be generated only at the first generation of a newly created file. According to this, it prevents user from doing roundtrip code engineering. Therefore, #include statement and forward class declaration are mapped to the model in MD 12.1. See the mapping section for more information.

At the generation time, the required #include statement will be generated by collecting data from model. Include generator will collect include information from

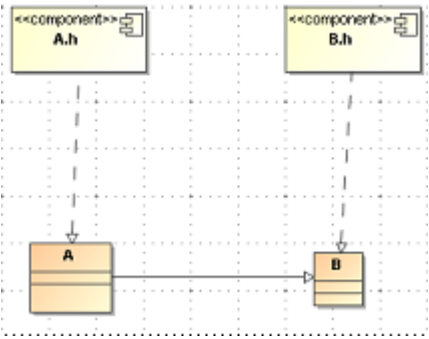
- Property
- Generalization
- Parameter
- Template Parameter Substitution
- Template Parameter default value
- Usage

Note: The usage relationship corresponding to the #include statement is also created in the model

### Property

Model	Code
 <pre>classDiagram     class A_h["&lt;&lt;component&gt;&gt; A.h"]     class B_h["&lt;&lt;component&gt;&gt; B.h"]     class A["A"]     class B["B"]     class C["C"]     A_h ..&gt; A     B_h ..&gt; B     B_h ..&gt; C     A --&gt; C : j</pre>	<pre>#include "B.h" class A{     B i;     C* j; };</pre>

Include information will be collected from both normal property (i) and association end property (j). Generalization

Model	Code
 <p>The diagram shows a UML model with two components, A.h and B.h, and two classes, A and B. A.h and B.h are represented as yellow boxes with a small icon in the top right corner and the text '&lt;&lt;component&gt;&gt;' above them. A and B are represented as orange boxes with a horizontal line near the top. A.h is associated with A, and B.h is associated with B, indicated by dashed lines with open arrowheads. A and B are associated with each other, indicated by a solid line with an open arrowhead pointing to B.</p>	<pre data-bbox="759 360 975 472">#include "B.h" class A:B{  };</pre>



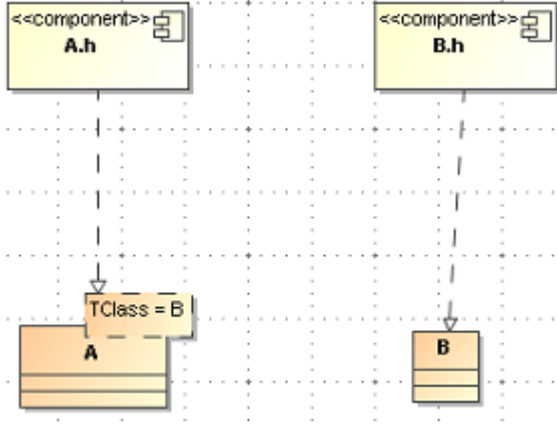
## Parameter

Model	Code
	<pre>#include "B.h" class A{     public:         void func(B i); };</pre>

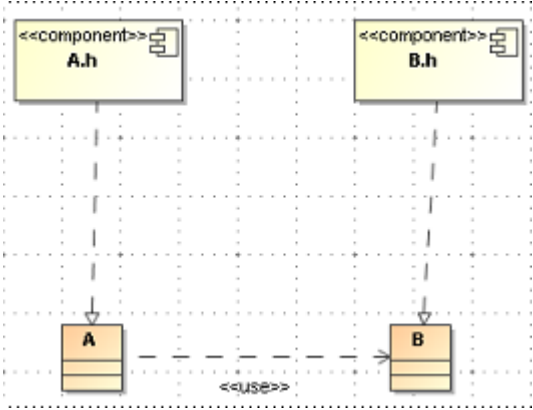
## Template Parameter Substitution

Model	Code
	<pre>#include "B.h" template &lt;class Tclass&gt; class TP{}; class A{     TP&lt;B&gt; i; };</pre>

## Template Parameter Default Value

Model	Code
 <p>The diagram illustrates a model with two components, A.h and B.h, each represented by a yellow box with a component icon. Below A.h is a class box for A, with a parameter TClass = B. Below B.h is a class box for B. Dashed arrows point from each component to its respective class.</p>	<pre data-bbox="805 336 1197 425">#include "B.h" template &lt;class TClass=B&gt; class A{};</pre>

### Usage

Model	Code
 <p>The diagram shows a UML model with a grid background. At the top, there are two component boxes: 'A.h' and 'B.h', both labeled with the stereotype &lt;&lt;component&gt;&gt;. Below them are two class boxes: 'A' and 'B'. A dashed arrow points from 'A.h' to 'A', and another from 'B.h' to 'B'. A dashed arrow points from 'A' to 'B' with the label &lt;&lt;use&gt;&gt;.</p>	<pre>#include "B.h" class A{ };</pre>

## Project Option and Code Generation Options

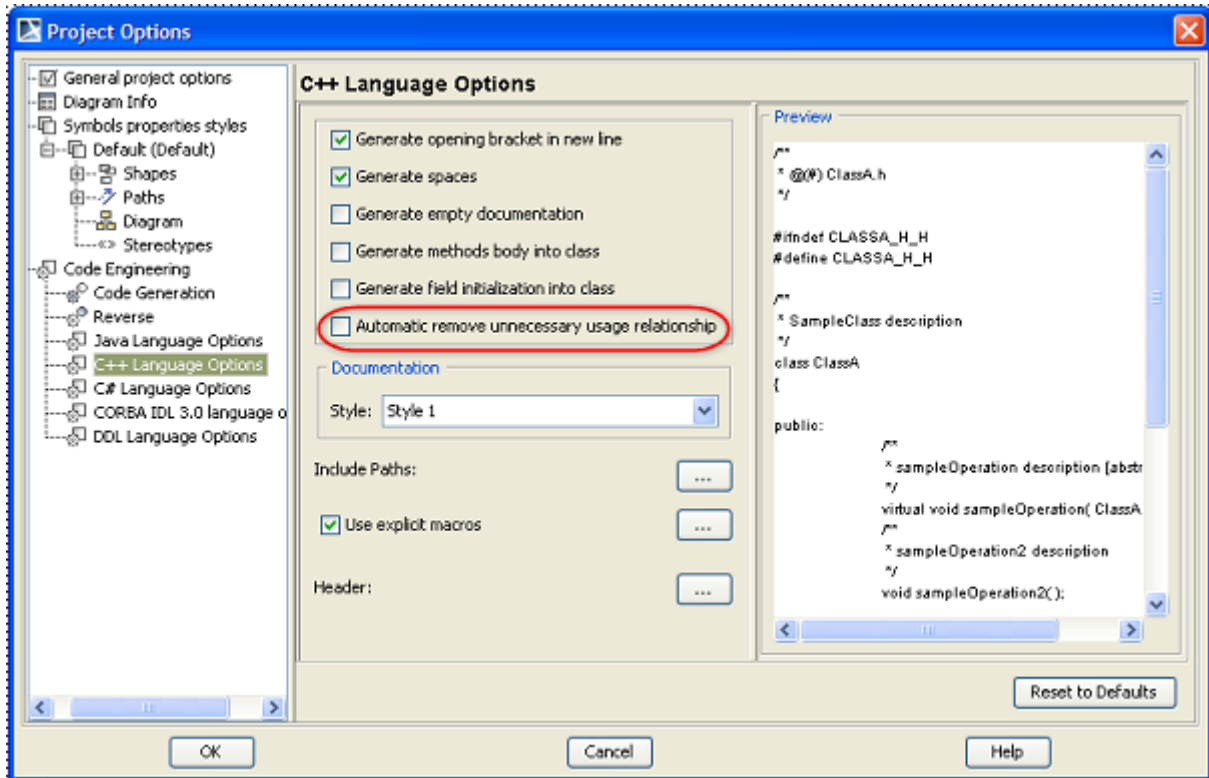


Figure 119 -- New Option in Project Options

If this new option is selected, the same option in Code Generation Option dialog will be selected as shown below.

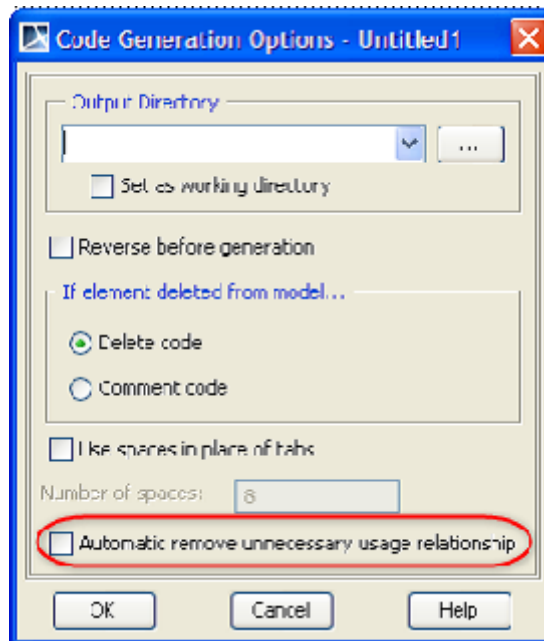
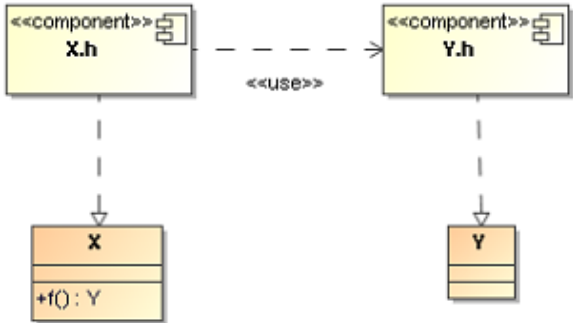
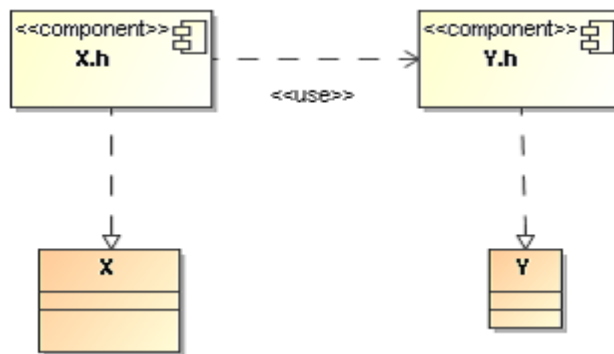


Figure 120 -- New option in Code Generation Options

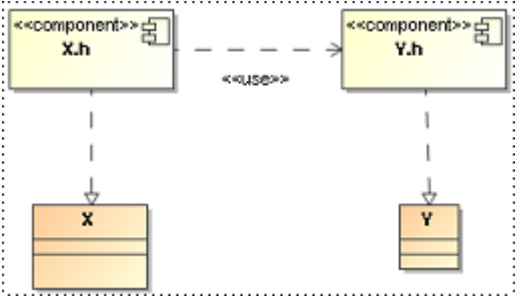
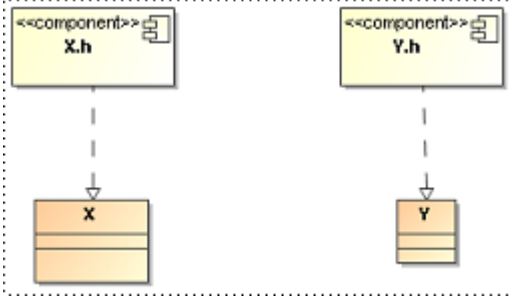
If the user generates code with this option selected, code generator will automatically detect and remove unnecessary usage relationship from model. See the example below for better understanding.

Model	Code
 <pre> classDiagram     class X_h["&lt;&lt;component&gt;&gt; X.h"]     class Y_h["&lt;&lt;component&gt;&gt; Y.h"]     class X["X"]     class Y["Y"]     X_h ..&gt; Y_h : &lt;&lt;use&gt;&gt;     X_h ..&gt; X     X ..&gt; Y     X ..&gt; Y : +f(): Y     </pre>	<pre> //X.h #include "Y.h" class X{ Y f(); };     </pre>

From model above, if the user removes function f():Y from class X, in the model point of view, the usage from X.h to Y.h is not necessary.



See the difference between outcomes when selecting and not selecting Automatic remove unnecessary usage relationship.

Automatic remove unnecessary usage relationship	Model	Code
<b>Selected</b>		<pre data-bbox="1002 421 1222 506">//X.h #include "Y.h" class X{};</pre>
<b>Not Selected</b>		<pre data-bbox="1002 790 1158 846">//X.h class X{};</pre>

## New in MagicDraw 14.0

### Support C++ dialects

C++ code engineering set in MagicDraw 14.0 supports three dialects

- ANSI – conform to ISO/IEC 14882 specification for C++ programming language
- CLI – conform to ECMA-372 C++/CLI Language Specification based on Microsoft Visual Studio 2005
- Managed – conform to Managed Extensions for C++ Specification introduced in Microsoft Visual Studio 2003

**Note** The syntax under Managed dialect is deprecated in Microsoft Visual Studio 2005.

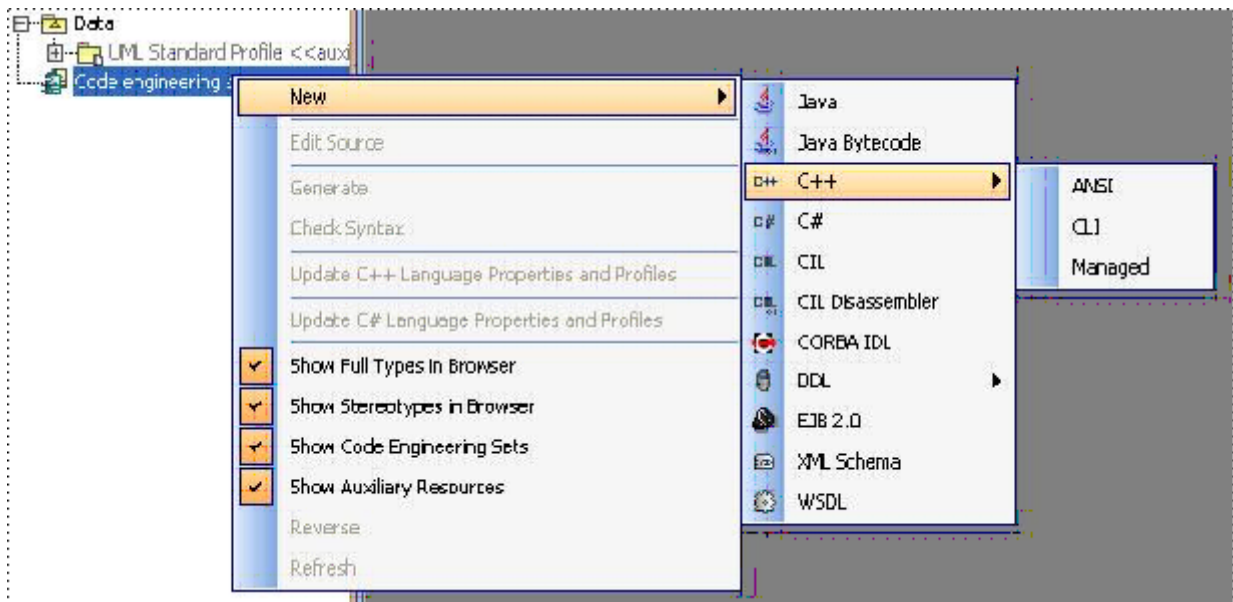


Figure 121 -- New dialects in C++ CE



## CG Properties Editor

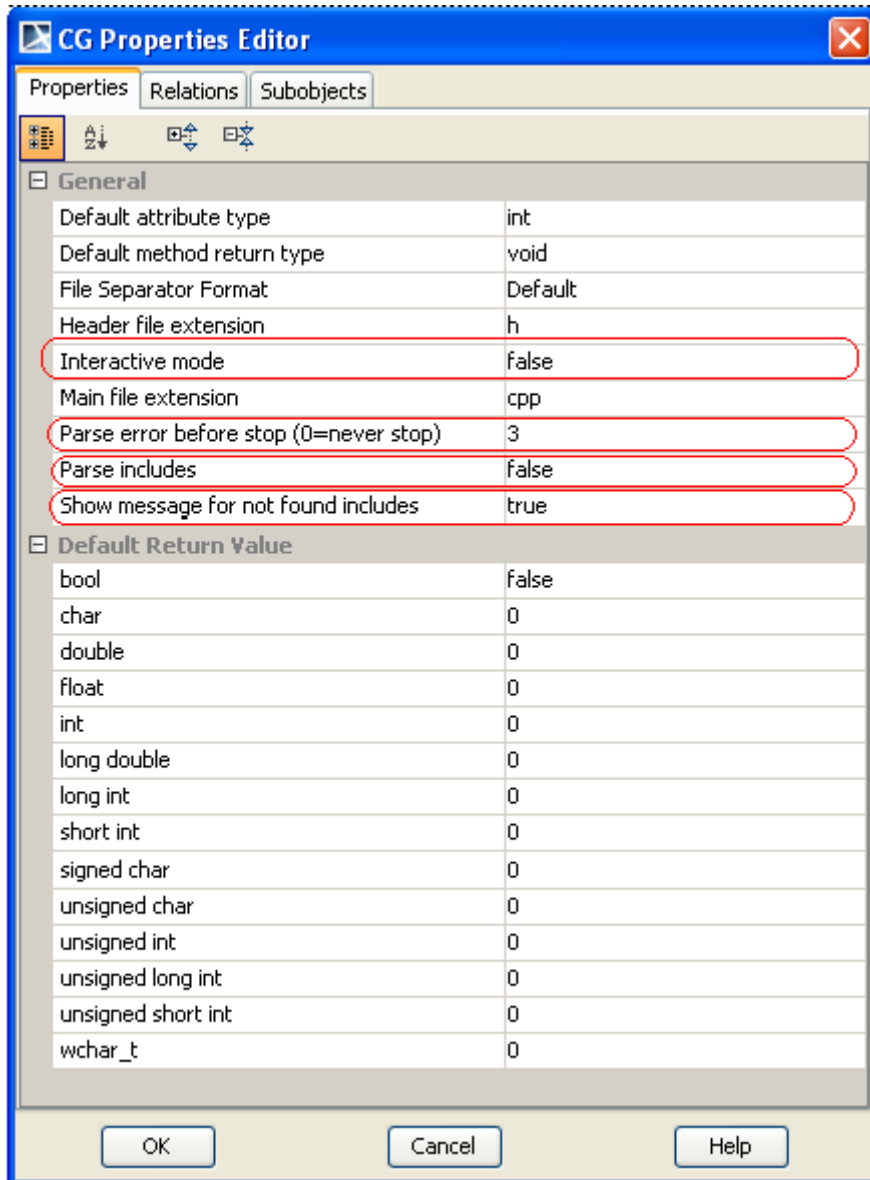
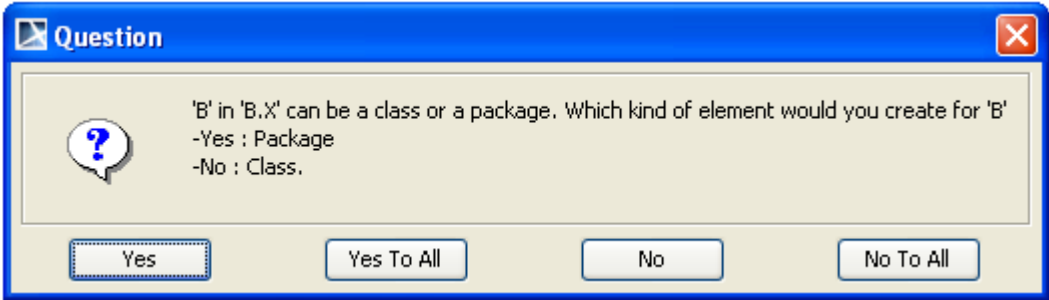


Figure 122 -- New properties

Property name	Description
<b>Interactive mode</b>	<p>If enabled, user will be questioned during the reverse process when the parser cannot decide whether the symbol is a class or namespace.</p> <p>For example</p> <p>If you reverse the following code</p> <pre>class A {     B::X i; };</pre> <p>The parser does not have enough information to decide whether B is class or namespace.</p>  <p><i>Figure 123 -- Question dialog for interactive mode</i></p> <p><b>NOTE</b> If the default mode, not interactive mode, the reverse module will automatically do the best guess according to the information in code.</p>
<b>Parse error before stop</b>	This property allows users to specify the number of errors to ignore before the reverse process will be terminated. The user can set the property to zero to allow CE to reverse all code before stopping.
<b>Parse Includes</b>	If set to false, the reverse module will reverse only the selected files and ignore all #include statements.
<b>Show message for not found includes</b>	<p>Used to display messages in the message window when the reverse module cannot find the files included in the #include statement.</p> <p><b>NOTE</b> You have to set the property <b>Parse Includes</b> to true in order to use the property <b>Show message for not found includes</b>.</p>

## New in MagicDraw 16.8

### Doxygen-after-member documentation

**NOTE** This feature only works for import-code-only mode.

A doxygen documentation after a member is reversed now, the comment after the member should use the special doxygen tag used to mark a documentation after a member.

- `'/**<`
- `'//<`

- `///`

## @see on documentation

**NOTE** This feature only works for import-code-only mode.

Documentation with doxygne `@see` tag is reversed with a link to the model. The documentation should use one of the following special doxygen tag to describe an `@see` element.

- `@see`
- `\see`
- `@sa`
- `\sa`

**NOTE** The documentation must be in the form of “`@see element`”. Only one element follows `@see` will be reversed as an element link in model documentation.

## Symbian macro

The following macro definitions have been added to the explicit macro list.

```
// Symbian macro
#define _L(string) TPtrC((const TText*) string)
#define _LIT(name,s) const static TListC<sizeof(s)> name = {sizeof(s) -1,s}
#define _LIT8(name,s) const static TListC<sizeof(s)> name = {sizeof(s) -1,s}

#define IMPORT_C /*__declspec(dllimport)*/
#define EXPORT_C /*__declspec(dllexport)*/

#define NONSHARABLE_CLASS(x) class x
#define NONSHARABLE_STRUCT(x) struct x

#define GLREF_D extern
#define GLDEF_D
#define LOCAL_D static
#define GLREF_C extern
#define GLDEF_C
#define LOCAL_C static
```

## Tutorial

### Type Modifier

In order to correctly generate code, the user has to put type modifier in the correct format. The dollar sign, \$, will be replaced with Type.

For example, If you want to generate the following code,

```
class A
{
    const int* const i;
};
```

type modifier must be set to “const \$\* const” as shown in the figure below.

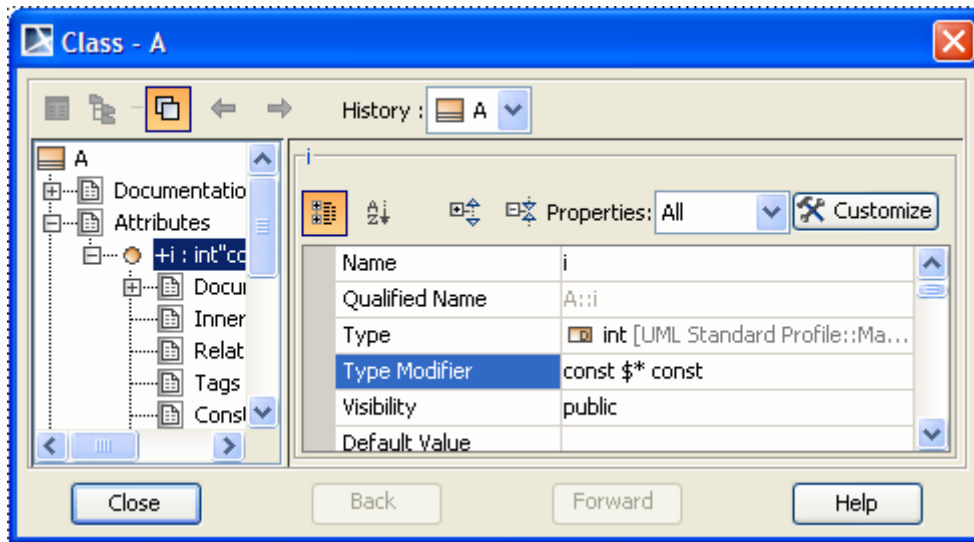


Figure 124 -- Type Modifier Example

## Global Member

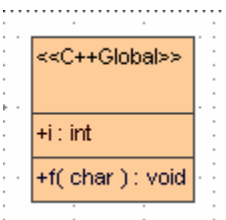
The new C++ ANSI profile allows you to model C++ global member. In order to do this, please follow the steps below.

1. Create a class
2. Apply stereotype <<C++Global>>



**Note** You can leave the class unnamed. Name of <<C++Global>> class does not have any effect on code generation.

3. Create members, attribute and function in this class. All members should set their visibilities to Public.



See "Global functions and variables", on page 94 for example.

## Typedef

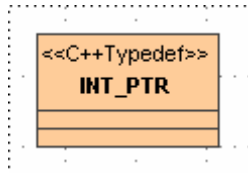
With new C++ ANSI profile, you can model both normal Typedef and Typedef of function pointer.

## Normal Typedef

The steps below show you how to create the model that can be used to generate the following code.

```
typedef int* INT_PTR;
```

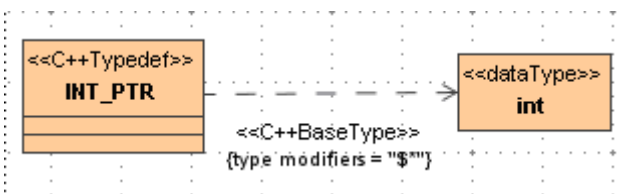
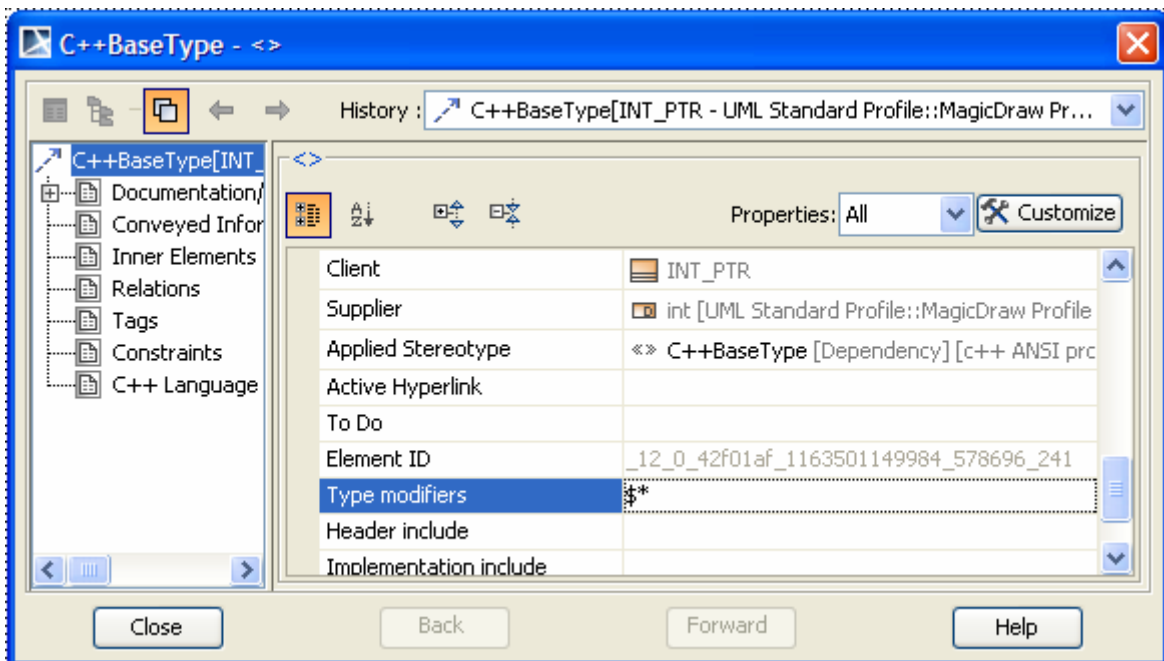
1. Create a class, apply stereotype <<C++Typedef>> and name it with typedef name. In this case, INT\_PTR.



2. Draw Dependency relationship from such class to base type element. In this case, int.



3. Apply the Dependency with <<C++BaseType>> and set type modifier value of Dependency to "\$\*".

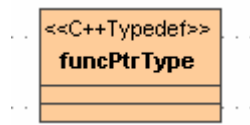


## Typedef of function pointer

The steps below show you how to create the model that can be used to generate the following code.

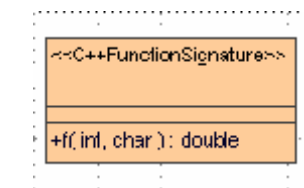
```
typedef double (*funcPtrType)(int, char);
```

1. Create a class, apply stereotype <<C++Typedef>> and name it with typedef name. In this case, funcPtrType.

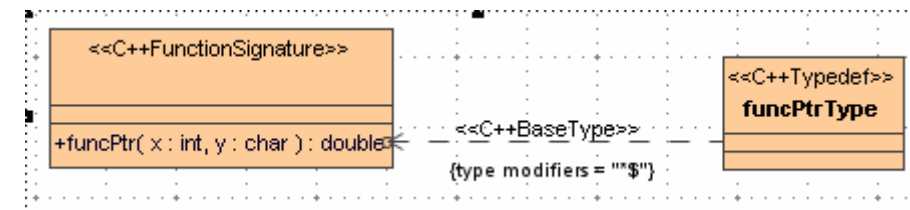


2. Create an operation with function signature type of function pointer in <<C++FunctionSignature>> class. The operation name does not have any effect on code generation.

**Note** If <<C++FunctionSignature>> class does not exist, create a new one.



3. Draw Dependency relationship from **funcPtrType** class to **f(int, char): double**, apply <<C++BaseType>> to the Dependency and set type modifier to \*\$.



See Typedef for mapping example.

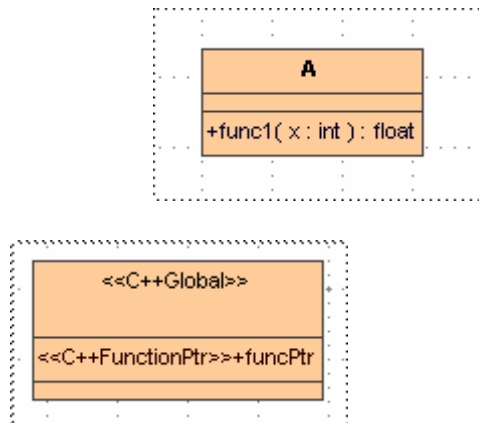
## Function Pointer

The steps below show you how to create the model that can be used to generate the following code.

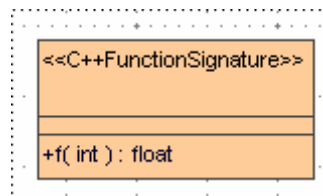
```
class A
{
public:
    float func1(int x);
};
float (A*funcPtr)(int);
```

Suppose that the model for class A is already created.

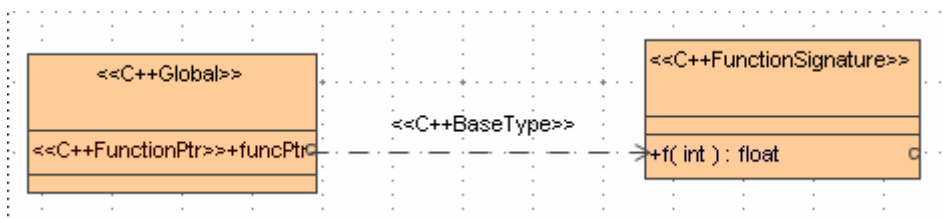
1. Create attribute with name, in this case, funcPtr and apply <<C++FunctionPtr>> stereotype. In this example, funcPtr is created in <<C++Global>> class means that funcPtr is global member.



2. Create an operation with function signature type of function pointer in <<C++FunctionSignature>> class. The operation name does not have any effect on code generation.

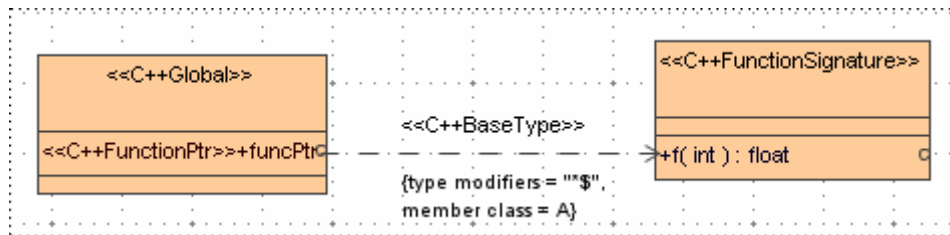
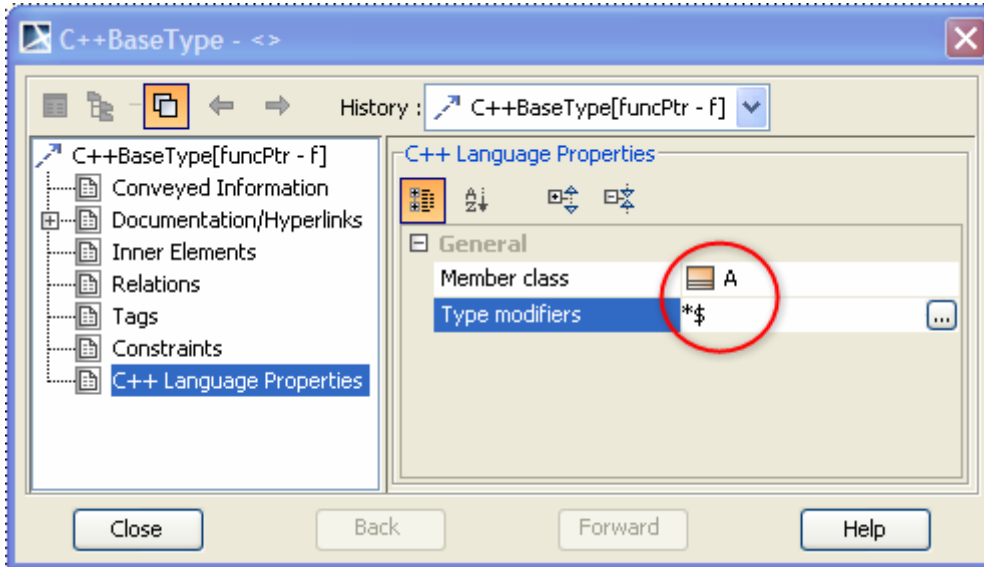


3. Draw a Dependency from funcPtr to f that is just created in <<C++FunctionSignature>> class and set <<C++BaseType>> stereotype to the Dependency.





- Open specification dialog of <<C++BaseType>> Dependency.  
Put \*\$ as value of Type Modifier in C++ Language Properties.  
Set Member class to class A.



See 3.16 Function pointer for mapping example.

## Friend

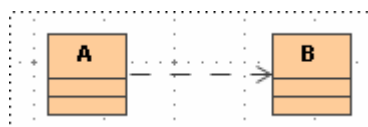
With new C++ ANSI profile, you can model C++ friend, both friend class and friend function.

### Friend Class

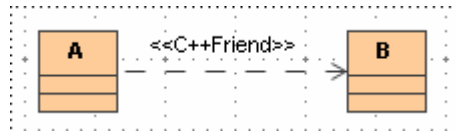
The steps below show you how to create the model that can be used to generate the following code.

```
class A{};
class B
{
    friend class A;
};
```

- From the existing classes, A and B, draw dependency from class **A** to class **B**.



2. Apply stereotype <<C++Friend>> to the dependency

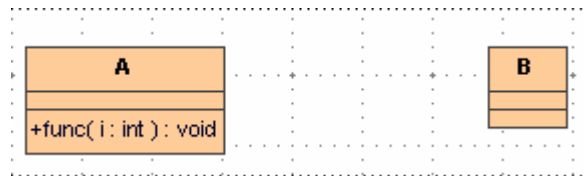


## Friend Function

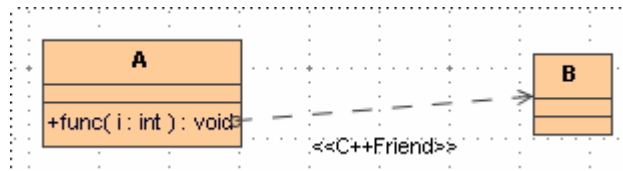
To model friend function, do the same steps as modeling friend class.

```
class A
{
    void func(int i);
};
class B
{
    friend void A::func(int i);
};
```

1. From the existing classes, A and B, draw dependency from **func** in class **A** to class **B**.



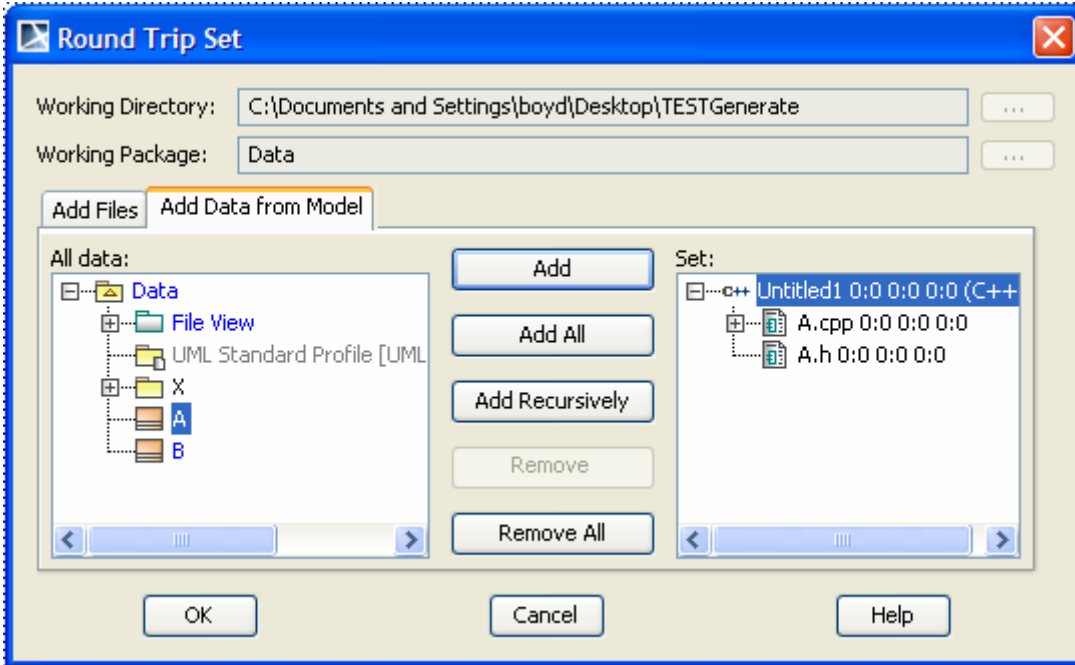
2. Apply stereotype <<C++Friend>> to the dependency.



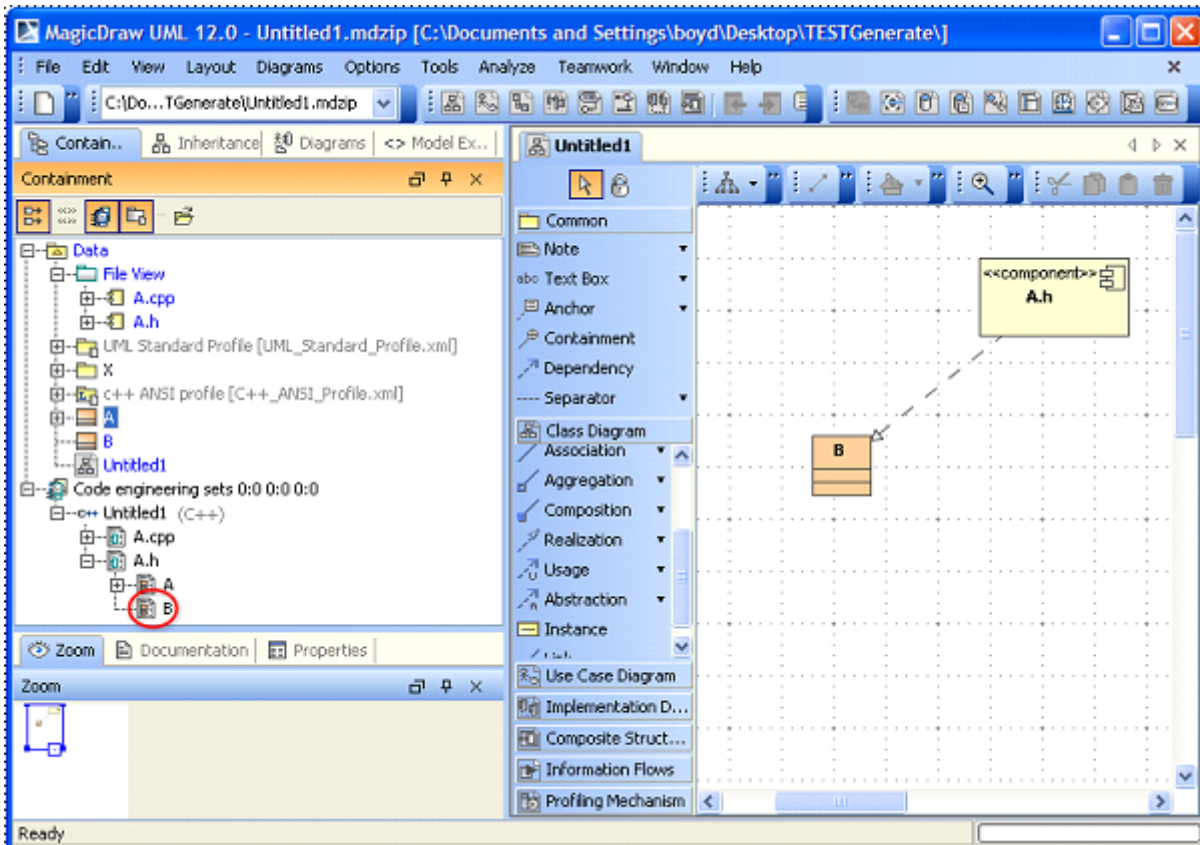
See Friend declaration for mapping example.

## How to specify component to generate code to

For general case, if the user adds data from a model in Round Trip Set (as shown below), the default component, in this case **A.cpp** and **A.h**, will be added to code engineering set, which means that on code generation, class A and its members will be created.

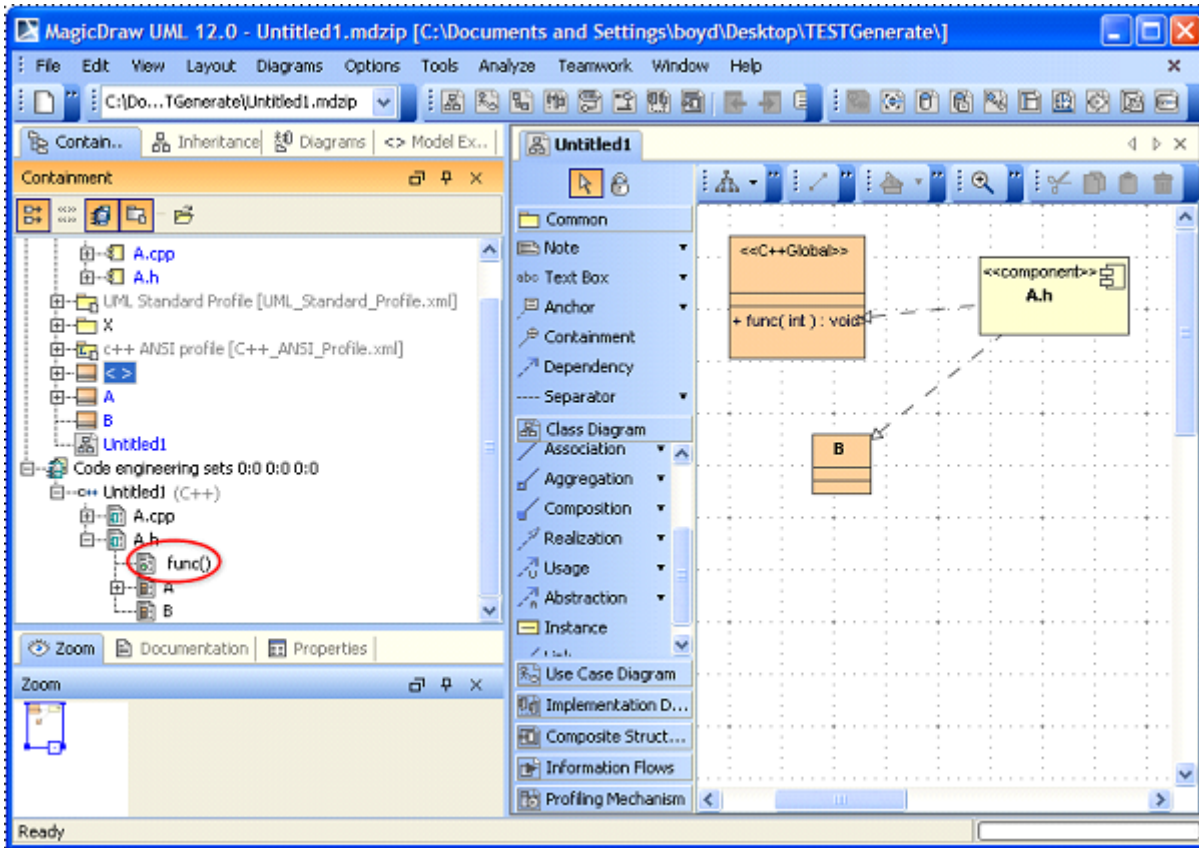


However, MagicDraw allows user to specify the component to generate the model into by using Realization. See the figure below for better understanding.



When drawing the Realization from component A.h to class B. Class B will be added to code engineering set as you can see in the red circle, which means that class B will be generated in A.h.

You can also specify component to generate global member as shown in the figure below.



## @see support for import-code-only mode

Documentation with doxygne @see tag is reversed with a link to the model. The documentation should use one of the following special doxygen tag to describe an @see element.

- @see
- \see
- @sa
- \sa

Example:

```
/**
 * Sample class
 */
class Sample {};

/**
 * @see Sample
 * \see Sample
 * @sa Sample
 * \sa Sample
 */
class Sample2: Sample {};
```

The screenshot displays the MagicDraw interface. On the left, the source code for the 'Sample' and 'Sample2' classes is shown. On the right, the UML model is visualized, showing a class 'Sample' and a class 'Sample2' with a generalization relationship. The documentation for 'Sample2' is displayed, showing links for @see, \see, @sa, and \sa pointing to 'Sample'. Arrows indicate that the links in the documentation are reversed to point to the 'Sample' class in the model.

**Sample is reversed as a link to Sample class in the model.**

## Navigable short cut from model link in documentation

- **Ctrl + click**  
MagicDraw will open the specification dialog of the model that the link point to.
- **Ctrl + Alt + click**  
MagicDraw will navigate to the documentation of the model that the link point to.

## Project constraint

This chapter describes how to set **include paths** for include **system file**.

```

1 // TestWinRoundtrip.cpp : Defines the entry point for the console application.
2 //
3
4 #include "stdafx.h"
5 #include <iostream>
6
7 class A {
8 public:
9     void print() {
10         std::cout << "Hello World" << std::endl;
11     }
12 };
13
14 int main(int argc, char* argv[])
15 {
16     A* myA = new A();
17
18     myA->print();
19
20     return 0;
21 }

```

← include system file

Figure 125 -- Include system file example

If you include any system file, you need to set system file path as well.

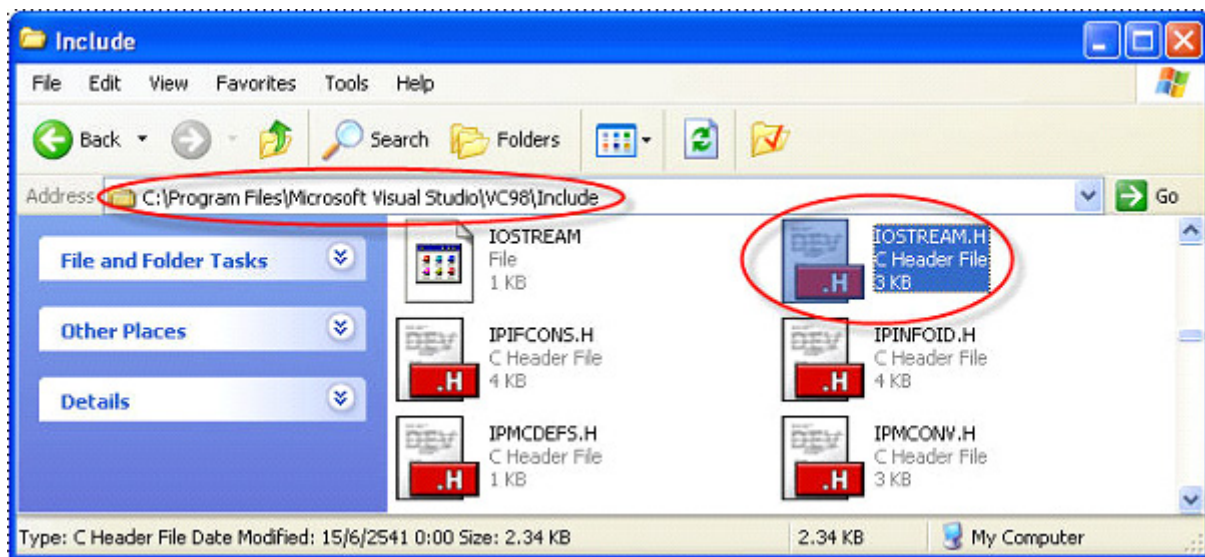


Figure 126 -- System file paths

### To set include path

1. From the **Options** menu choose **Project**.
2. Expand **Code Engineering**, choose **C++ Language Options**. Select the Include path checkbox. **Use include paths** button (...) appears on the right side of the Project options dialog box. Click this button.
3. The **Set Include Path** dialog appears.



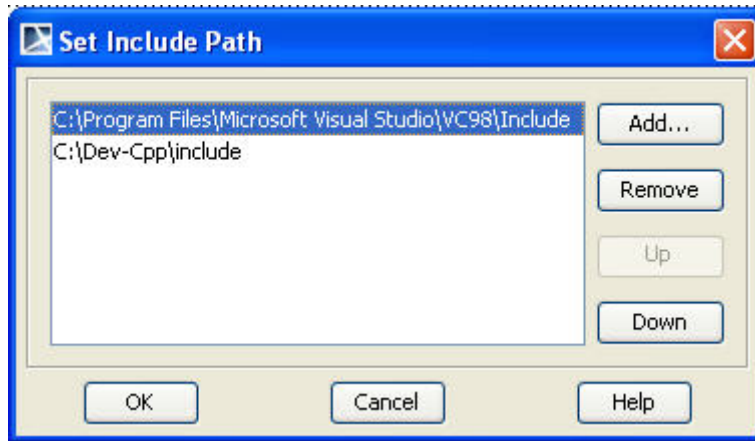
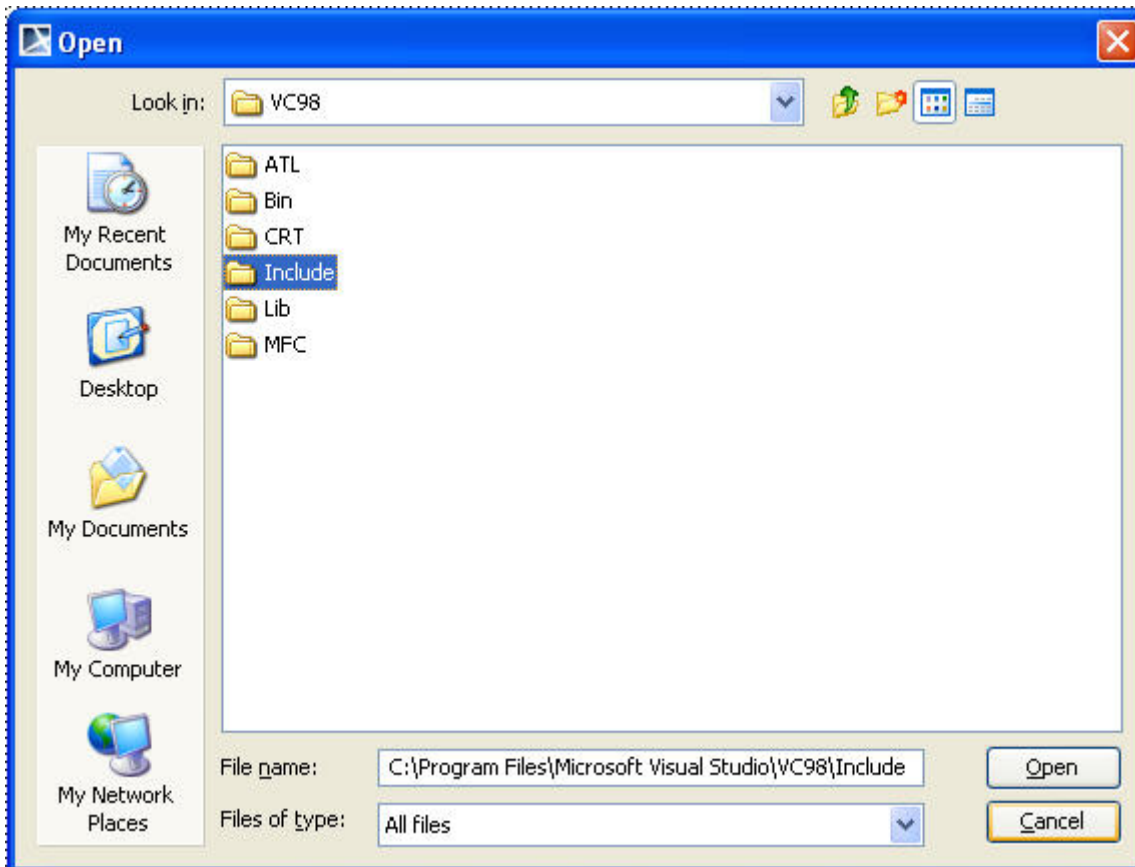


Figure 127 -- Set Include Path dialog box

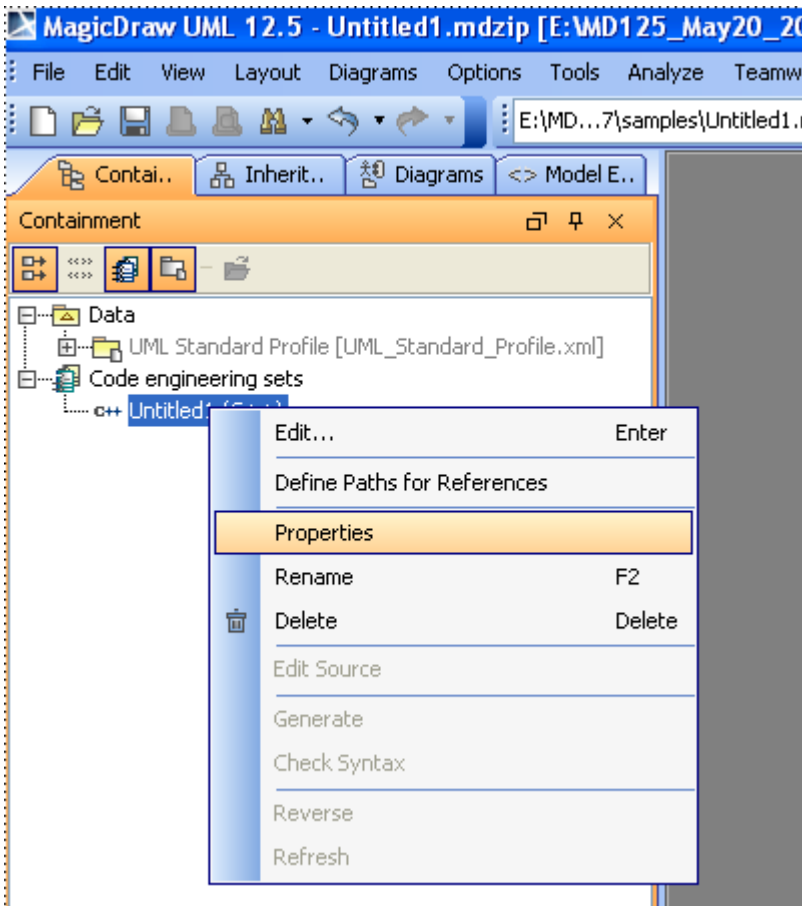
Button	Action
Add	The <b>Open</b> dialog box appears for select directory path.



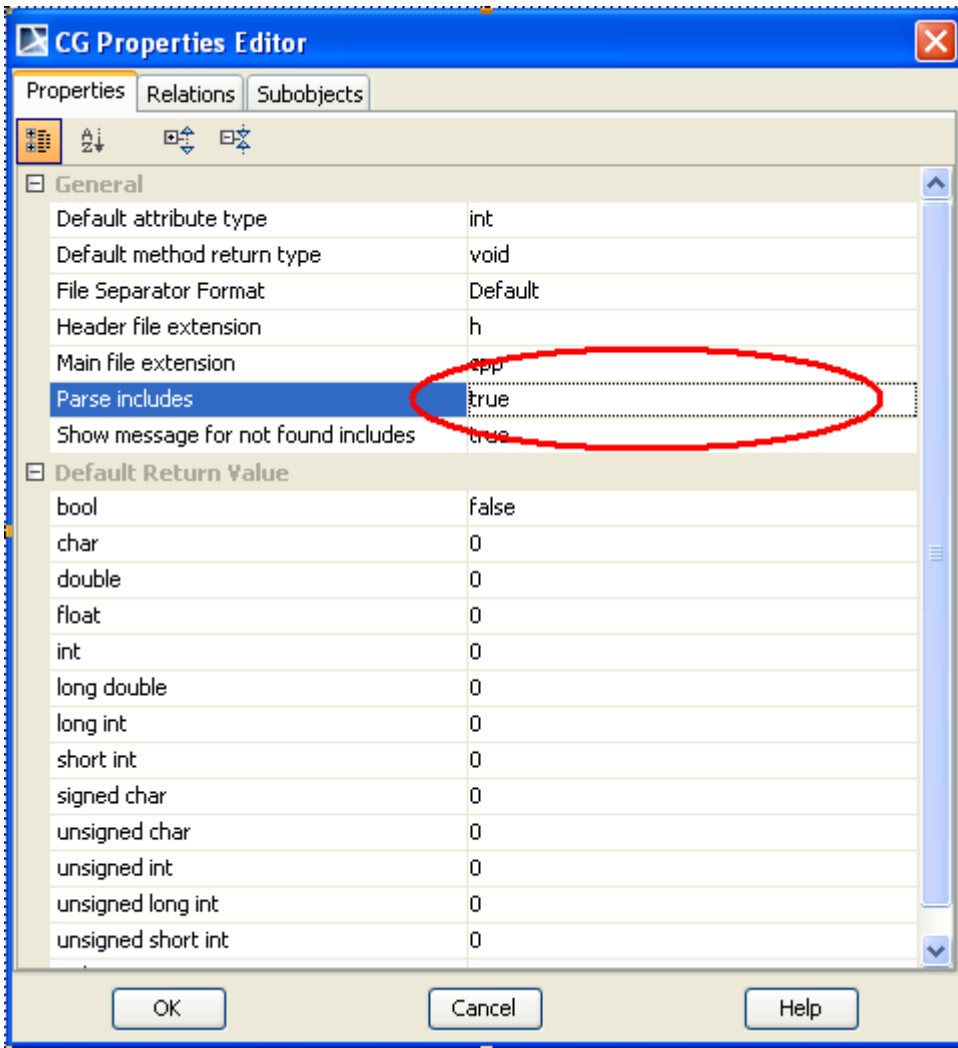
<b>Remove</b>	Remove the selected Include path.
<b>Up</b>	Move the selected Include path upward.
<b>Down</b>	Move the selected Include path downward.

Then select the code-engineering set and then select **Properties**.





Finally set the “Parse includes” property to “true”.

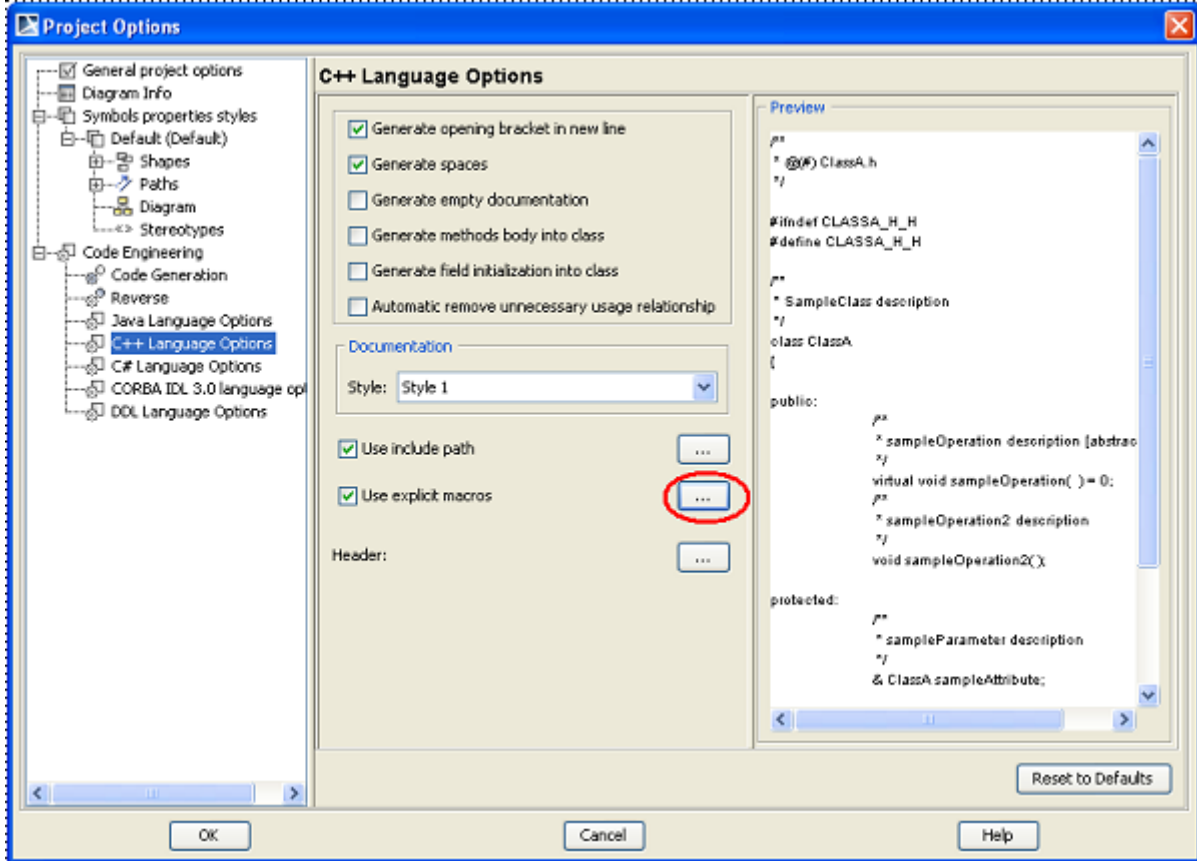


**Note**

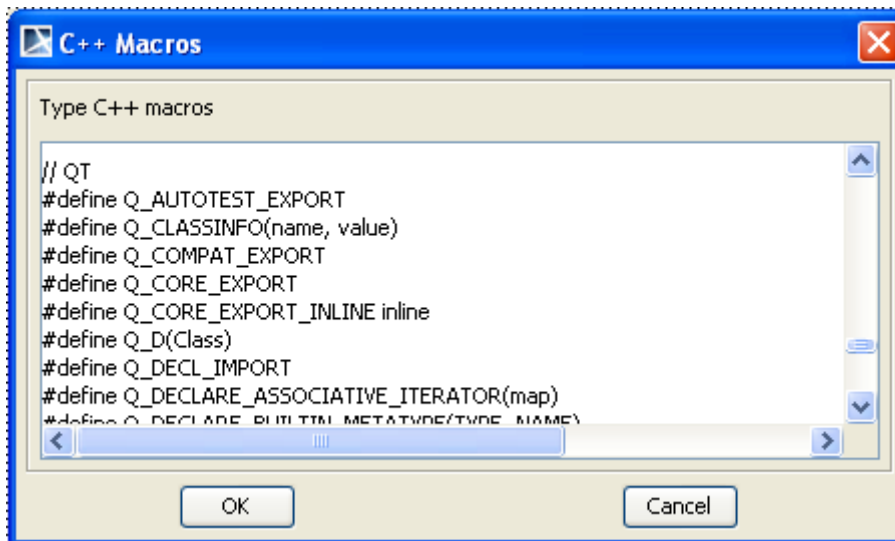
If the user sets the “Parse Includes” property to “false”, then during reverse engineering of the C++ code, MagicDraw will not parse inside the header file and all the unknown type will be created in the “Default” package of the model. This way the user can solve some keyword specific problems in the library, speed up the reverse process, and remove unnecessary model inside the header file.

## Working with QT

To reverse engineer QT code, we recommend you set **Parse Includes** option to "false". You will also need to set MagicDraw preprocessor to skip some QT macros ("Options"-> "Project"-> "C++ Language Options"-> "Use explicit macros").



MagicDraw has already defined some default skipped macros for QT.



## Microsoft C++ Profiles

Support for Microsoft specific structure and keywords, MagicDraw will provide new profiles named “Microsoft Visual C++ Profile”, “C++/CLI Profile”, and “C++ Managed Profile”.

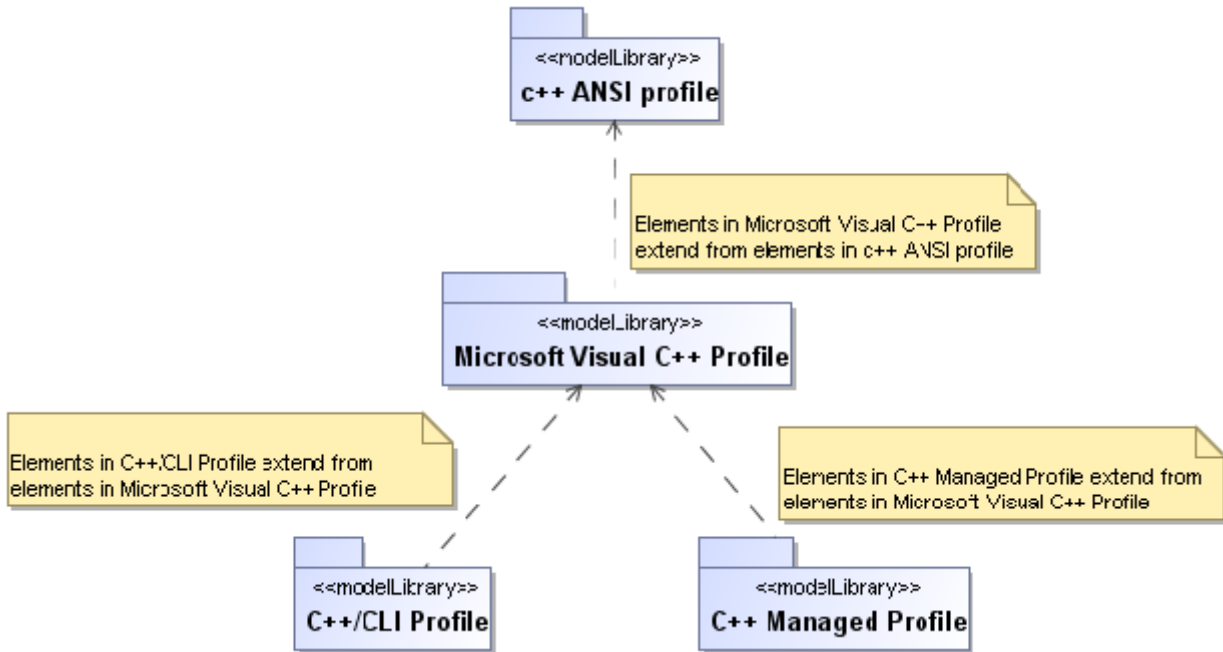


Figure 128 -- Profile Dependency

Microsoft Visual C++ Profile provides stereotypes and tagged values for modeling Microsoft Visual C++, which has additional specific keywords and constructs. It conforms to Microsoft Visual C++ 6.0 and later. However, it does not include keywords and constructs related to Managed features.

**Note** Microsoft Visual C++ with Managed features has been called Managed C++, Visual Studio 2003. Later in Visual Studio 2005, the set of keywords and constructs related to Managed features changed and is also known as C++/CLI.

### Microsoft Visual C++ Profile

Profile Name: **Microsoft Visual C++ Profile**

Module Name: **C++\_MS\_Profile.xml**

#### Data type

The Microsoft Visual C++ profile includes only the data types that do not exist in the ANSI C++ profile.

Microsoft C/C++ supports sized integer types. Declaration of 8-, 16-, 32-, or 64-bit integer variables can be done by using the `__intn` type specifier, where *n* is 8, 16, 32, or 64.

The types `__int8`, `__int16`, and `__int32` are synonyms for the ANSI types that have the same size, and are useful for writing portable code that behaves identically across multiple platforms.



Figure 129 -- Microsoft Visual C++ Data Type

## Stereotype

**NOTE** The profile table and description in this section does not include the tagged value inherited from C++ ANSI profile.

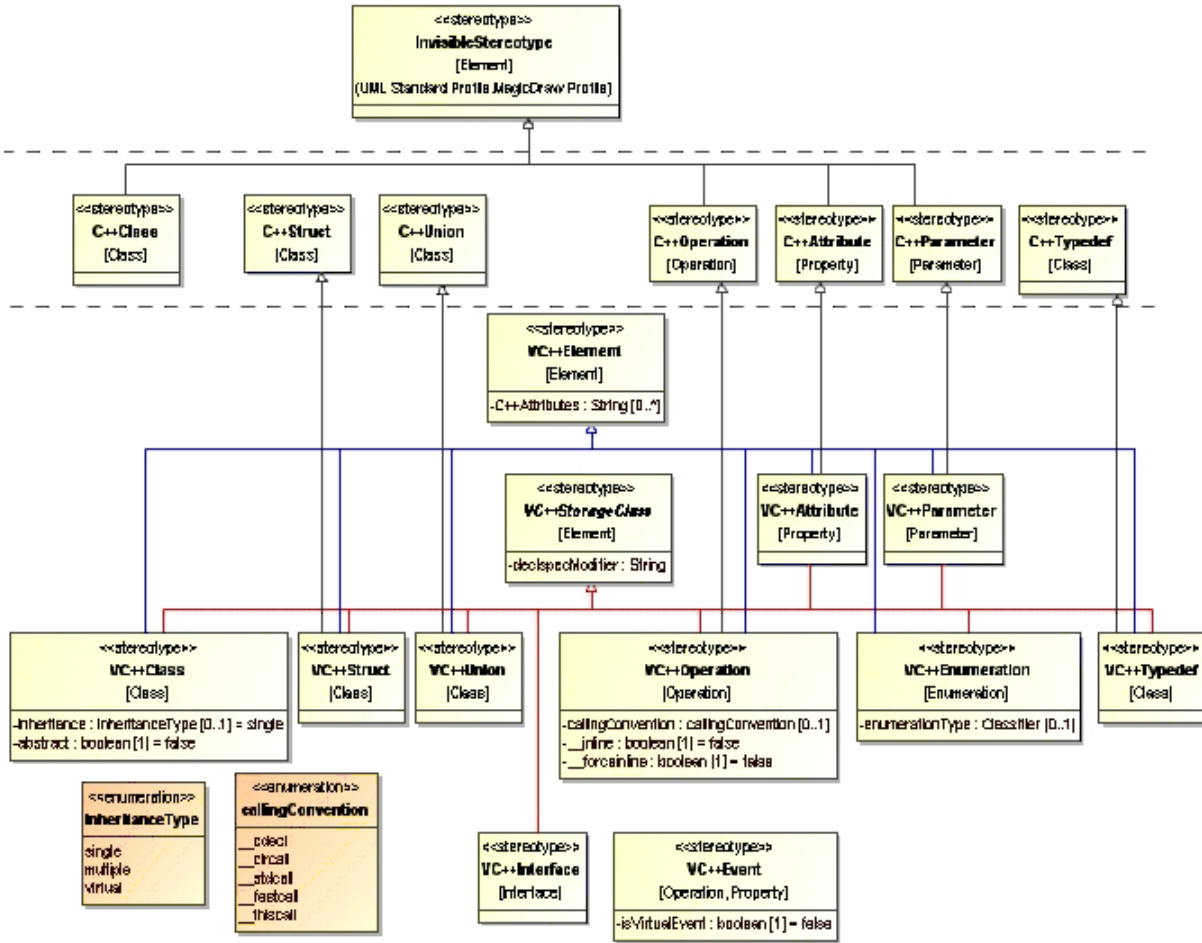


Figure 130 -- Microsoft Visual C++ Stereotypes

## VC++Class

<<VC++Class>> inherits from <<C++Class>> and <<VC++StorageClass>>

Name	Meta class	Constraints
VC++Class	Class	

Tag	Type	Description
inheritance	inheritanceType[0..1] = single (Enumeration) See <i>inheritanceType</i>	Represent the utilization of VC++ keywords, <b>__single_inheritance</b> , <b>__multiple_inheritance</b> and <b>__virtual_inheritance</b> .
abstract	boolean[1] = false	Represent the utilization of key-word <b>__abstract</b> or <b>abstract</b> , depending on C++ dialects.

Inherited tag	Type	Description
declspecModifier <<VC++StorageClass>>	String	For keeping extended declaration modifier of <b>__declspec</b> See <i>Extended storage-class attributes with __declspec</i>
C++Attributes <<VC++Element>>	String	For keeping C++ Attributes

## VC++Struct

<<VC++Struct>> inherits from <<C++Struct>> and <<VC++StorageClass>>

Name	Meta class	Constraints
VC++Struct	Class	

Inherited tag	Type	Description
declspecModifier <<VC++StorageClass>>	String	For keeping extended declaration modifier of <b>__declspec</b> See <i>Extended storage-class attributes with __declspec</i>
C++Attributes <<VC++Element>>	String	For keeping C++ Attributes



### VC++Union

Name	Meta class	Constraints
VC++Union	Class	

Inherited tag	Type	Description
declspecModifier <<VC++StorageClass>>	String	For keeping extended declaration modifier of <b>__declspec</b> See <i>Extended storage-class attributes with __declspec</i>
C++Attributes <<VC++Element>>	String	For keeping C++ Attributes

### VC++Enumeration

Name	Meta class	Constraints
VC++Enumeration	Enumeration	

Tag	Type	Description
Type	Classifier[0..1]	Represent the type of enumeration literal

### VC++Typedef

<<VC++Typedef>> inherits from <<C++Typedef>> and <<VC++StorageClass>>

Name	Meta class	Constraints
VC++Typedef	Class	

Inherited tag	Type	Description
C++Attributes <<VC++Element>>	String	For keeping C++ Attributes

## VC++Attribute

Name	Meta class	Constraints
VC++Attribute	Property	

Inherited tag	Type	Description
C++Attributes <<VC++Element>>	String	For keeping C++ Attributes

## VC++Operation

Name	Meta class	Constraints
VC++Operation	Operation	

Tag	Type	Description
callingConvention	callingConvention [0..1] (Enumeration) See <a href="#">callingConvention</a>	Represent the utilization of key-words. <ul style="list-style-type: none"> <li>• <code>__cdecl</code></li> <li>• <code>__clrcall</code></li> <li>• <code>__stdcall</code></li> <li>• <code>__fastcall</code></li> <li>• <code>__thiscall</code></li> </ul>
<code>__inline</code>	boolean[1] = false	Represent the utilization of <code>__inline</code> keyword.
<code>__forceinline</code>	boolean[1] = false	Represent the utilization of <code>__forceinline</code> keyword.

Inherited tag	Type	Description
declspecModifier <<VC++StorageClass>>	String	For keeping extended declaration modifier of <code>__declspec</code> See <a href="#">Extended storage-class attributes with __declspec</a>
C++Attributes <<VC++Element>>	String	For keeping C++ Attributes

## VC++Parameter

Name	Meta class	Constraints
VC++Parameter	Parameter	

Inherited tag	Type	Description
C++Attributes <<VC++Element>>	String	For keeping C++ Attributes

## VC++StorageClass

<<VC++StorageClass>> is an abstract stereotype corresponding to extended declaration modifier of **\_\_declspec**.

Name	Meta class	Constraints
VC++StorageClass	Element	

Tag	Type	Description
declspecModifier	String	For keeping extended declaration modifier of <b>__declspec</b> See <i>Extended storage-class attributes with __declspec</i>

## VC++Element

Name	Meta class	Constraints
VC++Element	Element	

Tag	Type	Description
C++Attributes	String	For keeping C++ Attributes

## VC++Interface

Name	Meta class	Description
VC++Interface	Interface	Represent interface declaration with <b>__interface</b> keyword.

## VC++Event

Name	Meta class	Description
VC++Event	Operation, Property	Represent interface, method or member declaration with <b>__event</b> keyword

Tag	Type	Description
isVirtualEvent	boolean[1] = false	Support virtual event as shown in the example below. // data member as event virtual <b>__event</b> ClickEventHandler* OnClick;

## Enumeration

### inheritanceType

**inheritanceType** will be used as value of tag named “**inheritance**” under <<VC++Class>>.

Literal	Description
single	Representation of <b>__single_inheritance</b> keyword
multiple	Representation of <b>__multiple_inheritance</b> keyword
Virtual	Representation of <b>__virtual_inheritance</b> keyword

### callingConvention

**callingConvention** will be used as value of tag named “**calling convention**” under <<VC++Operation>>

Literal	Description
<b>__cdecl</b>	Represent the utilization of <b>__cdecl</b> modifier.
<b>__cdecl</b>	Represent the utilization of <b>__cdecl</b> modifier.
<b>__stdcall</b>	Represent the utilization of <b>__stdcall</b> modifier.
<b>__fastcall</b>	Represent the utilization of <b>__fastcall</b> modifier.
<b>__thiscall</b>	Represent the utilization of <b>__thiscall</b> modifier.

## C++/CLI Profile

**Note** The profile table and description in this section does not include the tagged value inherited from C++ ANSI profile.

### Stereotype

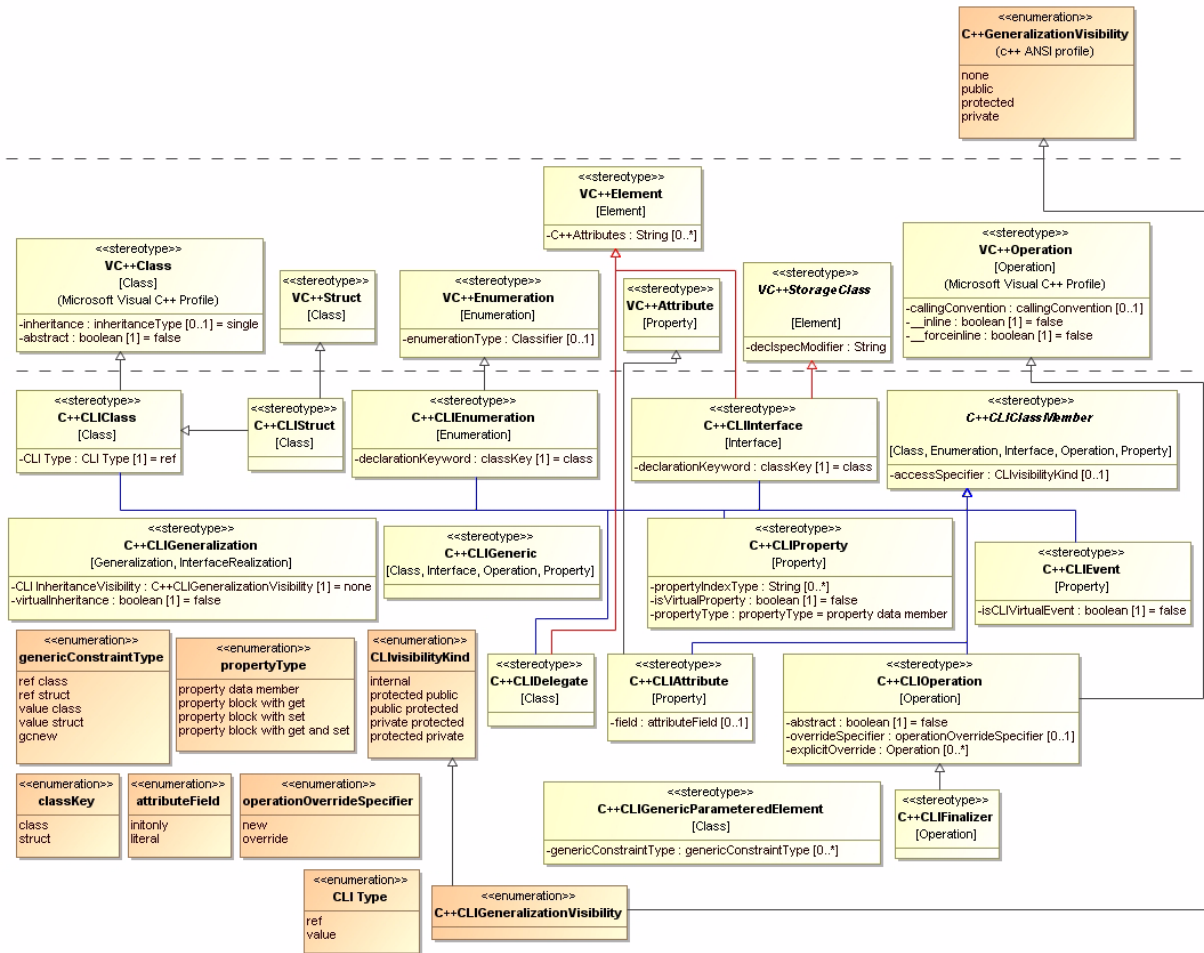


Figure 131 -- C++/CLI Stereotypes

### C++CLIClass

Name	Meta class	Constraints
C++CLIClass	Class	
Tag	Type	Description
CLI Type	CLI Type[1] = ref (Enumeration) See <i>CLI Type</i>	Represent the utilization of <b>ref class</b> or <b>value class</b> keywords for defining CLR class.

Inherited tag	Type	Description
inheritance <<VC++Class>>	inheritanceType[0..1] = single (Enumeration) See <a href="#">inheritanceType</a>	Represent the utilization of VC++ keywords, <b>__single_inheritance</b> , <b>__multiple_inheritance</b> and <b>__virtual_inheritance</b> .
declspecModifier <<VC++StorageClass>>	String	For keeping extended declaration modifier of <b>__declspec</b> See <a href="#">Extended storage-class attributes with __declspec</a>
C++Attributes <<VC++Element>>	String	For keeping C++ Attributes
abstract <<VC++Class>>	boolean[1] = false	Represent the utilization of <b>abstract</b> keyword.
accessSpecifier <<C++CLIClassMember>>	CLIVisibilityKind[0..1] (Enumeration) See <a href="#">CLIVisibilityKind</a>	Represent the usage of visibility kind introduced in C++/CLI including <ul style="list-style-type: none"> <li>• internal</li> <li>• protected public</li> <li>• public protected</li> <li>• private protected</li> <li>• protected private</li> </ul>

## C++CLIStruct

Name	Meta class	Constraints
C++CLIStruct	Class	

Inherited tag	Type	Description
managedType <<C++CLIClass>>	CLI Type[1] = ref (Enumeration) See <a href="#">CLI Type</a>	Represent the utilization of <b>ref class</b> or <b>value class</b> keywords for defining CLR class.
abstract <<C++CLIClass>>	boolean[1] = false	Represent the utilization of <b>abstract</b> keyword.
inheritance <<VC++Class>>	inheritanceType[0..1] = single (Enumeration) See <a href="#">inheritanceType</a>	Represent the utilization of VC++ keywords, <b>__single_inheritance</b> , <b>__multiple_inheritance</b> and <b>__virtual_inheritance</b> .
declspecModifier <<VC++StorageClass>>	String	For keeping extended declaration modifier of <b>__declspec</b> See <a href="#">Extended storage-class attributes with __declspec</a>

Inherited tag	Type	Description
VC++Attributes <<VC++Element>>	String	For keeping VC++ Attributes

## C++CLIEnumeration

Name	Meta class	Constraints
C++CLIEnumeration	Enumeration	

Tag	Type	Description
declarationKeyword	classKey[1] = class (Enumeration) See <a href="#">classKey</a>	Represent the utilization of <b>enum class</b> or <b>enum struct</b> . keywords. See <a href="#">enum class</a> , <a href="#">enum struct</a>

Inherited tag	Type	Description
type <<VC++Enumeration>>	Classifier[0..1]	Represent the type of enumeration literal

## C++CLIInterface

Name	Meta class	Constraints
C++CLIInterface	Interface	

Tag	Type	Description
declarationKeyword	classKey[1] = class (Enumeration) See <a href="#">classKey</a>	Represent the utilization of <b>interface class</b> or <b>interface struct</b> . keywords. See <a href="#">ref class</a> , <a href="#">ref struct</a> , <a href="#">value class</a> , <a href="#">value struct</a> , <a href="#">interface class</a> , <a href="#">interface struct</a>

Inherited tag	Type	Description
declspecModifier <<VC++StorageClass>>	String	For keeping extended declaration modifier of <b>__declspec</b> See <a href="#">Extended storage-class attributes with __declspec</a>
C++Attributes <<VC++Element>>	String	For keeping C++ Attributes



## C++CLIClassMember

Name	Meta class	Constraints
C++CLIClassMember	Operation, Property	This stereotype is an abstract stereotype for keeping accessSpecifier tag definition.

Tag	Type	Description
accessSpecifier	CLLvisibilityKind[0..1] (Enumeration) See <a href="#">CLLvisibilityKind</a>	Represent the usage of visibility kind introduce in C++/CLI including <ul style="list-style-type: none"> <li>• internal</li> <li>• protected public</li> <li>• public protected</li> <li>• private protected</li> <li>• protected private</li> </ul>

## C++CLIAttribute

Name	Meta class	Constraints
C++CLIAttribute	Property	

Tag	Type	Description
field	attributeField[0..1] (Enumeration) See <a href="#">attributeField</a>	Represent the usage of <b>initonly</b> or <b>literal</b> keywords.

Inherited tag	Type	Description
accessSpecifier << C++CLIClassMember >>	CLLvisibilityKind[0..1] (Enumeration) See <a href="#">CLLvisibilityKind</a>	Represent the usage of visibility kind introduce in C++/CLI including <ul style="list-style-type: none"> <li>• internal</li> <li>• protected public</li> <li>• public protected</li> <li>• private protected</li> <li>• protected private</li> </ul>
declspecModifier <<VC++StorageClass >>	String	For keeping extended declaration modifier of <b>__declspec</b> See <a href="#">Extended storage-class attributes with __declspec</a>
C++Attributes <<VC++Element >>	String	For keeping C++ Attributes

## C++CLIOperation

Name	Meta class	Constraints
C++CLIOperation	Operation	

Tag	Type	Description
abstract	boolean[1] = false	Represent the utilization of <b>abstract</b> keyword.
overrideSpecifier	operationOverrideSpecifier[0..1] (Enumeration) See <a href="#">Override Specifiers</a>	Represent the utilization of C++/CLI operation override specifiers, <b>new</b> and <b>override</b> .
explicitOverride	Operation[0..*]	Represent the usage of explicit override feature.

Inherited tag	Type	Description
accessSpecifier << C++CLIClassMember >>	CLIVisibilityKind[0..1] (Enumeration) See <a href="#">CLIVisibilityKind</a>	Represent the usage of visibility kind introduce in C++/CLI including <ul style="list-style-type: none"> <li>• internal</li> <li>• protected public</li> <li>• public protected</li> <li>• private protected</li> <li>• protected private</li> </ul>
callingConvention <<VC++Operation >>	callingConvention[0..1] See <a href="#">callingConvention</a>	Represent the utilization of keywords. <ul style="list-style-type: none"> <li>• <code>__cdecl</code></li> <li>• <code>__stdcall</code></li> <li>• <code>__fastcall</code></li> <li>• <code>__thiscall</code></li> </ul>
C++Attributes <<VC++Element >>	String	For keeping C++ Attributes

## C++CLIProperty

<<C++CLIProperty>> is used to define a CLR property, which has the appearance of an ordinary data member, and can be written to or read from using the same syntax as a data member.

Name	Meta class	Constraints
C++CLIProperty	Property	The stereotype must not be applied to the same attribute as <<C++Attribute>> or other stereotypes derived from <<C++Attribute>>

Tag	Type	Description
propertyIndexType	String[0..*]	index type keeps the list of property index types.
isVirtualProperty	boolean[1] = false	Specify whether the property is virtual or not.
propertyType	PropertyType[1] = property data member	Specify property type between <ul style="list-style-type: none"> <li>property data member</li> <li>property block with get</li> <li>property block with set</li> <li>property block with get and set</li> </ul>

Inherited tag	Type	Description
accessSpecifier << C++CLIClassMember>>	CLIVisibilityKind [0..1] (Enumeration) See <a href="#">CLIVisibilityKind</a>	Represent the usage of visibility kind introduce in C++/CLI including <ul style="list-style-type: none"> <li>internal</li> <li>protected public</li> <li>public protected</li> <li>private protected</li> <li>protected private</li> </ul>

## C++CLIDelegate

<<C++CLIDelegate>> is used to define a delegate, which is a reference type that can encapsulate one or more methods with a specific function prototype. Delegates provide the underlying mechanism (acting as a kind of pointer to member function) for events in the common language runtime component model.

Name	Meta class	Constraints
C++CLIDelegate	Class	Declaration of delegate can be only in a managed type. The declaration of delegate cannot have attribute and operation. Represent the utilization of <b>delegate</b> keyword.

## C++CLIEvent

Name	Meta class	Constraints
C++CLIEvent	Property	<<C++CLIEvent>> can be applied only when the attribute type is <<C++CLIDelegate>>. Represent the utilization of <b>event</b> keyword.
Tag	Type	Description
isCLIVirtualEvent	boolean[1] = false	Specify whether the event is virtual or not.
Inherited tag	Type	Description
accessSpecifier << C++CLIClassMember>>	CLlvisibilityKind[0..1] (Enumeration) See <a href="#">CLlvisibilityKind</a>	Represent the usage of visibility kind introduce in C++/CLI including <ul style="list-style-type: none"> <li>• internal</li> <li>• protected public</li> <li>• public protected</li> <li>• private protected</li> <li>• protected private</li> </ul>

## C++CLIGeneric

Name	Meta class	Constraints
C++CLIGeneric	Class, Interface, Operation, Property	Represent the usage of <b>generic</b> keyword.

## C++CLIGenericParameteredElement

Name	Meta class	Constraints
C++CLIGenericParameteredElement	Class	
Tag	Type	Description
genericConstraintType	genericConstraintType[0..*] (Enumeration) See <a href="#">genericConstraintType</a>	Specify generic constraint type including <ul style="list-style-type: none"> <li>• ref class</li> <li>• ref struct</li> <li>• value class</li> <li>• value struct</li> <li>• gnew</li> </ul>

## C++CLIGeneralization

Name	Meta class	Constraints
C++CLIGeneralization	Generalization	
Tag	Type	Description
CLIIInheritanceVisibility	C++CLIGeneralizationVisibility[1] = none (Enumeration) See <a href="#">C++CLIGeneralization-Visibility</a>	
virtualInheritance	boolean[1] = false	

## Enumeration

### CLI Type

**CLI Type** will be used as value of tag named "**CLI Type**" under `<<C++CLIClass>>`. The possible values are **ref** and **value**.

Literal	Description
ref	For defining a CLR reference class
value	For defining a CLR value class

### operationOverrideSpecifier

**operationOverrideSpecifier** will be used as value of tag named "**overrideSpecifier**" under `<<VC++Operation>>`.

The possible values are **new** and **override** which are keywords that can be used to qualify override behavior for derivation.

Literal	Description
new	Indicate the use of <b>new</b> keyword to qualify override behavior for derivation. In VC++, <b>new</b> is a keyword to indicate that a virtual member will get a new slot in the vtable; that the function does not override a base class method.
override	Indicate the use of <b>override</b> keyword to qualify override behavior for derivation. In VC++, <b>override</b> is a keyword to indicate that a member of a managed type must override a base class or a base interface member. If there is no member to override, the compiler will generate an error.

#### attributeField

**attributeField** will be used as value of tag named "field" under <<C++CLIAttribute>>. The possible values are **initonly** and **literal** which are keywords that can be used to qualify field type of attribute.

Literal	Description
initonly	Represent the utilization of <b>initonly</b> keyword. <b>initonly</b> indicates that variable assignment can only occur as part of the declaration or in a static constructor in the same class.
literal	Represent the utilization of <b>literal</b> keyword. It is the native equivalent of <b>static const</b> variable. Constraint: <b>Is Static</b> and <b>Is Read Only</b> must be set to <b>true</b> .

#### classKey

Literal	Description
class	Represent the keyword that has the word, <b>class</b> .
struct	Represent the keyword that has the word, <b>struct</b> .

Remark: **enum class** and **enum struct** are equivalent declarations.

**interface class** and **interface struct** are equivalent declarations.

---

 CLlvisibilityKind
 

---

Literal	Description
internal	Represent the <b>internal</b> visibility.
protected public	Represent the <b>protected public</b> visibility.
public protected	Represent the <b>public protected</b> visibility.
private protected	Represent the <b>private protected</b> visibility.
protected private	Represent the <b>protected private</b> visibility.

---

 genericConstraintType
 

---

Literal	Description
ref class	Represent the usage of <b>ref class</b> keyword in generic constraint clause.
ref struct	Represent the usage of <b>ref struct</b> keyword in generic constraint clause.
value class	Represent the usage of <b>value class</b> keyword in generic constraint clause.
value struct	Represent the usage of <b>value struct</b> keyword in generic constraint clause.
gcnew	Represent the usage of <b>gcnew</b> keyword in generic constraint clause.

---

 C++CLIGeneralizationVisibility
 

---

C++CLIGeneralizationVisibility is an enumeration inheriting from C++GeneralizationVisibility and CLlvisibilityKind, so its literals include **none**, **public**, **protected**, **private**, **internal**, **protected public**, **public protected**, **private protected**, and **protected private**.

## C++ Managed Profile

**NOTE** The profile table and description in this section does not include the tagged value inherited from other profiles.

## Sterotype

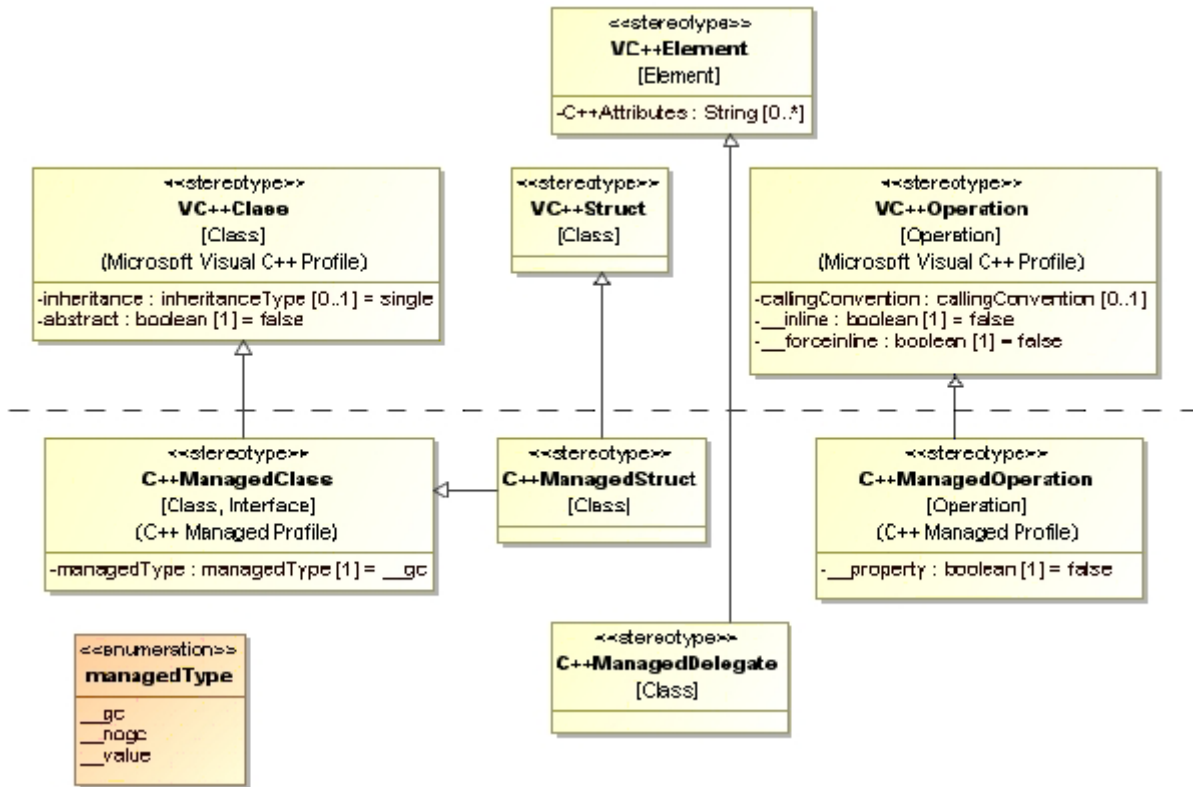


Figure 132 -- C++ Managed Stereotypes

### C++ManagedClass

Name	Meta class	Constraints
C++ManagedClass	Class	Represent the class declaration with version 1 of Managed Extension for C++.

Tag	Type	Description
managedType	managedType[1] = __gc (enumeration) See <a href="#">managedType</a>	

### C++ManagedStruct

Name	Meta class	Constraints
C++ManagedStruct	Class	Represent the struct declaration with version 1 of Managed Extension for C++.



## C++ManagedOperation

Name	Meta class	Constraints
C++ManagedOperation	Operation	

Tag	Type	Description
__property	boolean[1] = false	Represent the usage of <b>__property</b> keyword. It is a feature in version 1 of Managed Extension for C++.

## C++ManagedDelegate

Name	Meta class	Constraints
C++ManagedDelegate	Class	<p>Represent the delegate declaration with version 1 of Managed Extension for C++.</p> <p>It defines a reference type that can be used to encapsulate a method with a specific signature.</p>

## Enumeration

### managedType

Literal	Description
__gc	Represent managed declaration with <b>__gc</b> keyword.
__nogc	Represent the usage of <b>__nogc</b> keyword, which is used to explicitly specify that an object is allocated on the standard C++ heap.
__value	<p>Represent managed declaration with <b>__value</b> keyword.</p> <p>A <b>__value</b> type differs from <b>__gc</b> types in that <b>__value</b> type variables directly contain their data, whereas managed variables point to their data, which is stored on the common language runtime heap.</p>

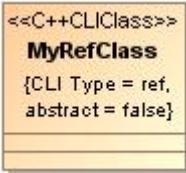
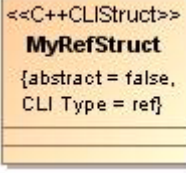
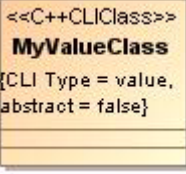
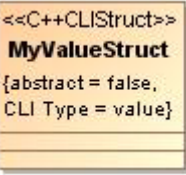


## Modeling with Microsoft Specific Profiles

### CLR Data Type Keyword

ref class, ref struct, value class, value struct, interface class, interface struct

```
class_access ref class name modifier : inherit_access base_type {};
class_access ref struct name modifier : inherit_access base_type {};
class_access value class name modifier : inherit_access base_type {};
class_access value struct name modifier : inherit_access base_type {};
interface_access interface class name : inherit_access base_interface {};
```

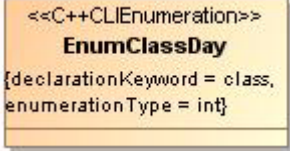
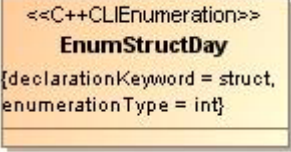
interface\_access interface struct name : inherit\_access base\_interface {};

Code	MD-UML
<pre>ref class MyRefClass { };</pre>	 <p>with &lt;&lt;C++CLIClass&gt;&gt; and tagged value <b>CLI Type = ref</b></p>
<pre>ref struct MyRefStruct { };</pre>	 <p>with &lt;&lt;C++CLIStruct&gt;&gt; and tagged value <b>CLI Type = ref</b></p>
<pre>value class MyValueClass { };</pre>	 <p>with &lt;&lt;C++CLIClass&gt;&gt; and tagged value <b>CLI Type = value</b></p>
<pre>value struct MyValueStruct { };</pre>	 <p>with &lt;&lt;C++CLIStruct&gt;&gt; and tagged value <b>CLI Type = value</b></p>
<pre>interface class MyInterfaceClass { };</pre>	 <p>with &lt;&lt;C++CLIInterface&gt;&gt; and tagged value <b>declarationKeyword = class</b></p>
<pre>interface struct MyInterfaceStruct { };</pre>	 <p>with &lt;&lt;C++CLIInterface&gt;&gt; and tagged value <b>declarationKeyword = struct</b></p>

enum class, enum struct

access **enum class** name [: type] { enumerator-list } var;

access **enum struct** name [:type] { enumerator-list } var;

Code	MD-UML
<pre>enum class EnumClassDay :int {     sun,     mon };</pre>	
<pre>enum struct EnumStructDay :int {     sun,     mon };</pre>	

### property

modifier **property** type property\_name; // property data member

modifier **property** type property\_name { // property block

modifier void **set**(type);

modifier type **get**();

}

modifier **property** type property\_name[,] {

modifier void **set**(type);

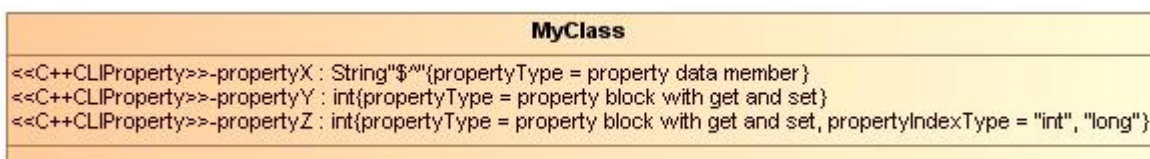
modifier type **get**();

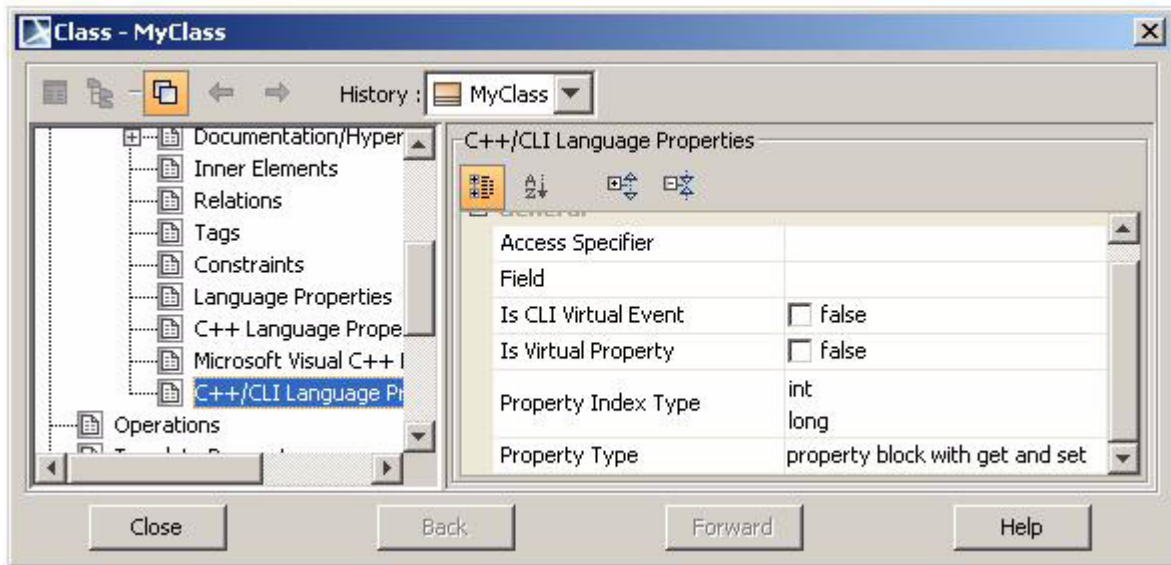
}

### Code

```
public ref class MyClass
{
    // property data memberproperty
    property String ^ propertyX;
    // property block
    property int propertyY
    {
        int get();
        void set(int value);
    }
    //property block with index
    property int propertyZ[int, long]
    {
        int get(int index1,long index2);
        void set(int index1,long index2, int value);
    }
};
```

### MD-UML





## delegate

---

access **delegate** function\_declaration

access (optional)

The accessibility of the delegate outside of the assembly can be public or private. The default is private. Inside a class, a delegate can have any accessibility.

function\_declaration

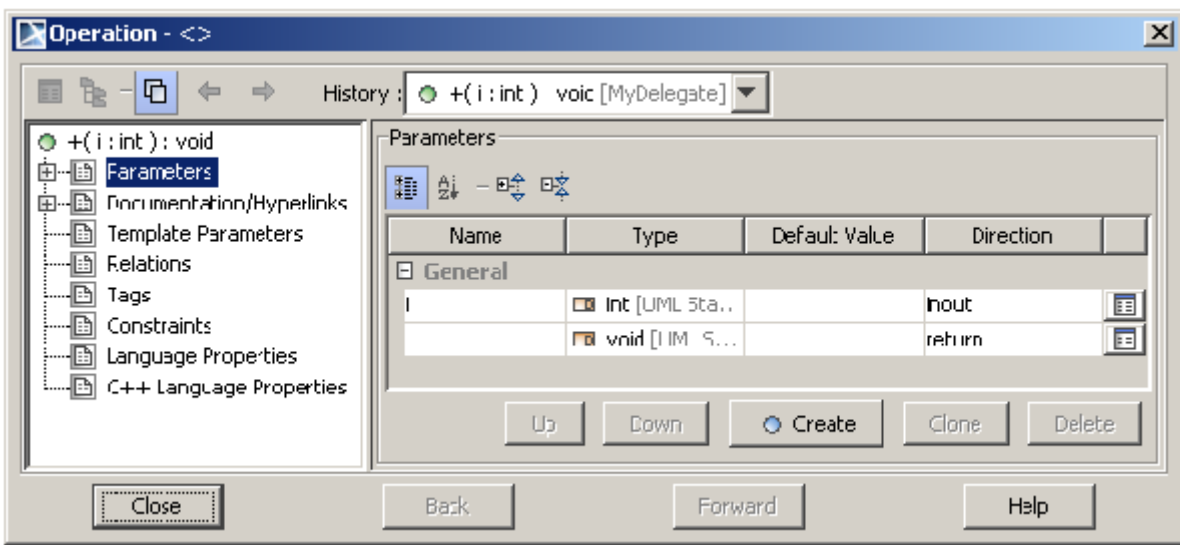
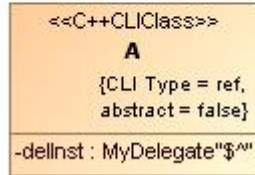
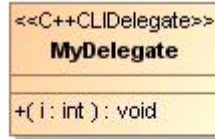
The signature of the function that can be bound to the delegate. The return type of a delegate can be any managed type. For interoperability reasons, it is recommended that the return type of a delegate be a CLS type.

To define an unbound delegate, the first parameter in function\_declaration should be the type of the this pointer for the object

**Code**

```
public delegate void MyDelegate(int
i);
ref class A
{
    MyDelegate^ delInst;
};
```

**MD-UML**



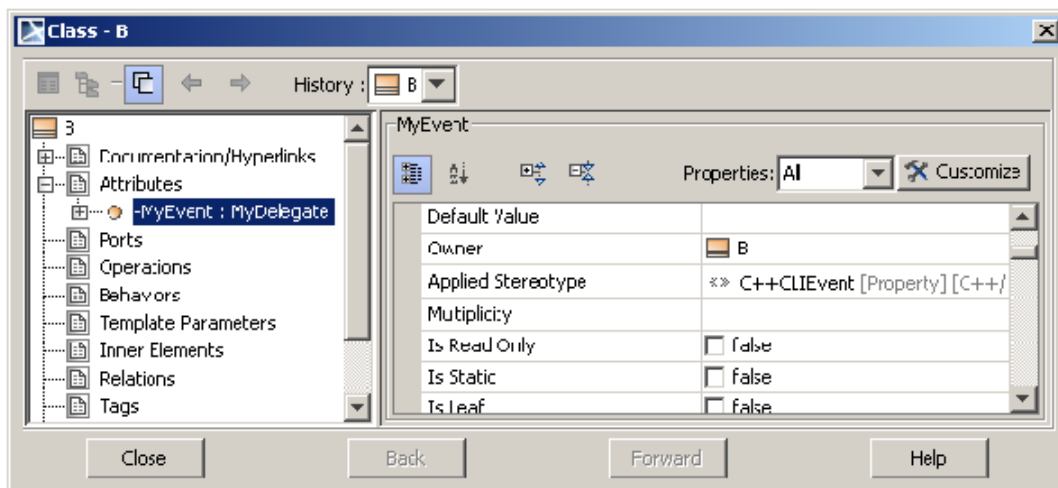
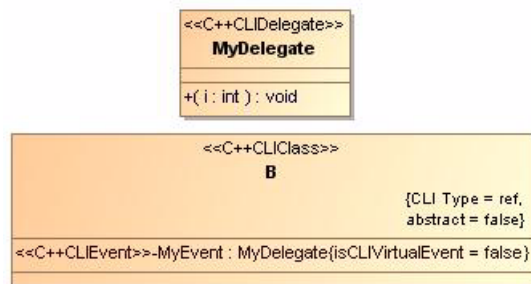
### event

modifier **event** delegate ^ event\_name; // event data member

modifier **event** delegate ^ event\_name

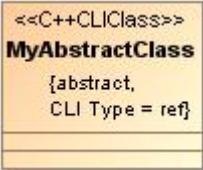
```
{
    modifier return_value add (delegate ^ name);
    modifier return_value remove(delegate ^ name);
    modifier return_value raise(parameters);
}
```

```
public delegate void
MyDelegate(int);
ref class B {
    event MyDelegate^ MyEvent;
};
```

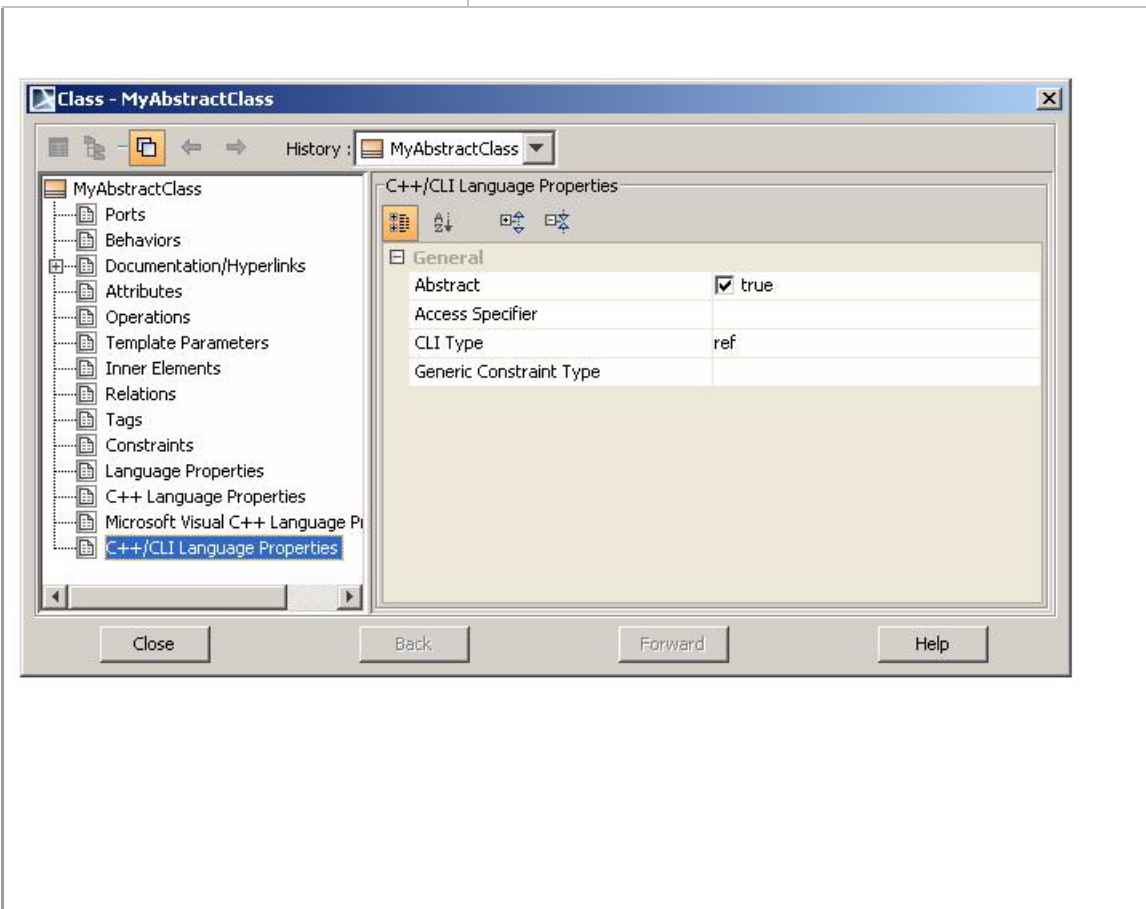


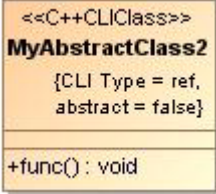
## Override Specifiers

abstract

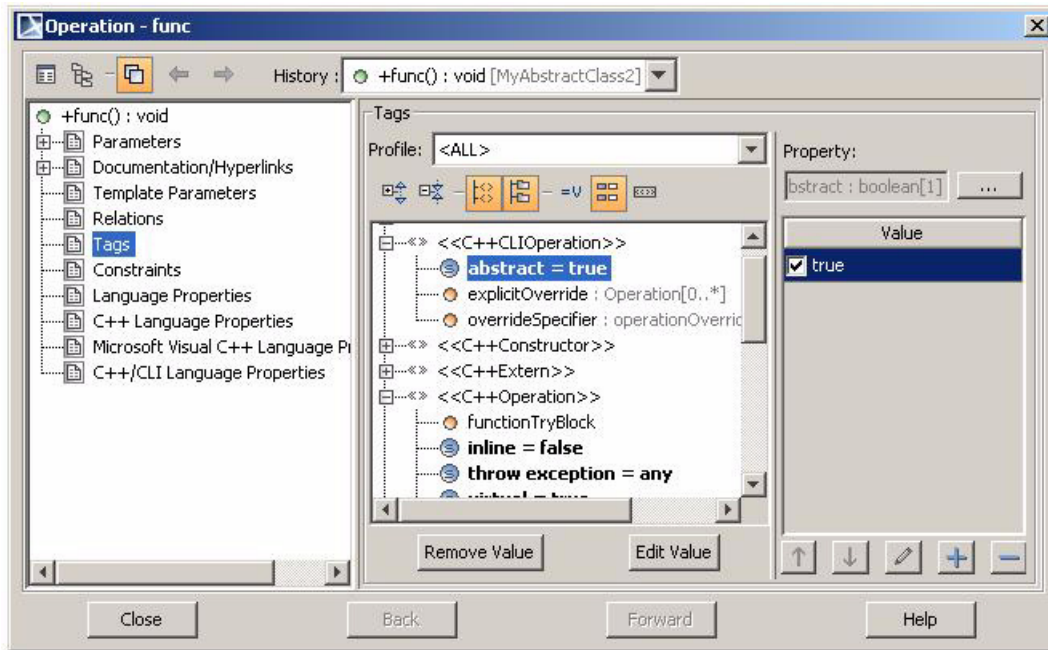
Code	MD-UML
<pre>ref class MyAbstractClass abstract { };</pre>	

with <<C++CLIClass>> and tagged value **abstract**=true



Code	MD-UML
<pre>ref class MyAbstractClass2 { public: virtual void func() abstract; };</pre>	

func() with <<C++CLIOperation>> and tagged value **abstract** = true and **virtual** = true



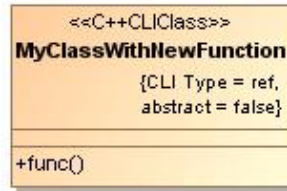


new

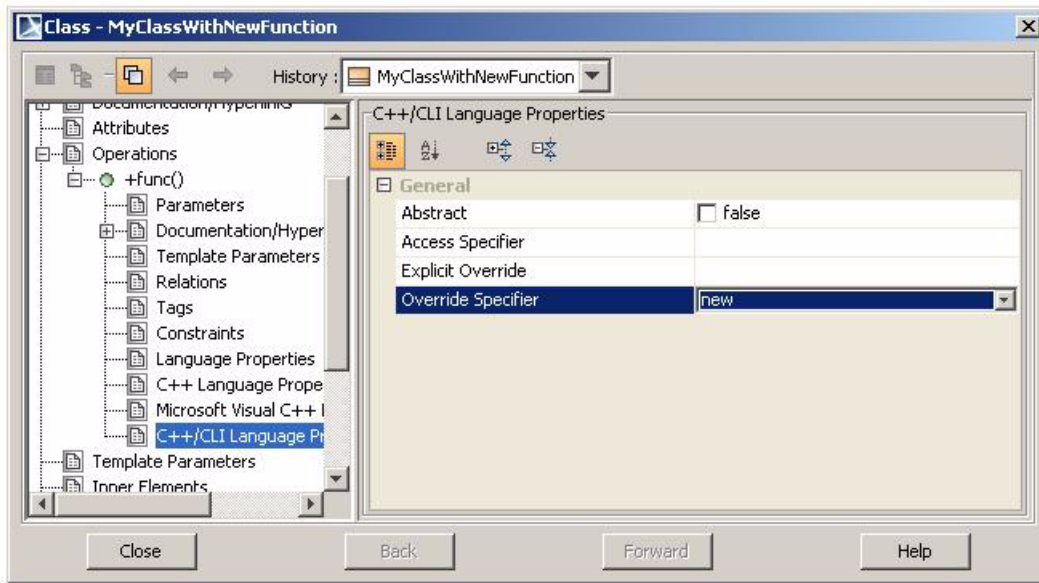
Code

```
ref class MyClassWithNewFunction
{
public:
    virtual void func() new {}
};
```

MD-UML



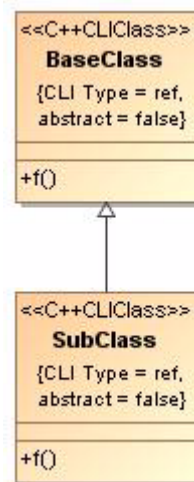
func() with <<C++CLIOperation>> and tagged value **overrideSpecifier** = new

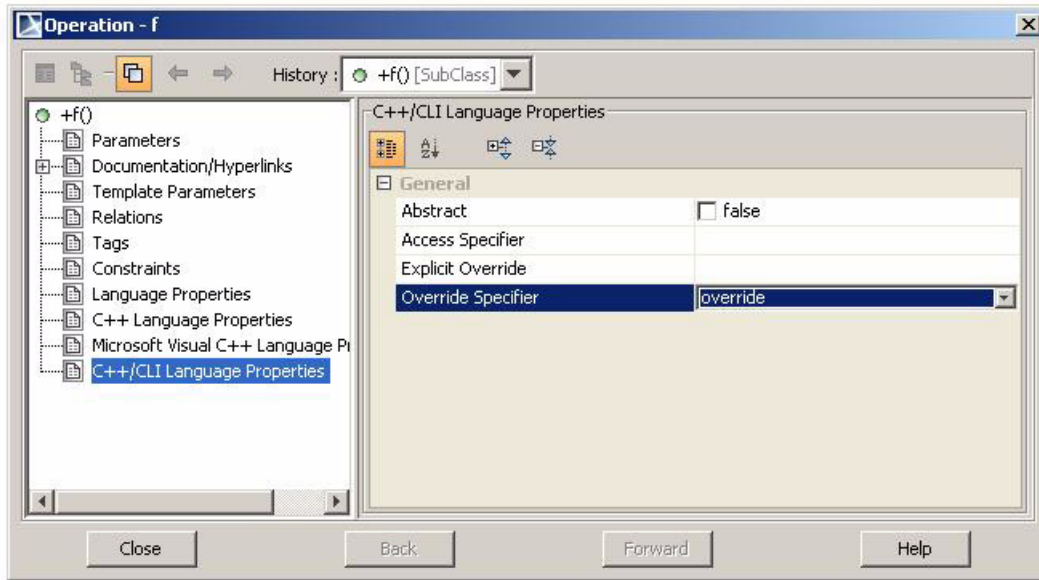


override

```
ref class BaseClass {
public:
    virtual void f();
};

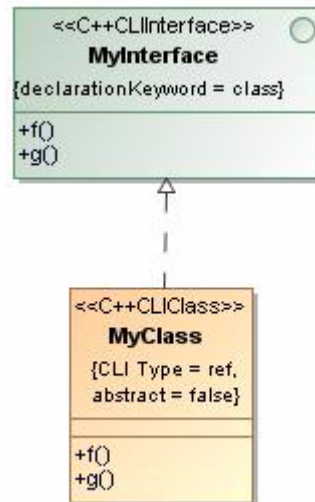
ref class SubClass : public
BaseClass {
public:
    virtual void f() override {}
};
```

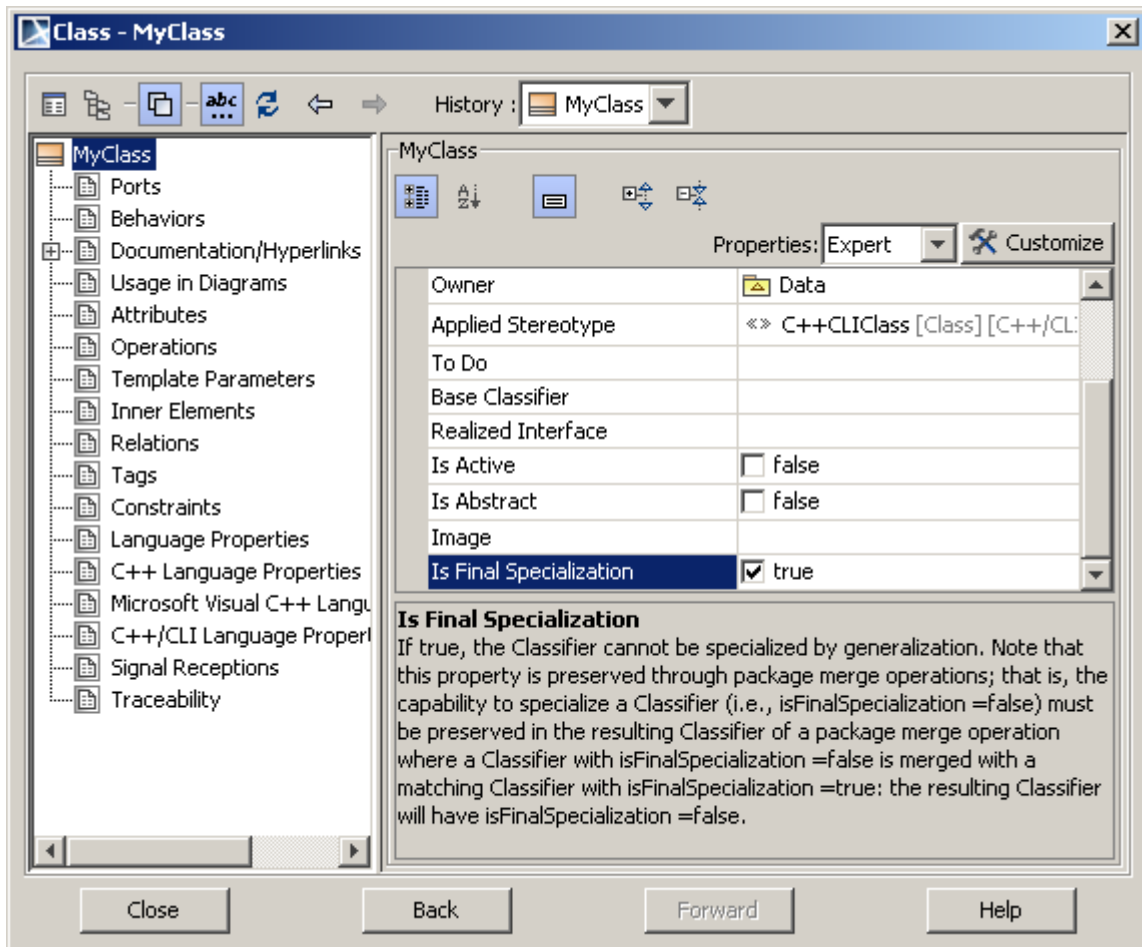




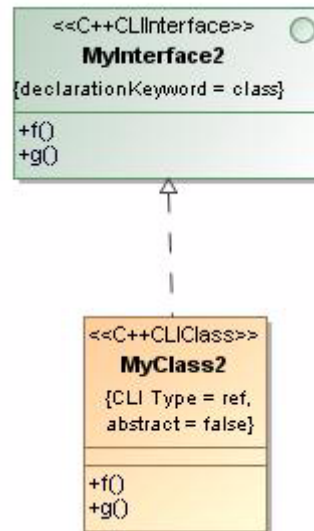
sealed

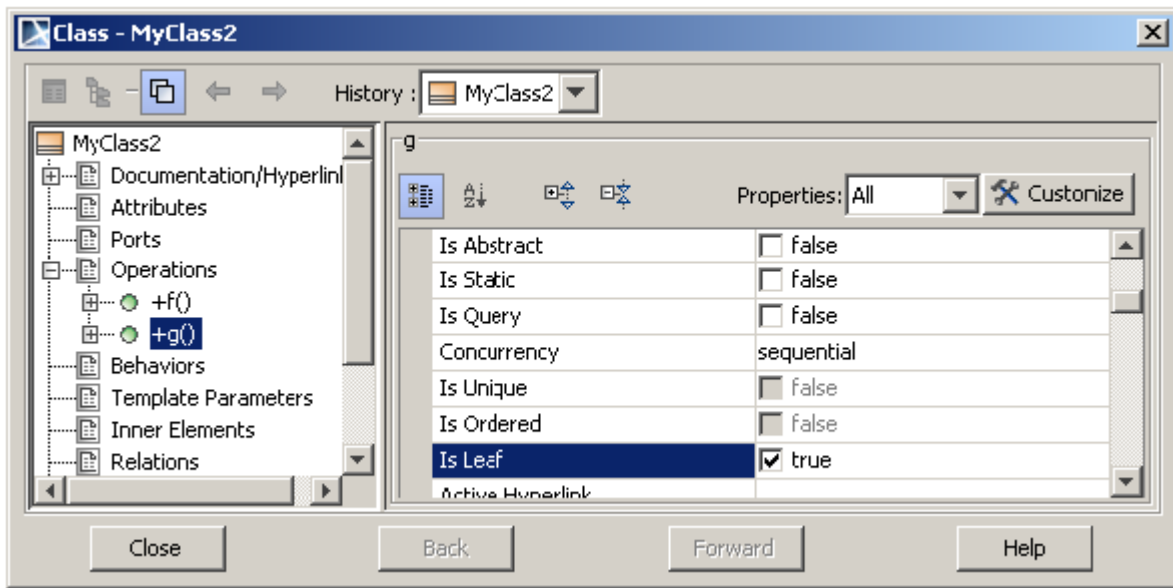
```
interface class MyInterface
{
    virtual void f();
    virtual void g();
};
ref class MyClass sealed: public
MyInterface
{
public:
    virtual void f(){};
    virtual void g(){};
};
```





```
interface class MyInterface2 {
public:
virtual void f();
virtual void g();
};
ref class MyClass2 : MyInterface2 {
public:
virtual void f() { }
virtual void g() sealed { }
// sub class cannot override g()
};
```





## Keywords for Generics

### Generic Functions

```
[attributes] [modifiers]  
return-type identifier <type-parameter identifier(s)>  
[type-parameter-constraints clauses]  
  
([formal-parameters])  
{  
    function-body  
}
```

#### Parameters

*attributes* (Optional)

Additional declarative information. For more information on attributes and attribute classes, see attributes.

*modifiers* (Optional)

A modifier for the function, such as **static**. **virtual** is not allowed since virtual methods may not be generic.

*return-type*

The type returned by the method. If the return type is void, no return value is required.

*identifier*

The function name.

*type-parameter identifier(s)*

Comma-separated identifiers list.

*formal-parameters* (Optional)

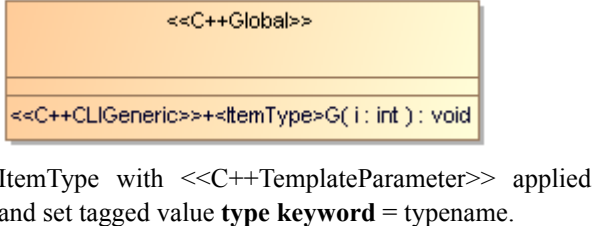
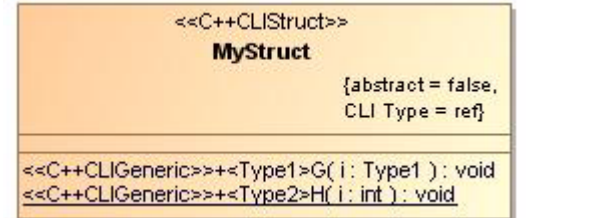
Parameter list.

*type-parameter-constraints-clauses*

This specifies restrictions on the types that may be used as type arguments, and takes the form specified in Constraints.

*function-body*

The body of the method, which may refer to the type parameter identifiers.

Code	MD-UML
<pre>generic &lt;typename ItemType&gt; void G(int i) {}</pre>	
<pre>ref struct MyStruct {     generic &lt;typename Type1&gt;     void G(Type1 i) {}      generic &lt;typename Type2&gt;     static void H(int i) {} };</pre>	

## Generic Classes

[attributes]

**generic** <class-key type-parameter-identifier(s)>

[constraint-clauses]

[accessibility-modifiers] **ref class** identifier [modifiers]

[: base-list]

```
{
    class-body
} [declarators] [;]
```

Parameters

attributes (optional)

Additional declarative information. For more information on attributes and attribute classes, see Attributes.

class-key

Either class or typename

type-parameter-identifier(s)

Comma-separated list of identifiers specifying the names of the type parameters.

constraint-clauses

A list (not comma-separated) of where clauses specifying the constraints for the type parameters. Takes the form:

where type-parameter-identifier : constraint-list ...

constraint-list

class-or-interface[, ...]

accessibility-modifiers

Allowed accessibility modifiers include public and private.

identifier

The name of the generic class, any valid C++ identifier.

modifiers (optional)

Allowed modifiers include sealed and abstract.

base-list

A list that contains the one base class and any implemented interfaces, all separated by commas.

class-body

The body of the class, containing fields, member functions, etc.

declarators

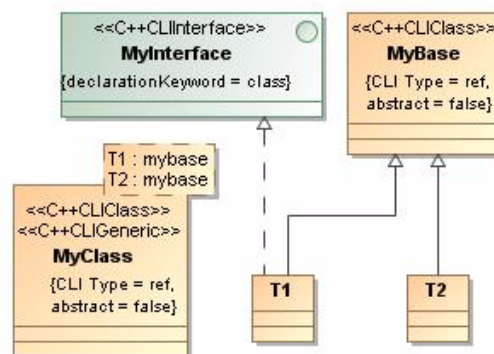
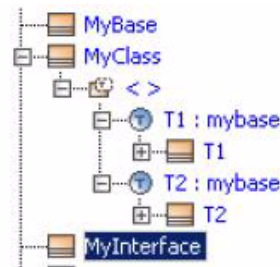
Declarations of any variables of this type. For example: ^identifier[, ...]

## Code

```
interface class MyInterface {};
ref class MyBase{};

generic <class T1, class T2>
where T1 : MyInterface, MyBase
where T2 : MyBase
ref class MyClass {};
```

## MD-UML



## Generic Interfaces

[attributes] **generic** <class-key type-parameter-identifier[, ...]>

[type-parameter-constraints-clauses]

[accessibility-modifiers] **interface class** identifier [: base-list] { interface-body } [declarators] ;

### Parameters

attributes (optional)

Additional declarative information. For more information on attributes and attribute classes, see Attributes.

class-key

class or typename

type-parameter-identifier(s)

Comma-separated identifiers list.

type-parameter-constraints-clauses

Takes the form specified in Constraints

accessibility-modifiers (optional)

Accessibility modifiers (e.g. public, private).

identifier

The interface name.

base-list (optional)

A list that contains one or more explicit base interfaces separated by commas.

interface-body

Declarations of the interface members.

declarators (optional)

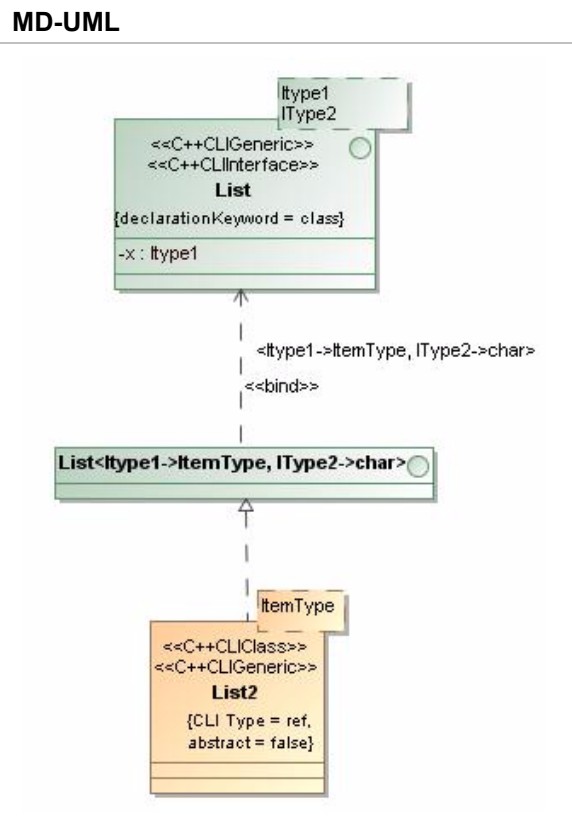
Declarations of variables based on this type.



```

Code
generic <typename Itype1,
typename Itype2>
public interface class List {
    Itype1 x;
};

generic <typename ItemType>
ref class List2 : public
List<ItemType, char>
{
};
    
```



## Generic Delegates

[attributes]  
**generic** < [class | **typename**] type-parameter-identifiers >  
 [type-parameter-constraints-clauses]  
 [accessibility-modifiers] **delegate** result-type identifier  
 ([formal-parameters]);

### Parameters

attributes (Optional)

Additional declarative information. For more information on attributes and attribute classes, see Attributes.

*type-parameter-identifier(s)*

Comma-separated list of identifiers for the type parameters.

*type-parameter-constraints-clauses*

Takes the form specified in Constraints

*accessibility-modifiers* (Optional)

Accessibility modifiers (e.g., **public**, **private**).

*result-type*

The return type of the delegate.

*identifier*

The name of the delegate.

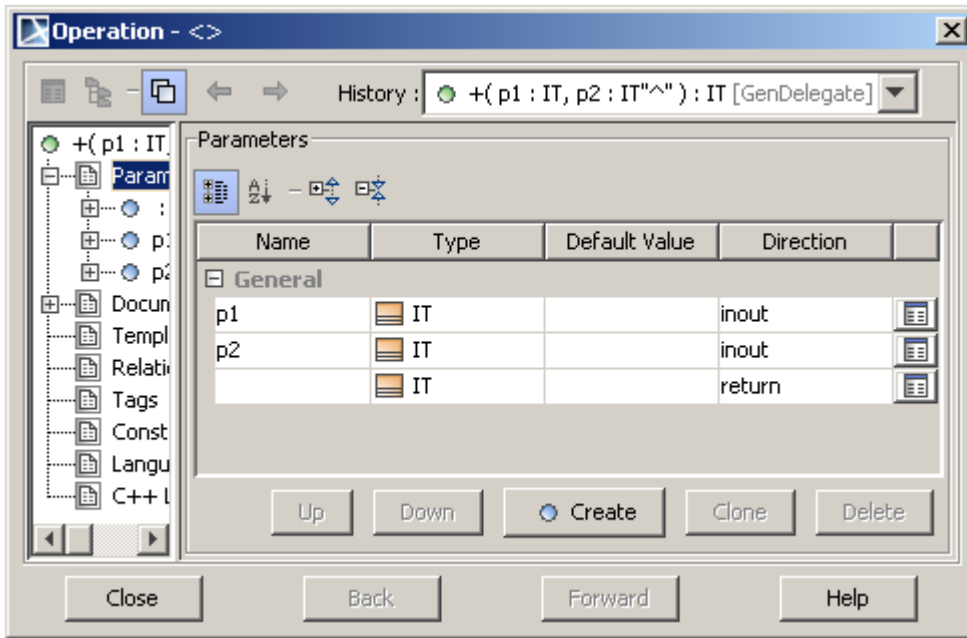
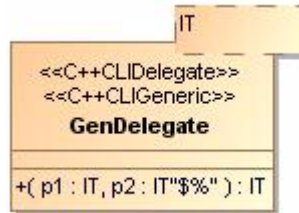
*formal-parameters* (Optional)

The parameter list of the delegate.

## Code

```
generic < class IT>  
delegate IT GenDelegate (IT p1, IT%  
p2) ;
```

## MD-UML



## Clr Data Member option

initonly

**initonly** indicates that variable assignment can only occur as part of the declaration or in a static constructor in the same class.

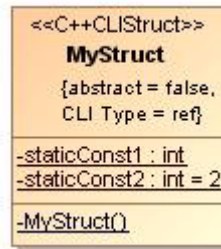
**Code**

```
ref struct MyStruct {
    initonly
    static int staticConst1;

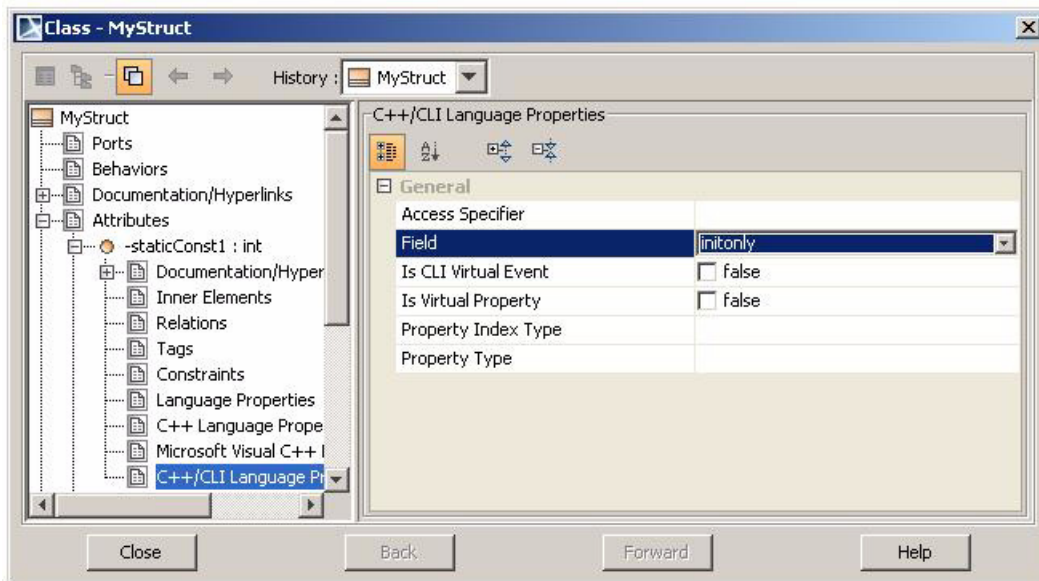
    initonly
    static int staticConst2 = 2;

    static MyStruct() {
        staticConst1 = 1;
    }
};
```

**MD-UML**



MyStruct with <<C++CLIStruct>> staticConst1 and staticConst2 have <<C++CLIAttribute>> applied with tagged value **field = initonly** and **Is Static = true**



literal

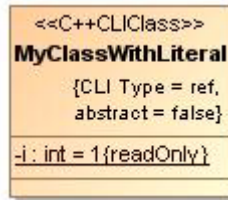
A variable (data member) marked as literal in a /clr compilation is the native equivalent of a **static const** variable.

A data member marked as **literal** must be initialized when declared and the value must be a constant integral, enum, or string type. Conversion from the type of the initialization expression to the type of the static const data-member must not require a user-defined conversion.

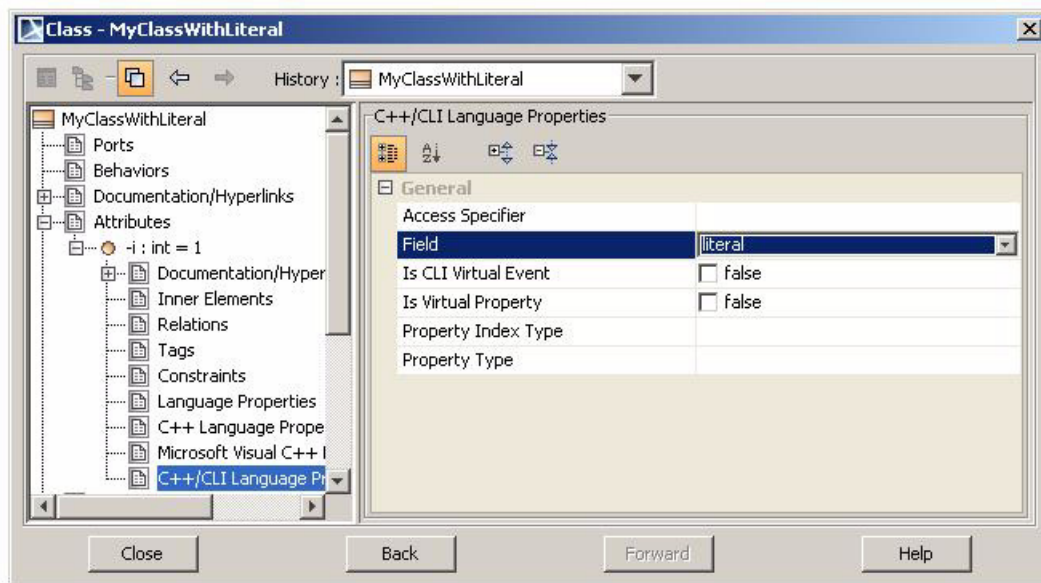
**Code**

**MD-UML**

```
ref class MyClassWithLiteral  
{  
    literal int i = 1;  
};
```



MyClassWithLiteral with <<C++CLIClass>>  
Attribute i has <<C++CLIAttribute>> applied  
with tagged value **field** = literal and set to **Is  
Static** = true and **Is Read Only** = true



## Inheritance Keywords

`__single_inheritance`, `__multiple_inheritance`, `__virtual_inheritance`

Grammar:

`class [__single_inheritance] class-name;`


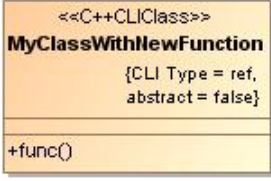
`class [__multiple_inheritance] class-name;`

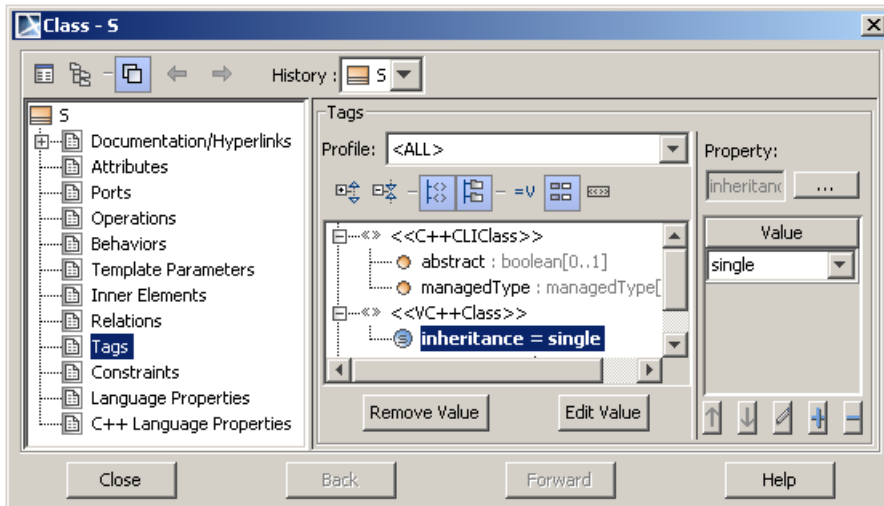
`class [__virtual_inheritance] class-name;`

### Parameter

*class-name*

The name of the class being declared.

Code	MD-UML
<pre>class __single_inheritance S;</pre>	 <p data-bbox="735 577 1066 607">Class with &lt;&lt;VC++Class&gt;&gt;</p> 



## Microsoft-Specific Native declaration keywords

### \_\_interface

Grammar:

modifier **\_\_interface** interface-name {interface-definition};

A Visual C++ interface can be defined as follows:

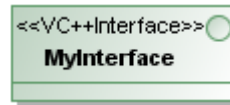
- Can inherit from zero or more base interfaces.
- Cannot inherit from a base class.
- Can only contain public, pure virtual methods.
- Cannot contain constructors, destructors, or operators.
- Cannot contain static methods.

- Cannot contain data members; properties are allowed.

**Code**

```
__interface MyInterface {};
```

**MD-UML**



**\_\_delegate**

Grammar:

**\_\_delegate** function-declarator

A delegate is roughly equivalent to a C++ function pointer except for the following difference:

- A delegate can only be bound to one or more methods within a `__gc` class.

When the compiler encounters the `__delegate` keyword, a definition of a `__gc` class is generated. This `__gc` class has the following characteristics:

- It inherits from **System::MulticastDelegate**.
- It has a constructor that takes two arguments: a pointer to a `__gc` class or **NULL** (in the case of binding to a static method) and a fully qualified method of the specified type.
- It has a method called **Invoke**, whose signature matches the declared signature of the delegate.

**Code**

```
__delegate int MyDelegate();
```

**MD-UML**



**\_\_event**

Grammar:

**\_\_event** method-declarator;

**\_\_event** **\_\_interface** interface-specifier;

**\_\_event** member-declarator;

## Native Events

Code	MD-UML
<pre>class Source { public:     __event void MyEvent(int i); };</pre>	

## Com Events

The `__interface` keyword is always required after `__event` for a COM event source.

Code	MD-UML
<pre>__interface IEvents { }; class CSource { public:     __event __interface IEvents; };</pre>	

**NOTE** This mapping produces the exceptional case for syntax checker. The syntax checker has to allow attribute without name for this case.

## Managed Events

Code	MD-UML
<pre>public __delegate void D(); public __gc class X { public:     __event D* E;     __event void noE(); };</pre>	

**NOTE** `__delegate` is in C++ Managed Profile whereas `__event` is in Microsoft Visual C++ Profile.

## Microsoft-Specific Modifiers

### Based addressing with `__based`

The `__based` keyword allows you to declare pointers based on pointers (pointers that are offsets from existing pointers).

type `__based( base )` declarator

Example

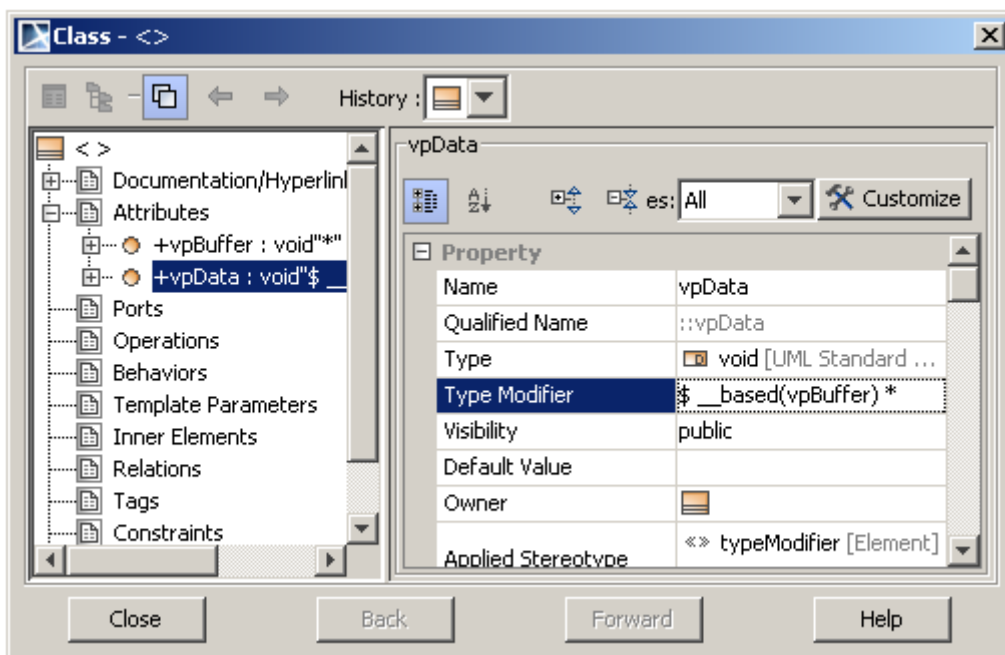
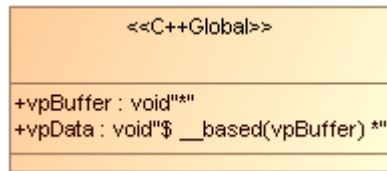
```
// based_pointers1.cpp
// compile with: /c
void *vpBuffer;
```

```
struct llist_t {
    void __based( vpBuffer ) *vpData;
    struct llist_t __based( vpBuffer ) *lNext;
};
```

**Code**

```
void *vpBuffer;
void __based(vpBuffer) *vpData;
```

**MD-UML**



**Function calling conventions**

The Visual C/C++ compiler provides several different conventions for calling internal and external functions.

**\_\_cdecl**

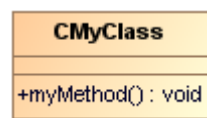
return-type **\_\_cdecl** function-name[(argument-list)]

This is the default calling convention for C and C++ programs. Because the stack is cleaned up by the caller, it can do **vararg** functions. The **\_\_cdecl** calling convention creates larger executables than **\_\_stdcall**, because it requires each function call to include stack cleanup code. The following list shows the implementation of this calling convention.

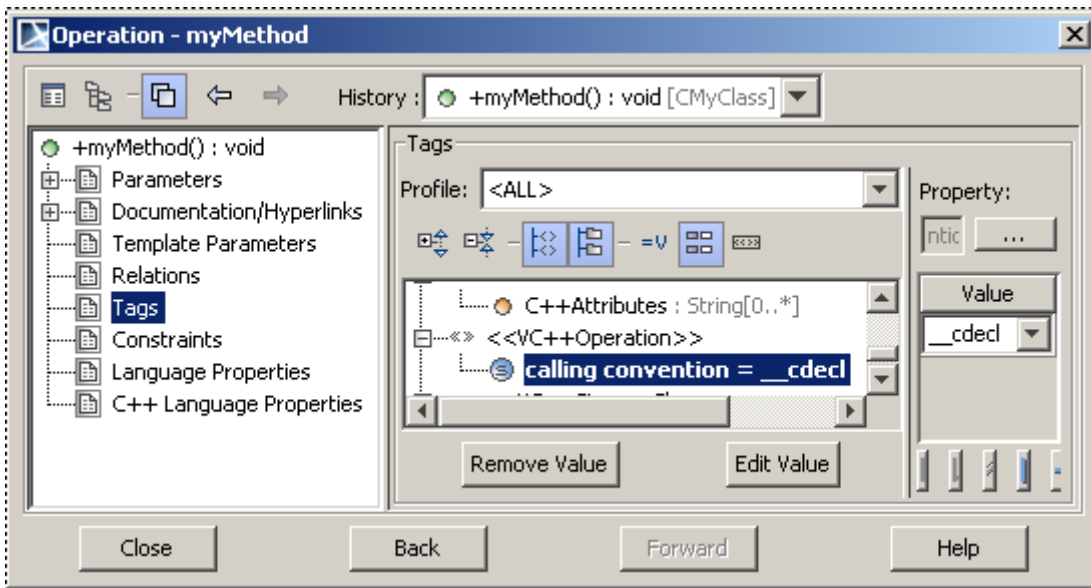
**Code**

```
class CMyClass {
    void __cdecl myMethod();
};
```

**MD-UML**







### **\_\_cdecl**

return-type **\_\_cdecl** function-name[(argument-list)]

Specifies that a function can only be called from managed code. Use **\_\_cdecl** for all virtual functions that will only be called from managed code. However, this calling convention cannot be used for functions that will be called from native code.

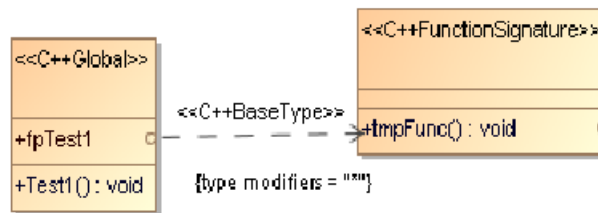
### **Example**

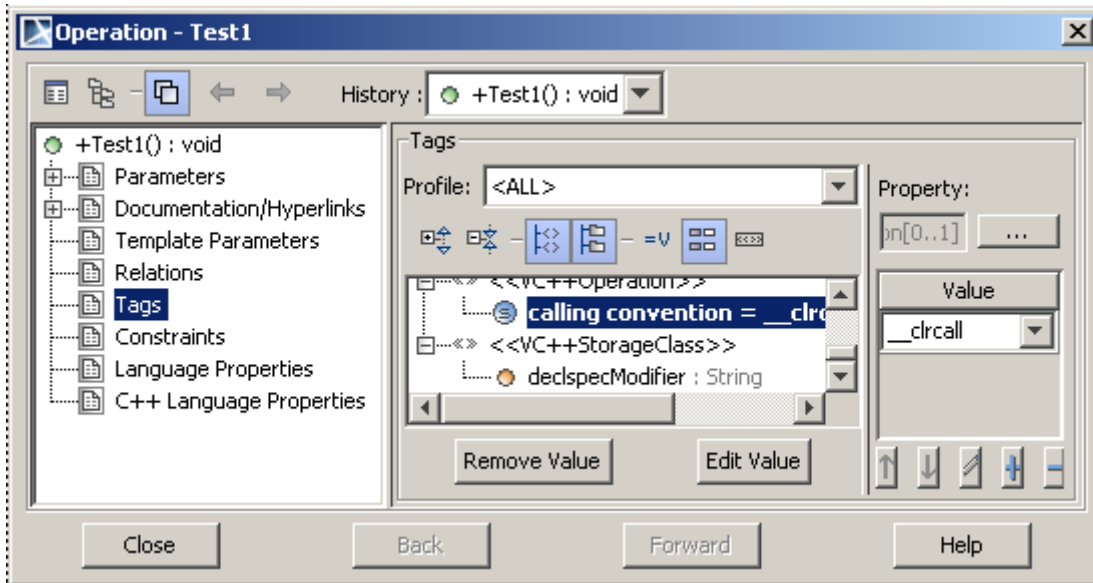
```
// compile with: /clr:oldSyntax /LD
void __cdecl Test1( ) {}
void (__cdecl *fpTest1)( ) = &Test1;
```

### **Code**

```
void __cdecl Test1( ) {}
void (__cdecl *fpTest1)(
) = &Test1;
```

### **MD-UML**





## \_\_stdcall

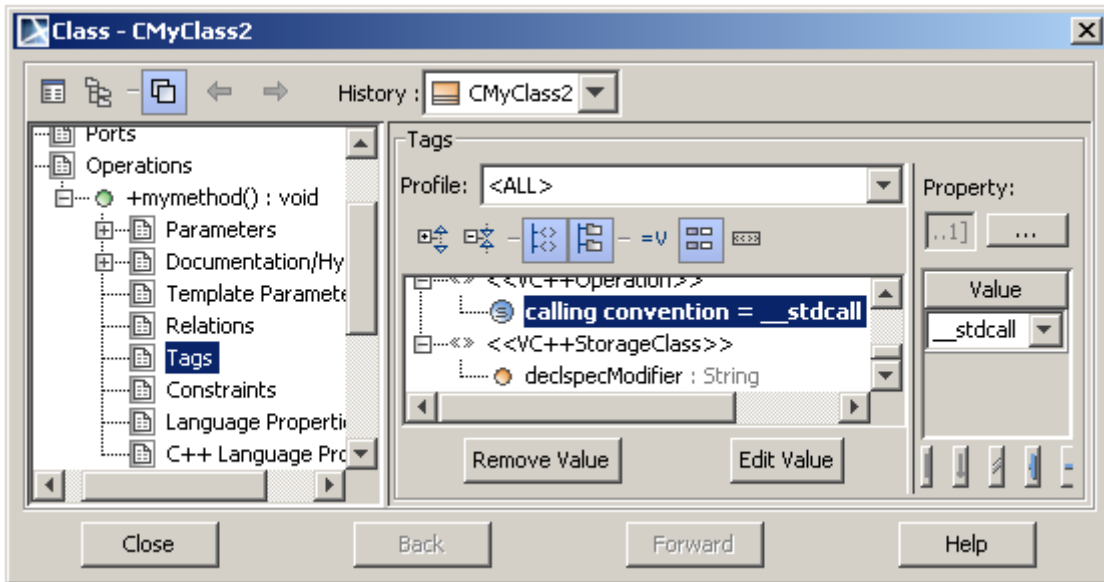
return-type **\_\_stdcall** function-name[(argument-list)]

The **\_\_stdcall** calling convention is used to call Win32 API functions. The callee cleans the stack, so the compiler makes **vararg** functions **\_\_cdecl**. Functions that use this calling convention require a function prototype.

**Code**

```
class CMyClass2 {
    void __stdcall mymethod();
};
```

**MD-UML**



**\_\_fastcall**

return-type **\_\_fastcall** function-name[(argument-list)]

The **\_\_fastcall** calling convention specifies that arguments to functions are to be passed in registers, when possible. The following list shows the implementation of this calling convention.

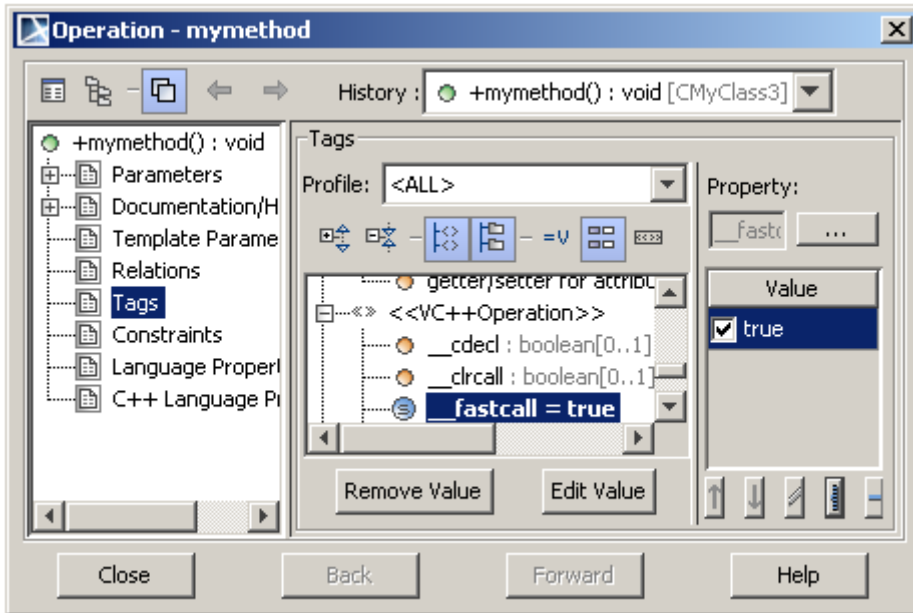
**Example**

```
class CMyClass3 {
    void __fastcall mymethod();
};
```

**Code**

```
class CMyClass3 {
    void __fastcall mymethod();
};
```

**MD-UML**



**\_\_thiscall**

return-type **\_\_thiscall** function-name[(argument-list)]

The **\_\_thiscall** calling convention is used on member functions and is the default calling convention used by C++ member functions that do not use variable arguments. Under **\_\_thiscall**, the callee cleans the stack, which is impossible for **vararg** functions. Arguments are pushed on the stack from right to left, with the **this** pointer being passed via register ECX, and not on the stack, on the x86 architecture.

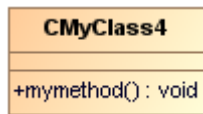
**Example**

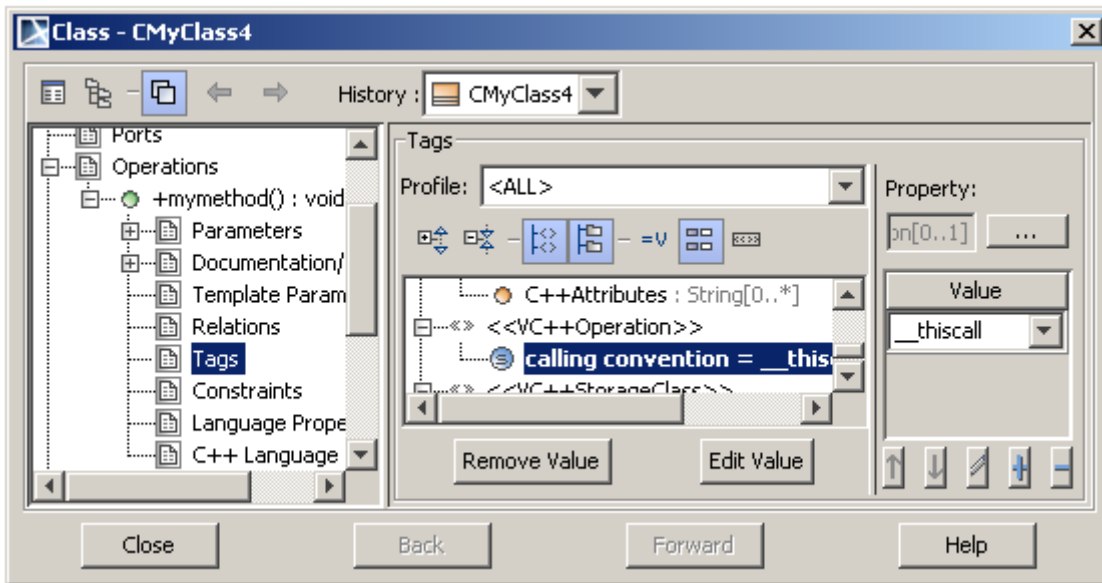
```
// compile with: /c /clr:oldSyntax
class CMyClass4 {
    void __thiscall mymethod();
    void __cdecl mymethod2();
};
```

**Code**

```
class CMyClass4 {
    void __thiscall mymethod();
};
```

**MD-UML**





## \_\_unaligned

type **\_\_unaligned** pointer\_identifier

When a pointer is declared as **\_\_unaligned**, the compiler assumes that the type or data pointed to is not aligned. **\_\_unaligned** is only valid in compilers for x64 and the Itanium Processor Family (IPF).

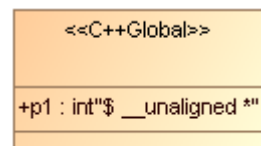
### Example

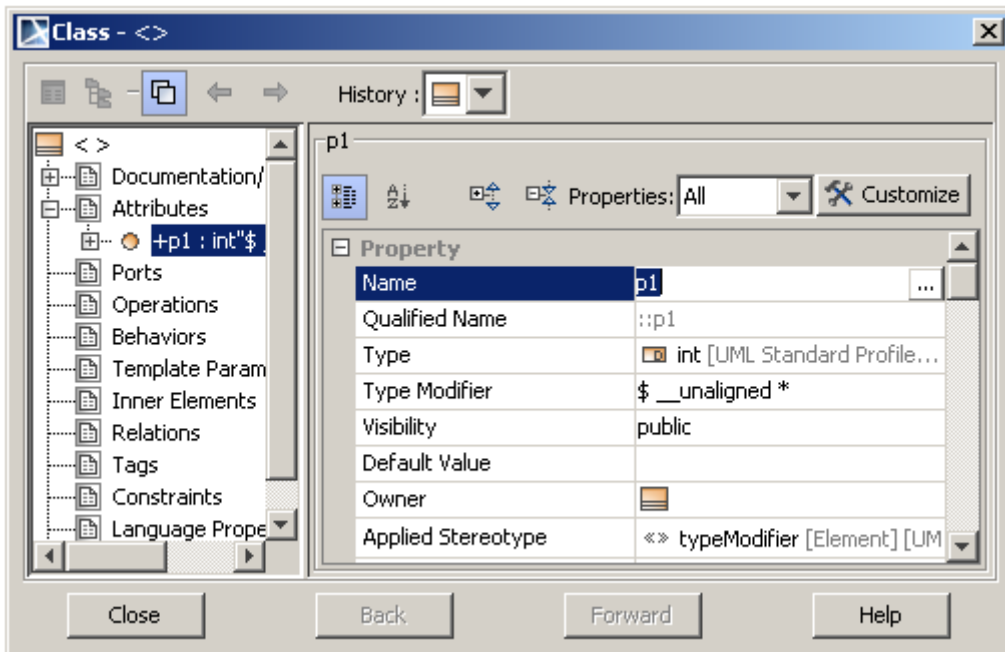
```
// compile with: /c
// processor: x64 IPF
#include <stdio.h>
int main() {
    char buf[100];
    int __unaligned *p1 = (int*)&buf[37];
    int *p2 = (int *)p1;
    *p1 = 0; // ok
    __try {
        *p2 = 0; // throws an exception
    }
    __except(1) {
        puts("exception");
    }
}
```

### Code

```
int __unaligned *p1;
```

### MD-UML





## \_\_w64

type \_\_w64 identifier

### Parameters

type

One of the three types that might cause problems in code being ported from a 32-bit to a 64-bit compiler: **int**, **long**, or a pointer.

identifier

The identifier for the variable you are creating.

\_\_w64 lets you mark variables, such that when you compile with /Wp64 the compiler will report any warnings that would be reported if you were compiling with a 64-bit compiler.

### Example

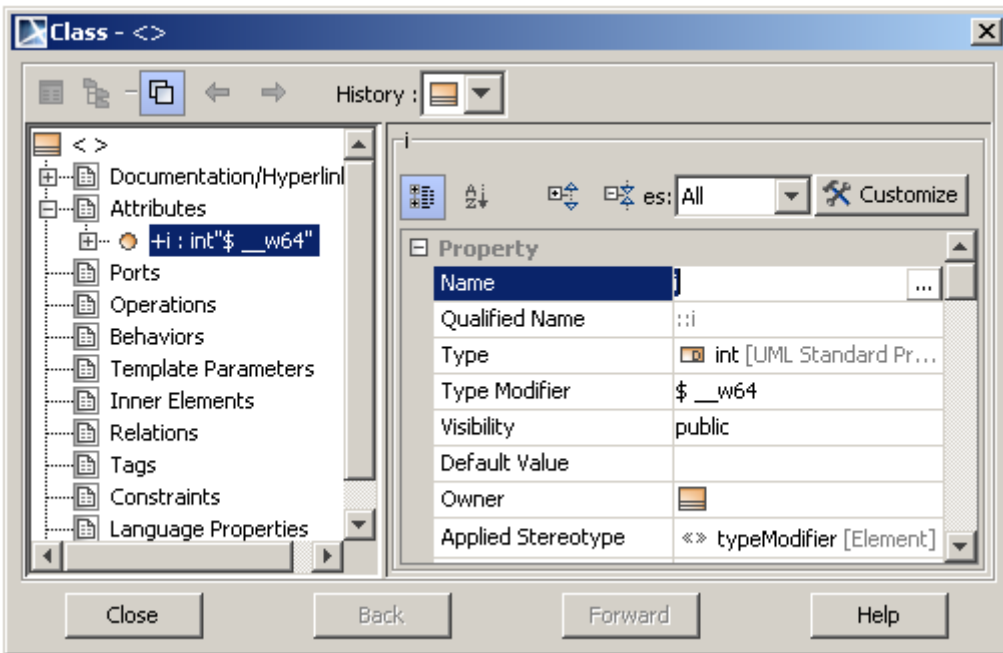
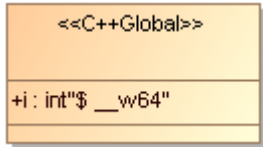
```
// compile with: /W3 /Wp64
typedef int Int_32;
#ifdef _WIN64
typedef __int64 Int_Native;
#else
typedef int __w64 Int_Native;
#endif
int main() {
    Int_32 i0 = 5;
    Int_Native i1 = 10;
    i0 = i1;    // C4244 64-bit int assigned to 32-bit int

    // char __w64 c; error, cannot use __w64 on char
}
```

**Code**

```
int __w64 i;
```

**MD-UML**



Extended storage-class attributes with `__declspec`

The extended attribute syntax for specifying storage-class information uses the `__declspec` keyword, which specifies that an instance of a given type is to be stored with a Microsoft-specific storage-class attribute listed below. Examples of other storage-class modifiers include the `static` and `extern` keywords. However, these keywords are part of the ANSI specification of the C and C++ languages, and as such are not covered by extended attribute syntax. The extended attribute syntax simplifies and standardizes Microsoft-specific extensions to the C and C++ languages.

decl-specifier:

`__declspec ( extended-decl-modifier-seq )`

extended-decl-modifier-seq:

extended-decl-modifieropt

extended-decl-modifier extended-decl-modifier-seq

extended-decl-modifier:

**align(#)**

**allocate("segname")**

**appdomain**

**deprecated**

**dllimport**

**dllexport**

**jitintrinsic**

**naked**

**noalias**  
**noinline**  
**noreturn**  
**nothrow**  
**novtable**  
**process**  
**property**({**get**=get\_func\_name|**put**=put\_func\_name})  
**restrict**  
**selectany**  
**thread**  
**uuid**("ComObjectGUID")

White space separates the declaration modifier sequence.

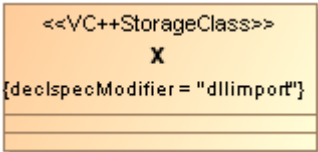
Extended attribute grammar supports these Microsoft-specific storage-class attributes: **align**, **allocate**, **appdomain**, **deprecated**, **dlexport**, **dllimport**, **jitintrinsic**, **naked**, **noalias**, **noinline**, **noreturn**, **nothrow**, **novtable**, **process**, **restrict**, **selectany**, and **thread**. It also supports these COM-object attributes: **property** and **uuid**.

The **dlexport**, **dllimport**, **naked**, **noalias**, **nothrow**, **property**, **restrict**, **selectany**, **thread**, and **uuid** storage-class attributes are properties only of the declaration of the object or function to which they are applied. The **thread** attribute affects data and objects only. The **naked** attribute affects functions only. The **dllimport** and **dlexport** attributes affect functions, data, and objects. The **property**, **selectany**, and **uuid** attributes affect COM objects.

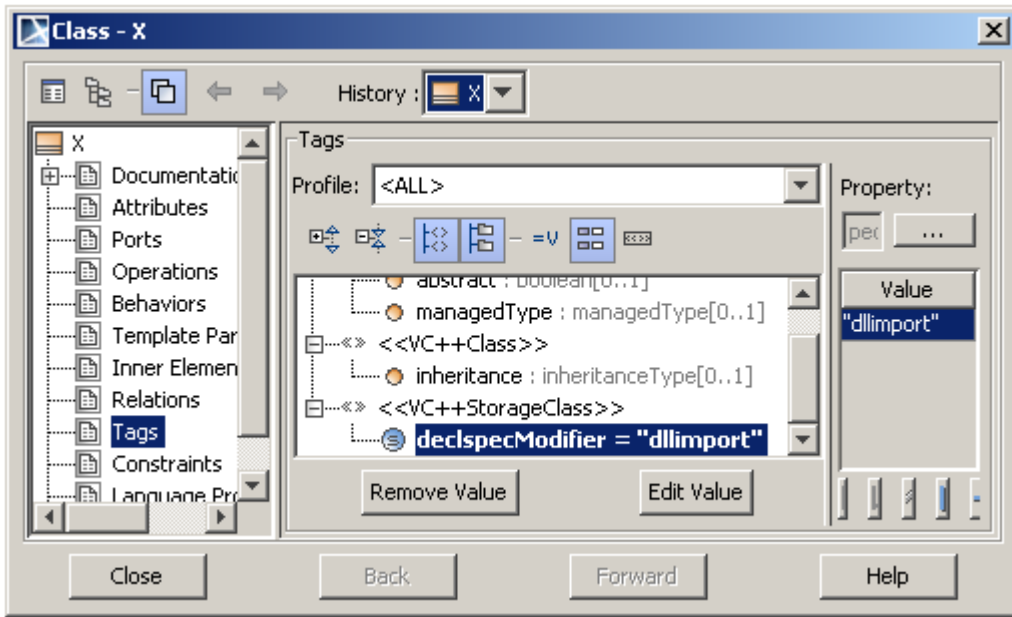
The **\_\_declspec** keywords should be placed at the beginning of a simple declaration. The compiler ignores, without warning, any **\_\_declspec** keywords placed after **\*** or **&** and in front of the variable identifier in a declaration.

**Note:**

- A **\_\_declspec** attribute specified in the beginning of a user-defined type declaration applies to the variable of that type.
- A **\_\_declspec** attribute placed after the **class** or **struct** keyword applies to the user-defined type.

Code	MD-UML
<pre>class __declspec(dllimport) X {};</pre>	 <pre> classDiagram     class X {         &lt;&lt;VC++StorageClass&gt;&gt;         {declspecModifier = "dllimport"}     }     </pre>

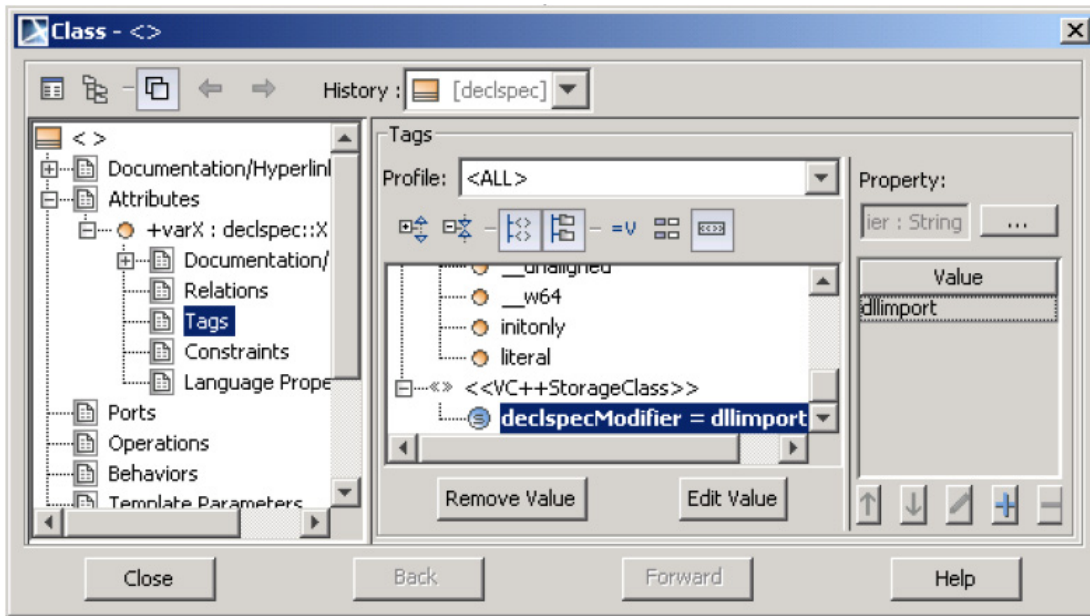
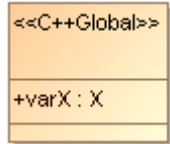




**Code**

```
__declspec(dllimport) class X {  
varX;
```

**MD-UML**



**\_\_restrict**

The **\_\_restrict** keyword is valid only on variables, and **\_\_declspec(restrict)** is only valid on function declarations and definitions.

When `__restrict` is used, the compiler will not propagate the no-alias property of a variable. That is, if you assign a `__restrict` variable to a non-`__restrict` variable, the compiler will not imply that the non-`__restrict` variable is not aliased.

Generally, if you affect the behavior of an entire function, it is better to use the `__declspec` than the keyword.

`__restrict` is similar to `restrict` from the C99 spec, but `__restrict` can be used in C++ or C programs.

### Example

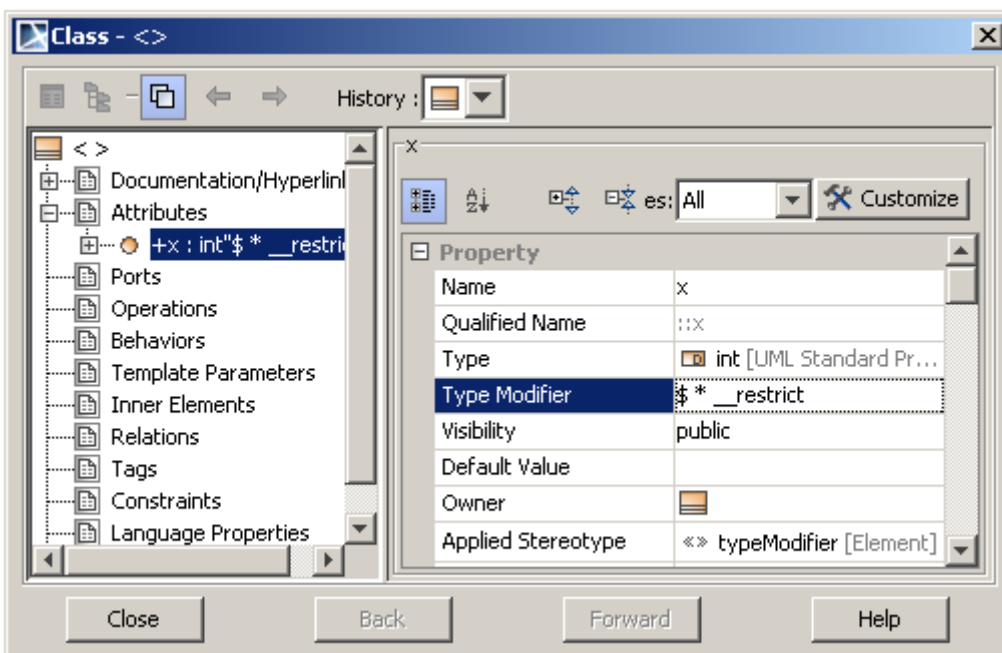
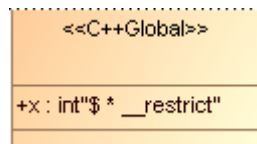
```
// __restrict_keyword.c
// compile with: /LD
// In the following function, declare a and b as disjoint arrays
// but do not have same assurance for c and d.
void sum2(int n, int * __restrict a, int * __restrict b,
          int * c, int * d) {
    int i;
    for (i = 0; i < n; i++) {
        a[i] = b[i] + c[i];
        c[i] = b[i] + d[i];
    }
}

// By marking union members as __restrict, tells the compiler that
// only z.x or z.y will be accessed in any given scope.
union z {
    int * __restrict x;
    double * __restrict y;
};
```

### Code

```
int * __restrict x;
```

### MD-UML



### \_\_forceinline, \_\_inline

The insertion (called inline expansion or inlining) occurs only if the compiler's cost/benefit analysis show it to be profitable. Inline expansion alleviates the function-call overhead at the potential cost of larger code size.

The **\_\_forceinline** keyword overrides the cost/benefit analysis and relies on the judgment of the programmer instead. Exercise caution when using **\_\_forceinline**. Indiscriminate use of **\_\_forceinline** can result in larger code with only marginal performance gains or, in some cases, even performance losses (due to increased paging of a larger executable, for example).

Using inline functions can make your program faster because they eliminate the overhead associated with function calls. Functions expanded inline are subject to code optimizations not available to normal functions.

The compiler treats the inline expansion options and keywords as suggestions. There is no guarantee that functions will be inlined. You cannot force the compiler to inline a particular function, even with the **\_\_forceinline** keyword. When compiling with **/clr**, the compiler will not inline a function if there are security attributes applied to the function.

The **inline** keyword is available only in C++. The **\_\_inline** and **\_\_forceinline** keywords are available in both C and C++. For compatibility with previous versions, **\_\_inline** is a synonym for **\_\_inline**.

#### Grammar:

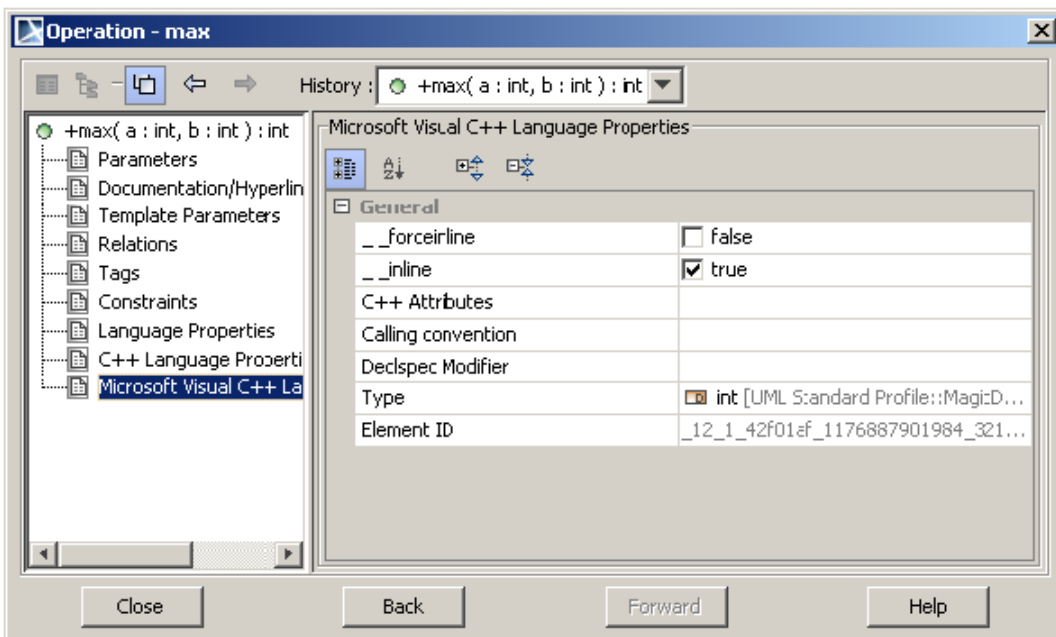
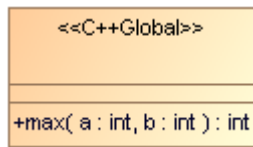
**\_\_inline** function\_declarator;

**\_\_forceinline** function\_declarator;

#### Code

```
__inline int max( int a , int b ) {
    if( a > b )
        return a;
    return b;
}
```

#### MD-UML



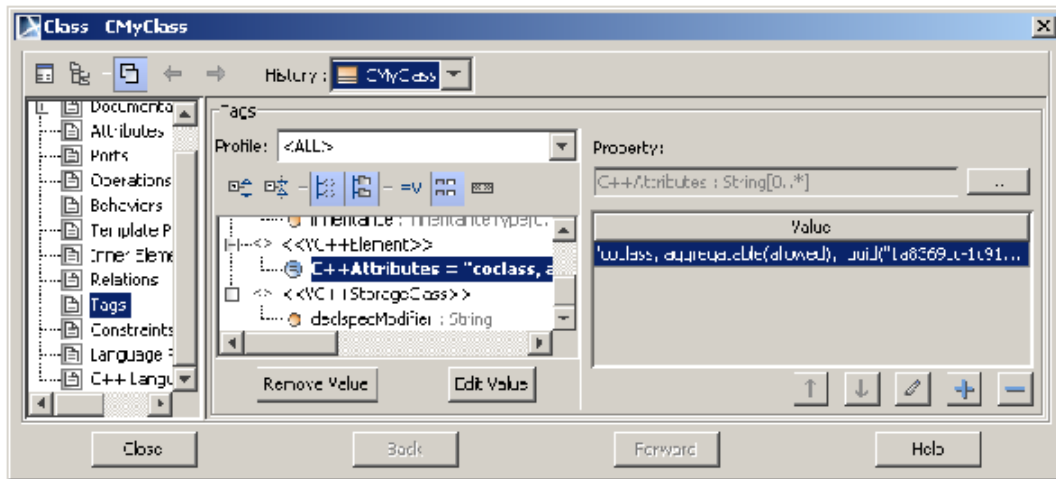
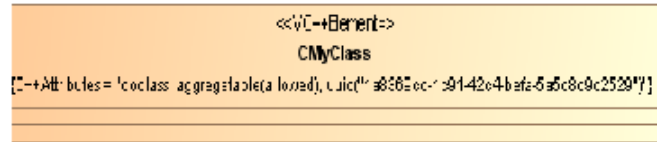
## C++ Attributes

Attributes are designed to simplify COM programming and .NET Framework common language runtime development. When you include attributes in your source files, the compiler works with provider DLLs to insert code or modify the code in the generated object files.

### Code

```
[ coclass,
  aggregatable(allowed),
  uuid("1a8369cc-1c91-42c4-befa-5a5d8c9d2529")]
class CMyClass {};
```

### MD-UML



# C# CODE ENGINEERING

## C# 2.0 Description

The MagicDraw UML C# Code Engineering Project is responsible for providing round-trip functionality between The MagicDraw UML and C# codes. In the current version of this project, it supports up to C# version 3.0.

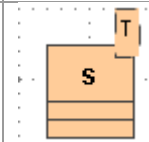
## Generics

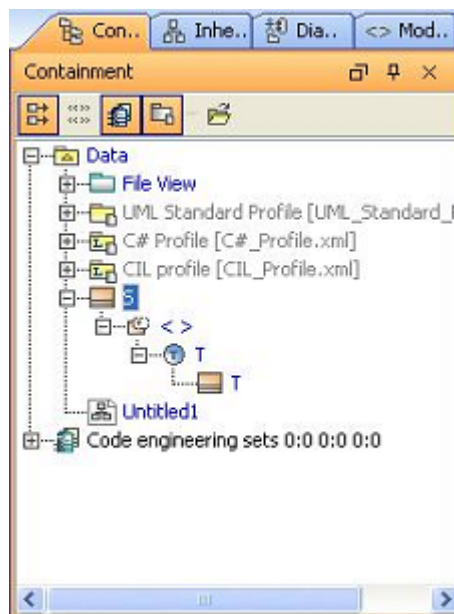
Generics permit classes, structs, interfaces, delegates, and methods to be parameterized by the types of data they store and manipulate. Generic class declaration should be mapped to the UML classifier (class or interface) with a template parameter. (See the detail of Generics in C# Specification chapter 20)

Additionally, Generics still affect other parts of the program structure such as attribute, operation, parameter, parent class, and overloading operators. The mappings of these are shown in the next part of this document.

## Generic Class

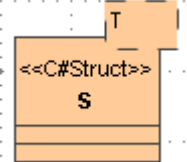
Class S has one template parameter named T. The default type of T is Class.

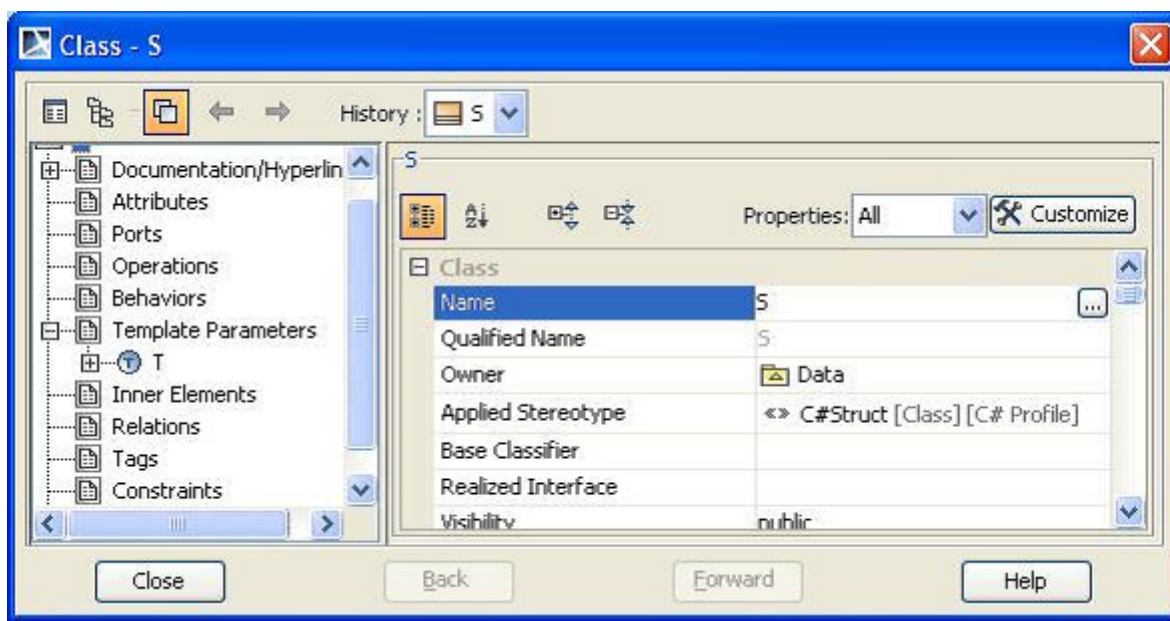
Code	MD-UML
<pre>public class S&lt;T&gt; { } </pre>	



### Generic Struct

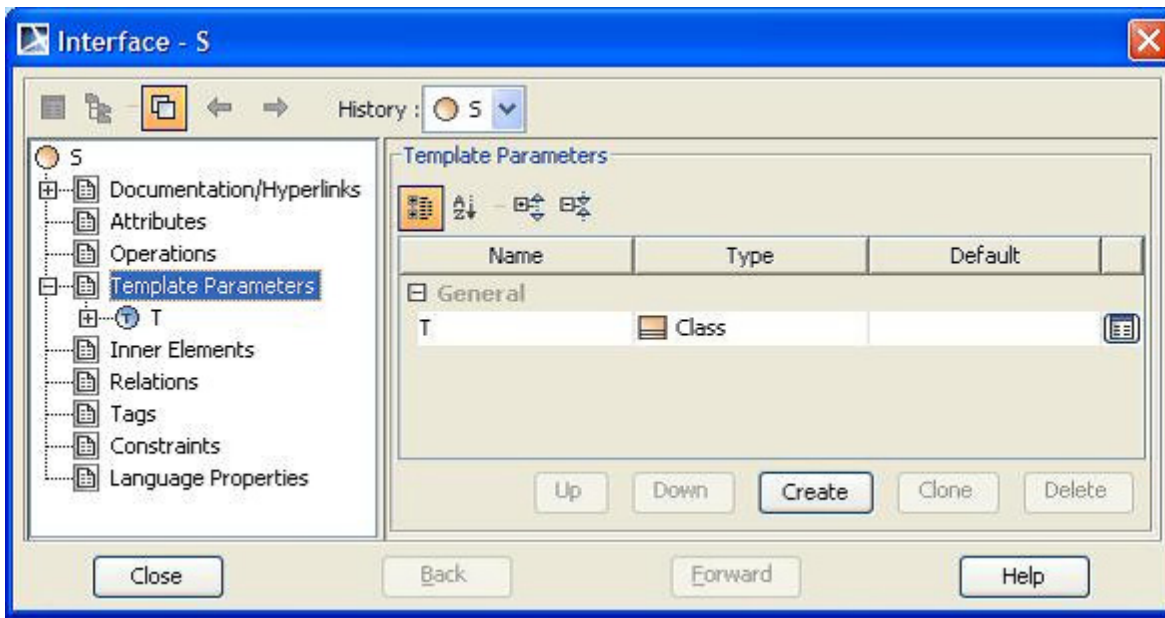
The type parameter of generic struct is created the same as generic class, but we apply <<C#Struct>> stereotype to the model.

Code	MD-UML
<pre>struct S&lt;T&gt; { } </pre>	



### Generic Interface

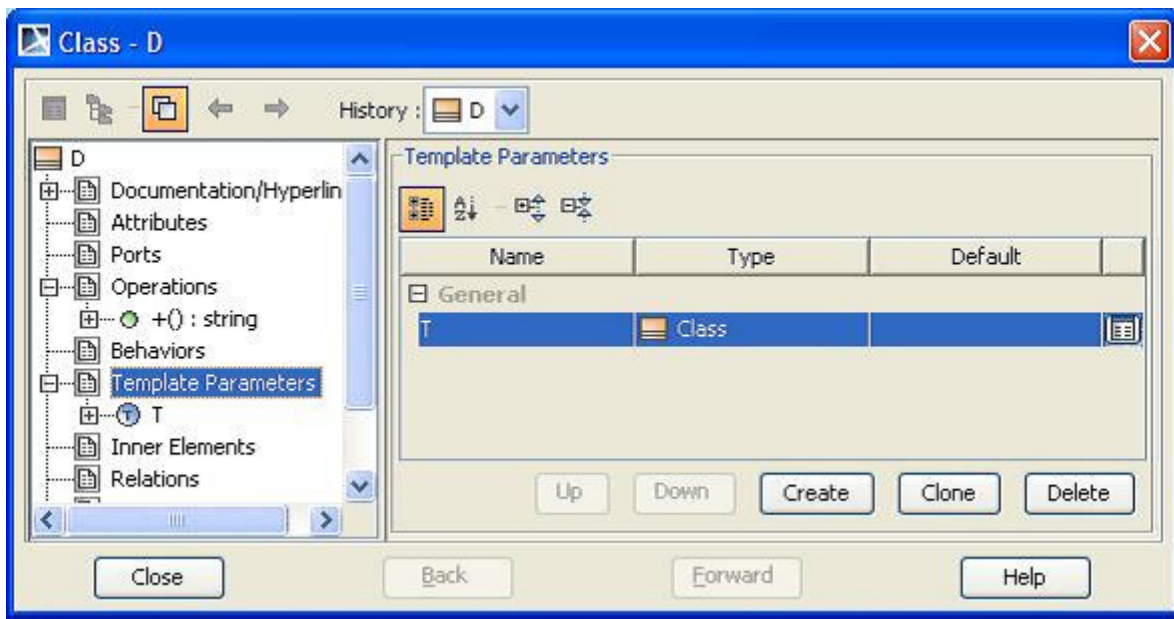
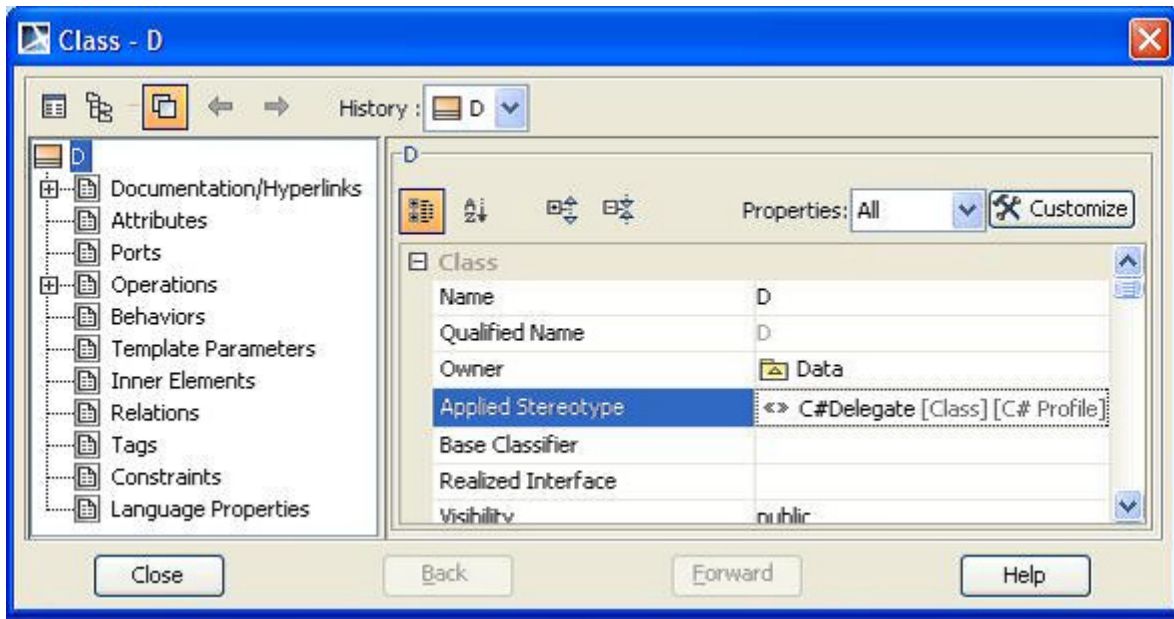
Code	MD-UML
<pre>interface S&lt;T&gt; { }</pre>	



### Generic Delegate

To create a generic delegate, we create a class model and apply the <<C#Delegate>> to the model. We, then, create an empty named method with delegate return type, and add template parameter like a normal generic class.

Code	MD-UML
<pre>delegate string D&lt;T&gt;();</pre>	

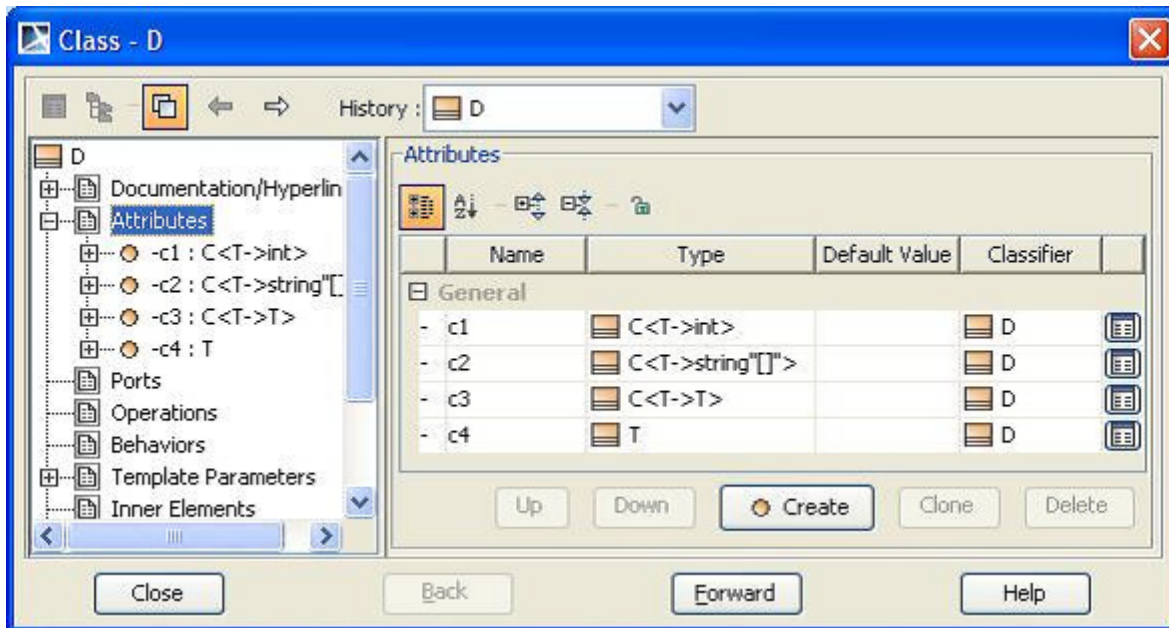
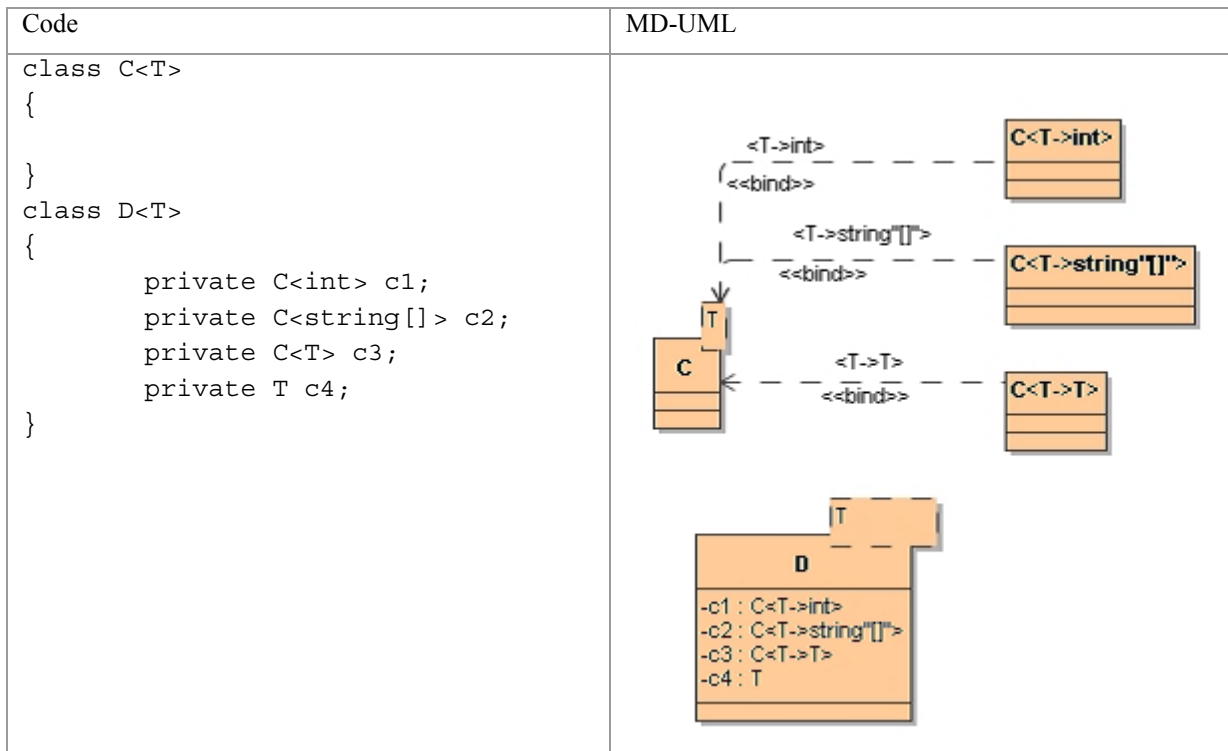


## Generic Attribute

The type of attributes in the class can be generic binding (template binding) or template parameter of owner class.

The example code shows attributes that use template binding as its type



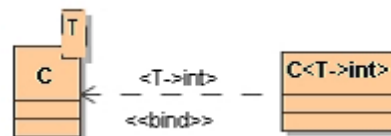


The following shows how to create each attribute.

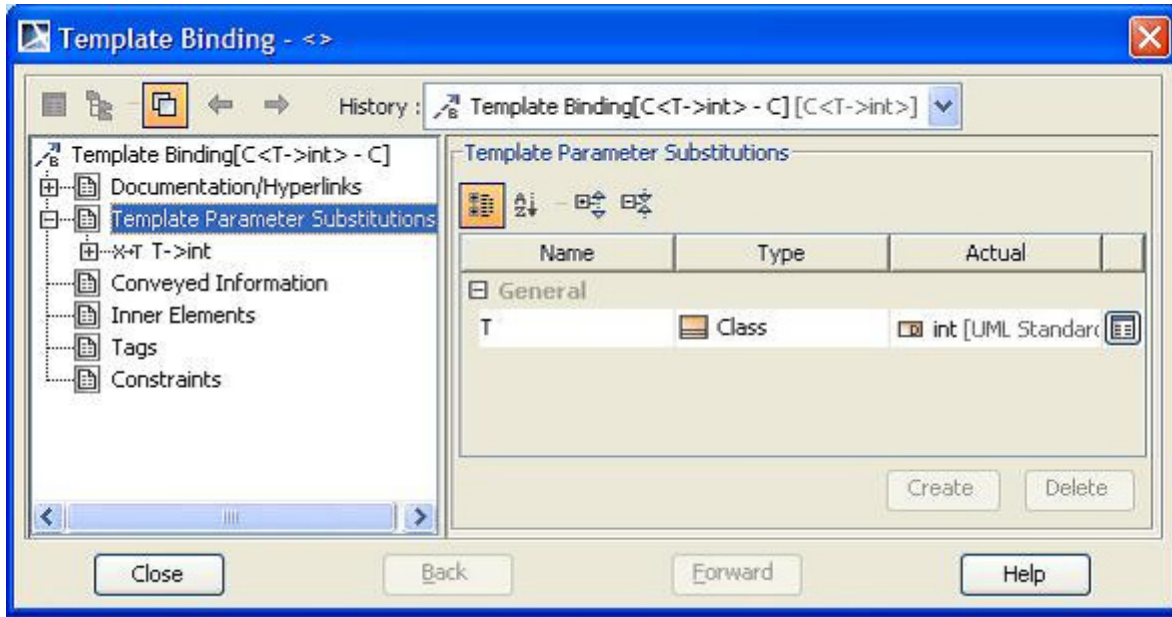
```

..
Private C<int> c1;
        
```

template binding for C<int>

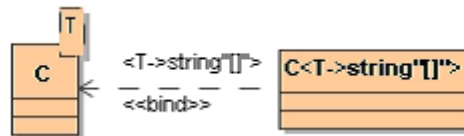


Open the specification of template binding link and add the binding type in Template Parameter Substitutions section. For binding with int datatype, select data type int in the Actual.

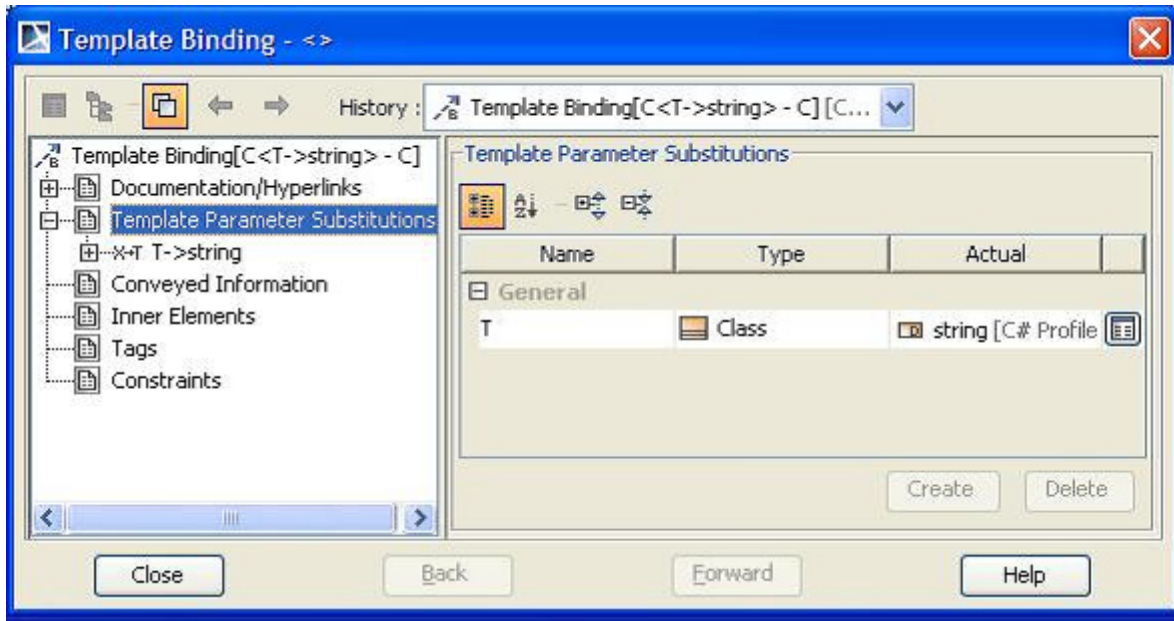


template binding for C<string[]>

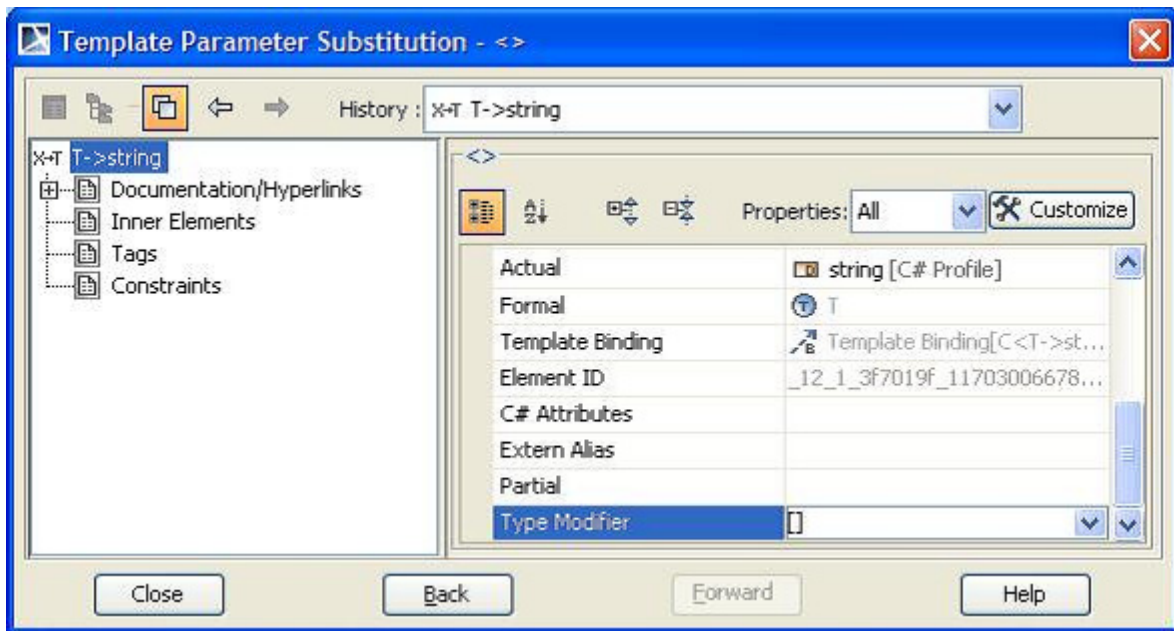
```
..
private C<string[]> c2;
```



Create binding type as usual.

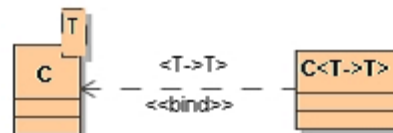


To add type modifier [] to string, open the specification of template parameter substitution, and then add [] to Type Modifier property.

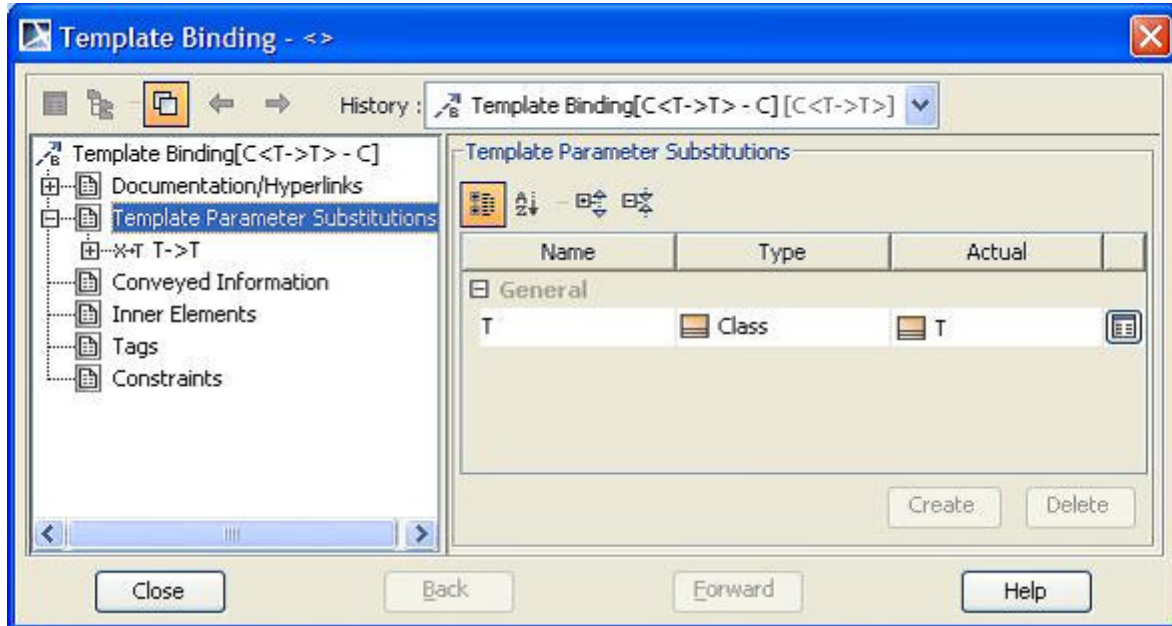


```
..
private C<T> c3;
```

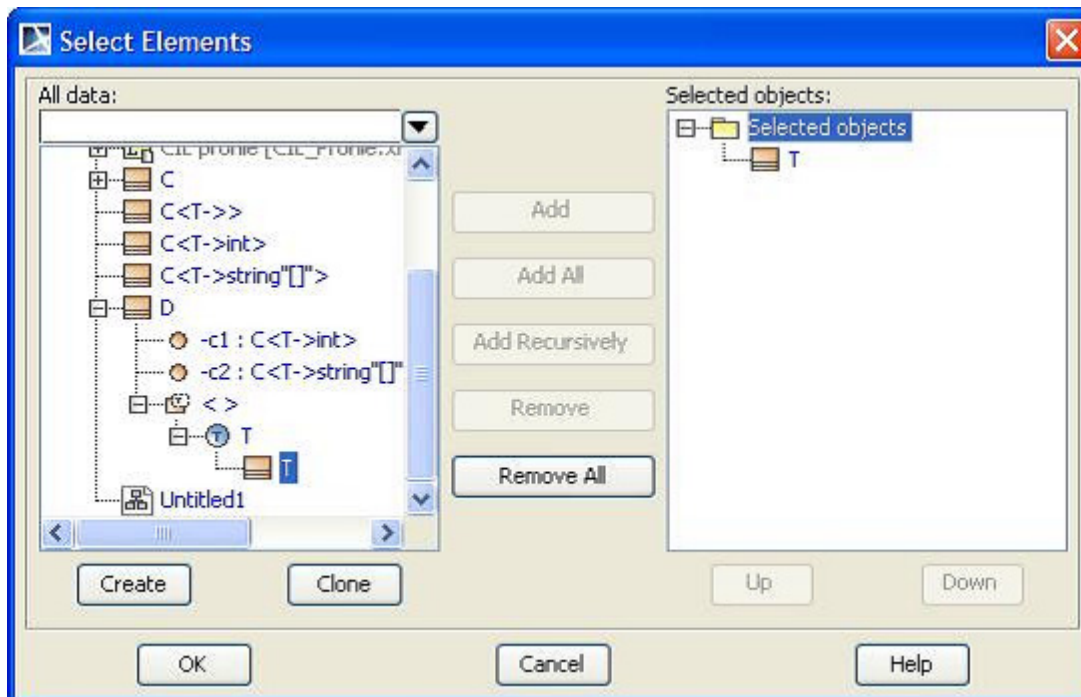
template binding for C<T>



To create template binding for C<T>, we have to select the correct T element for the Actual. In this case, the binding type T is the template parameter of its owner class that is D<T>.



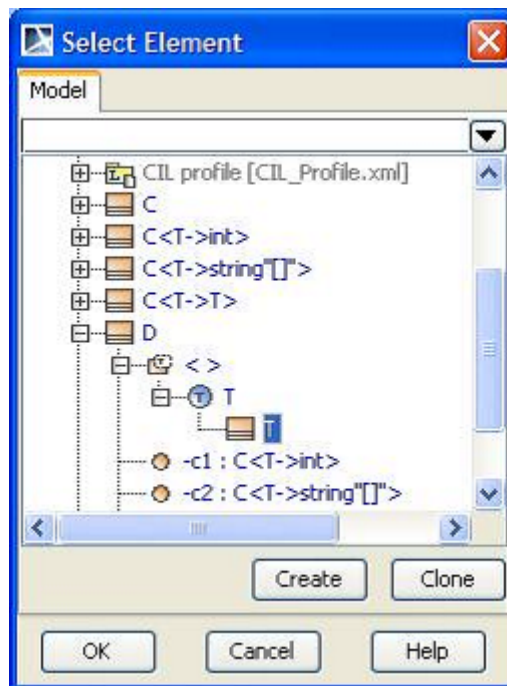
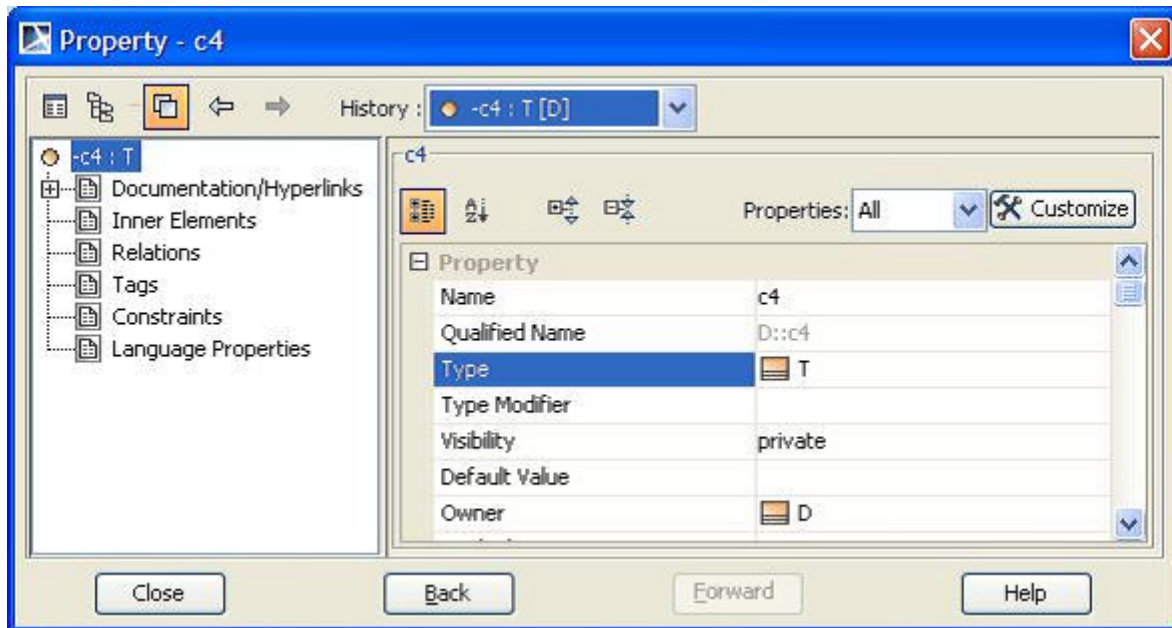
The following panel shows the correct type for creating template binding type T.



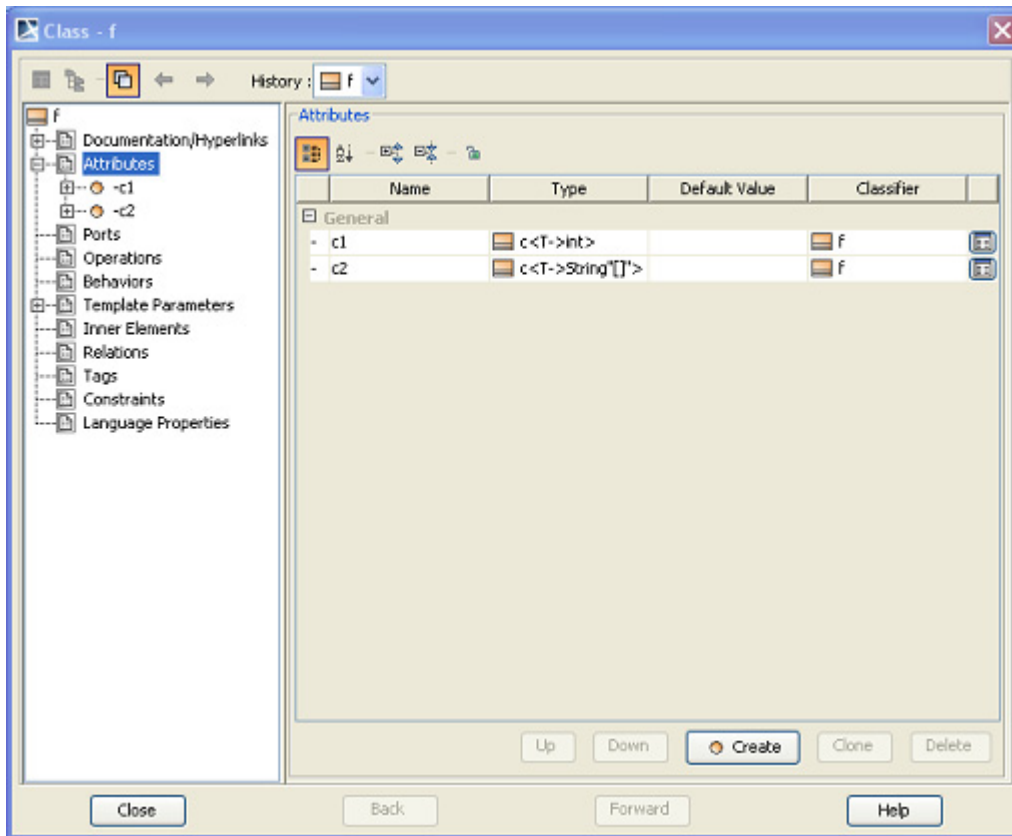
```
..
private T c4;
```

The type of attribute, c4 is not a template binding class. Its type is a template parameter of the owner class D<T>

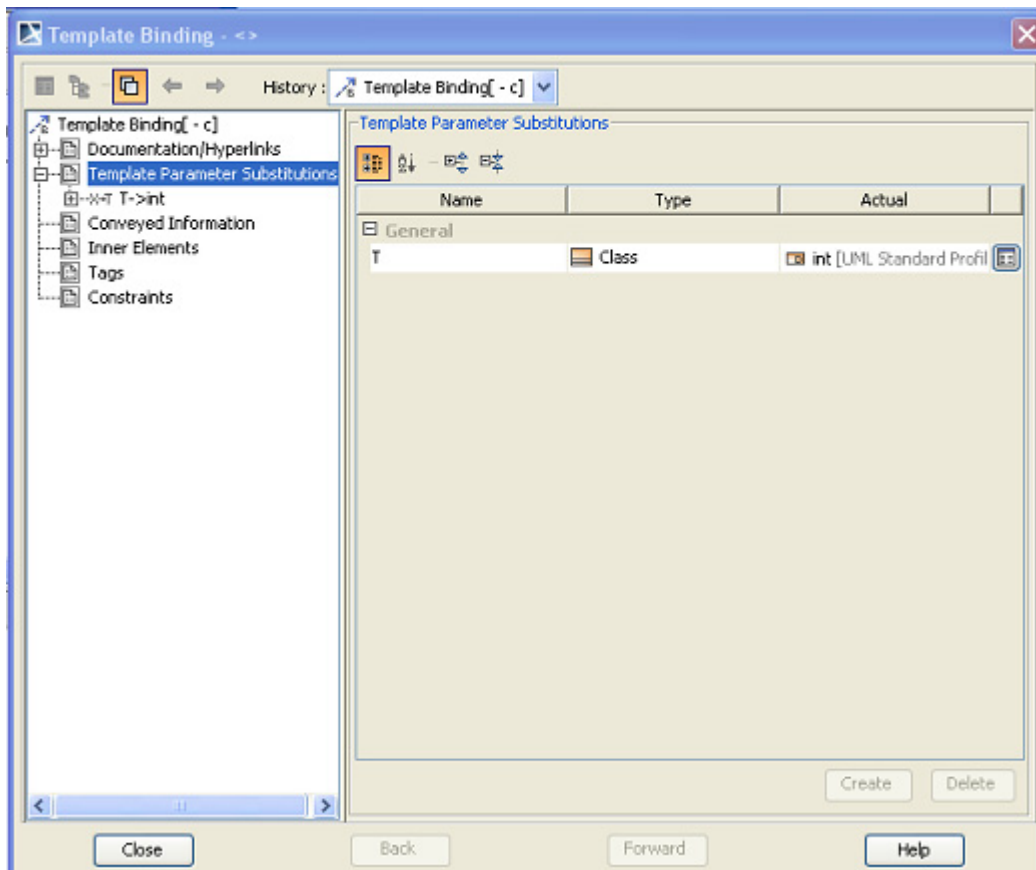
We create the attribute, c4 as usual, but select the correct T type. In this case, it is template parameter of the owner class, D<T>.







The property of template binding is shown below

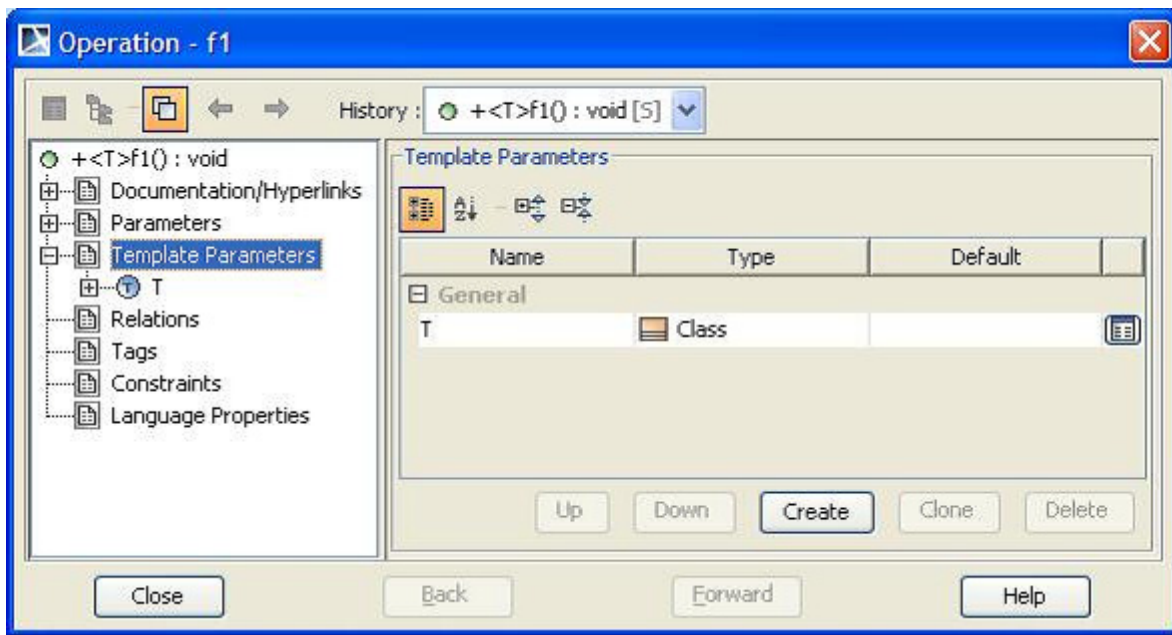


### Generic Operation

Generic can be applied to operation such as operation name, return type and operation parameter.

Code	MD-UML
<pre>public class S&lt;T&gt; {     public void f1&lt;T&gt;() {} }</pre>	

To create generic operation, open the specification of operation, and create template parameter as usual.



For the return type and parameters of the operation, we create them like a generic attribute creation.

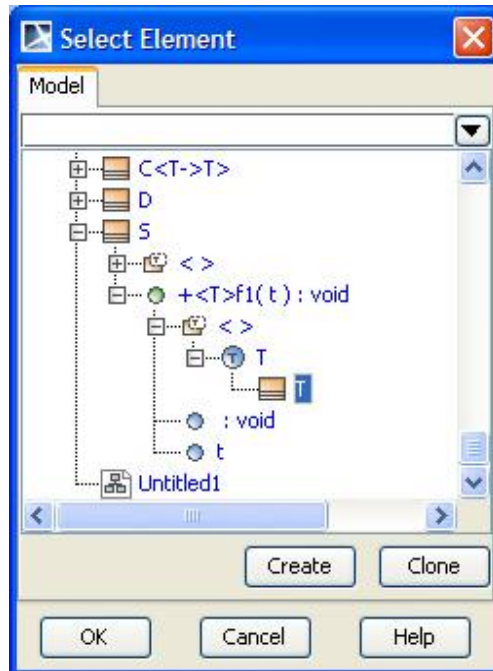
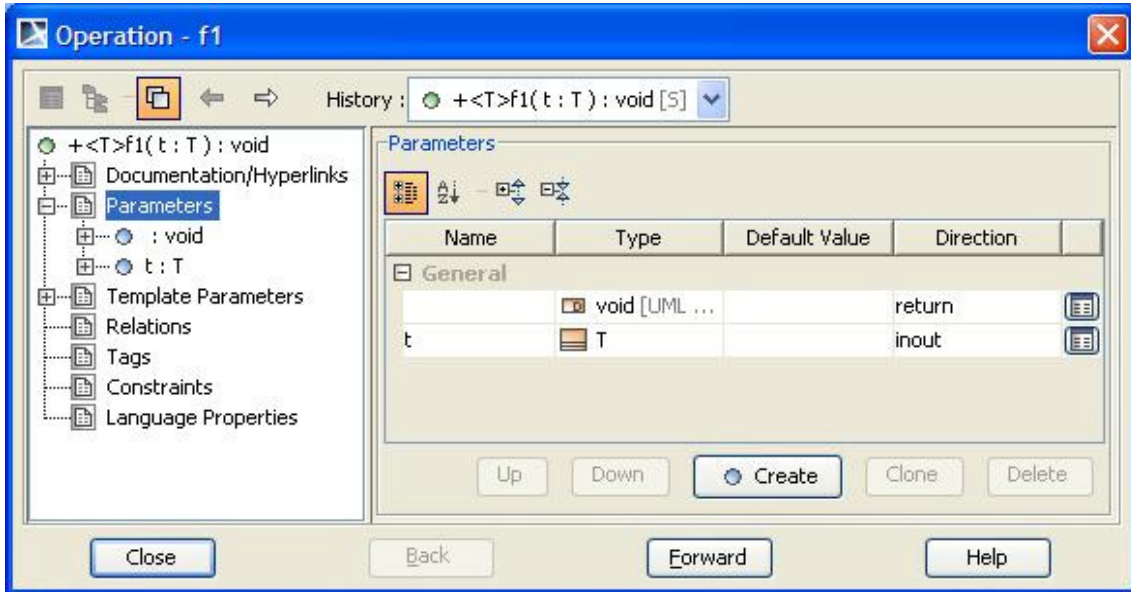
Important: We have to create or select the correct binding types.

In the example code, the type of method parameter *t* in the first method, `public void f1<T> (T t)`, must be the template parameter of the owner method, `f1<T>`, not the template parameter of the owner class, `S<T>`.

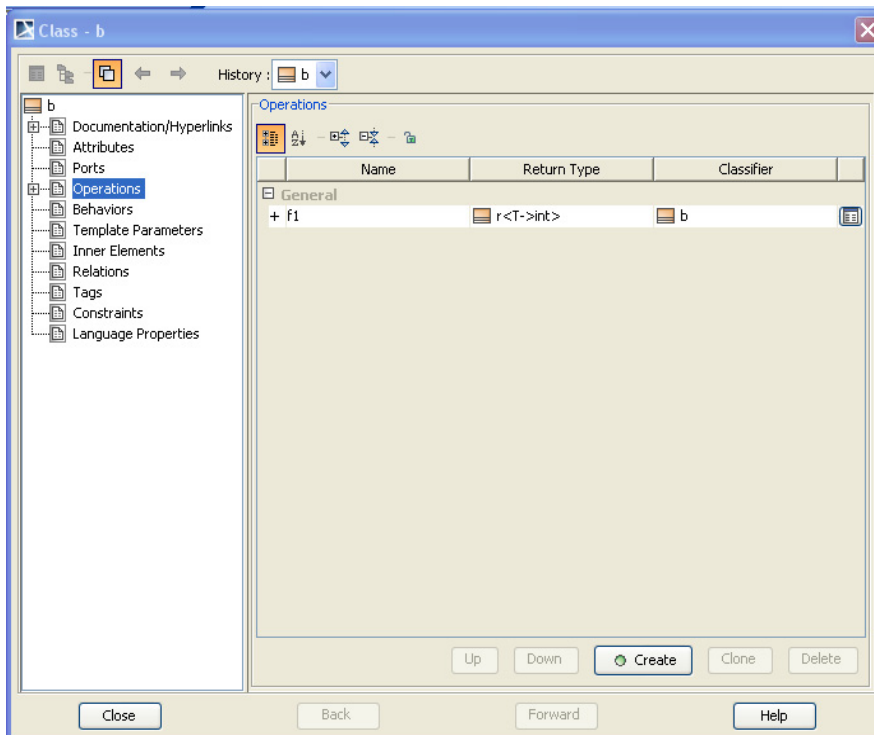
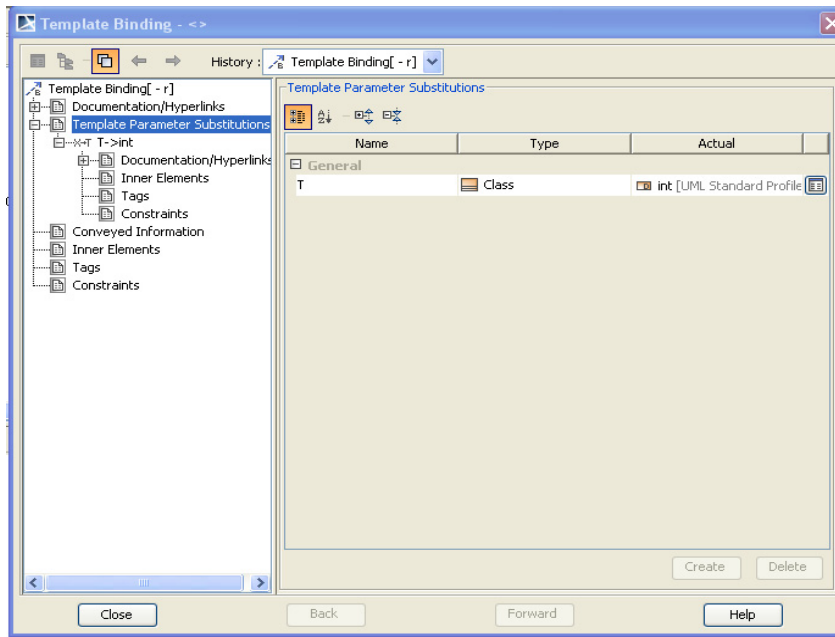
For the second method, `public void f1<U, V> (T t, U u)`, the type of parameter *t* must be the template parameter of the owner class, `S<T>`

Code	MD-UML
<pre>public class S&lt;T&gt; {     public void f1&lt;T&gt; (T t)     {     }     public void f1&lt;U, V&gt; (T t, U u)     {     } }</pre>	

The following panels show the selection of method parameter t of the first method.



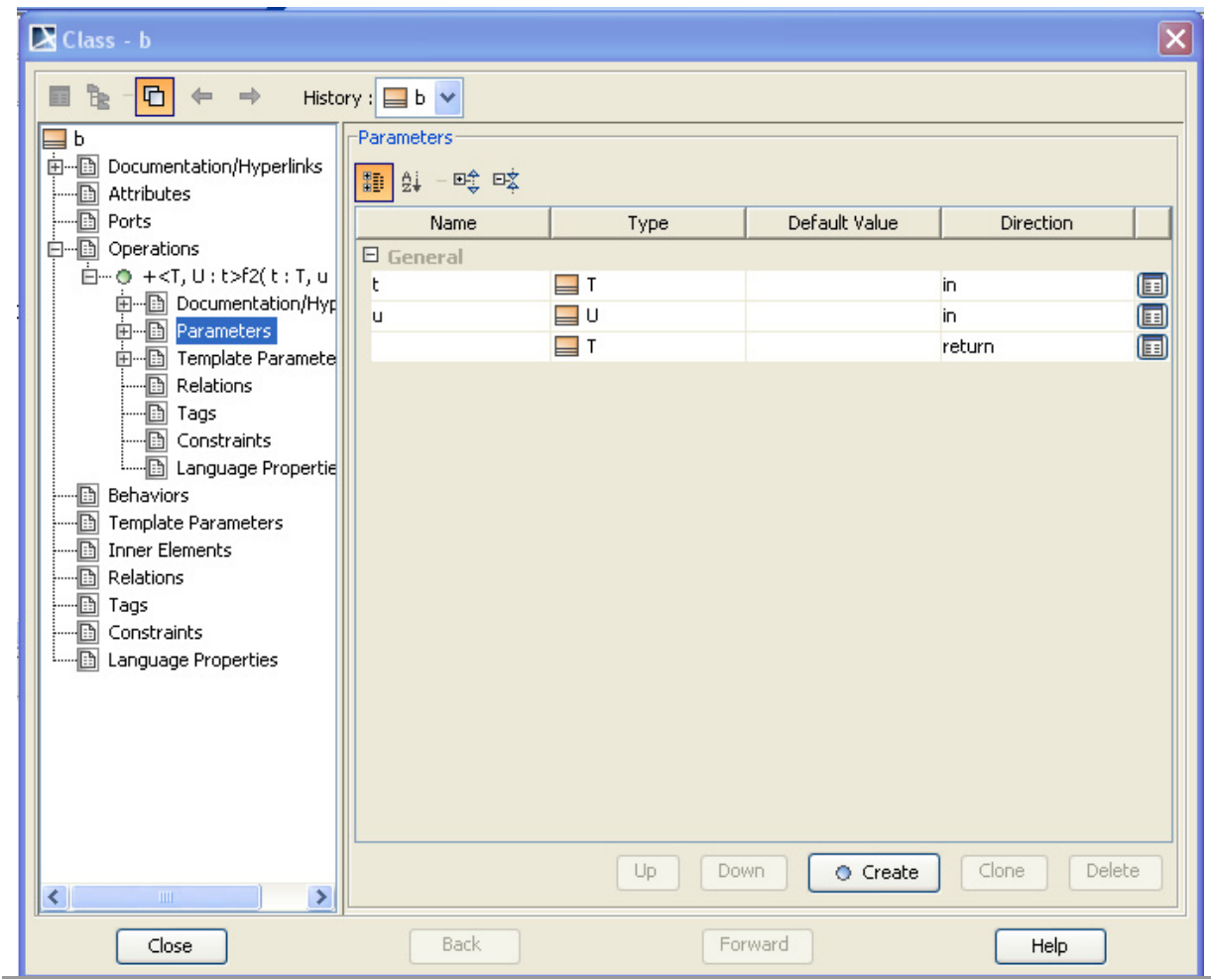
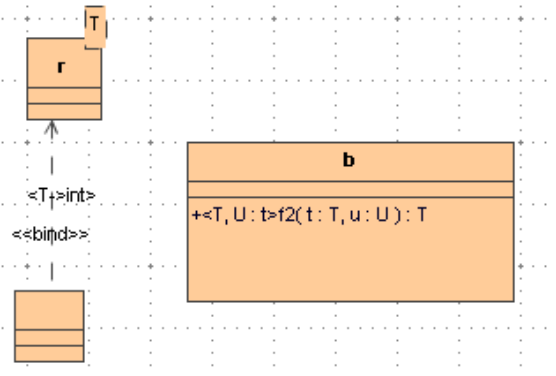




# C# CODE ENGINEERING

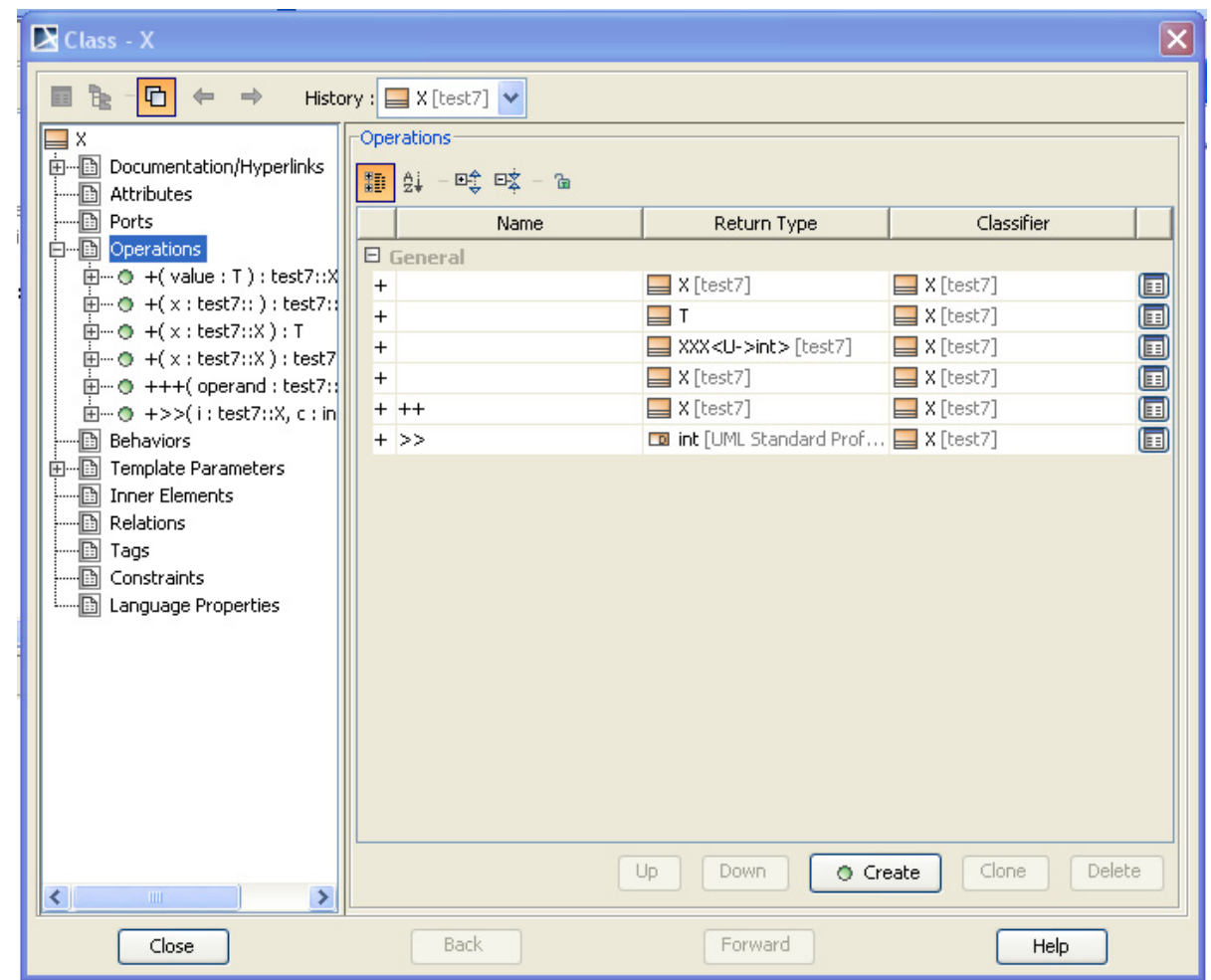
## C# 2.0 Description

```
class b
{
    public T f2<T, U>(T t, U u)
    where U : T { return t; }
}
```

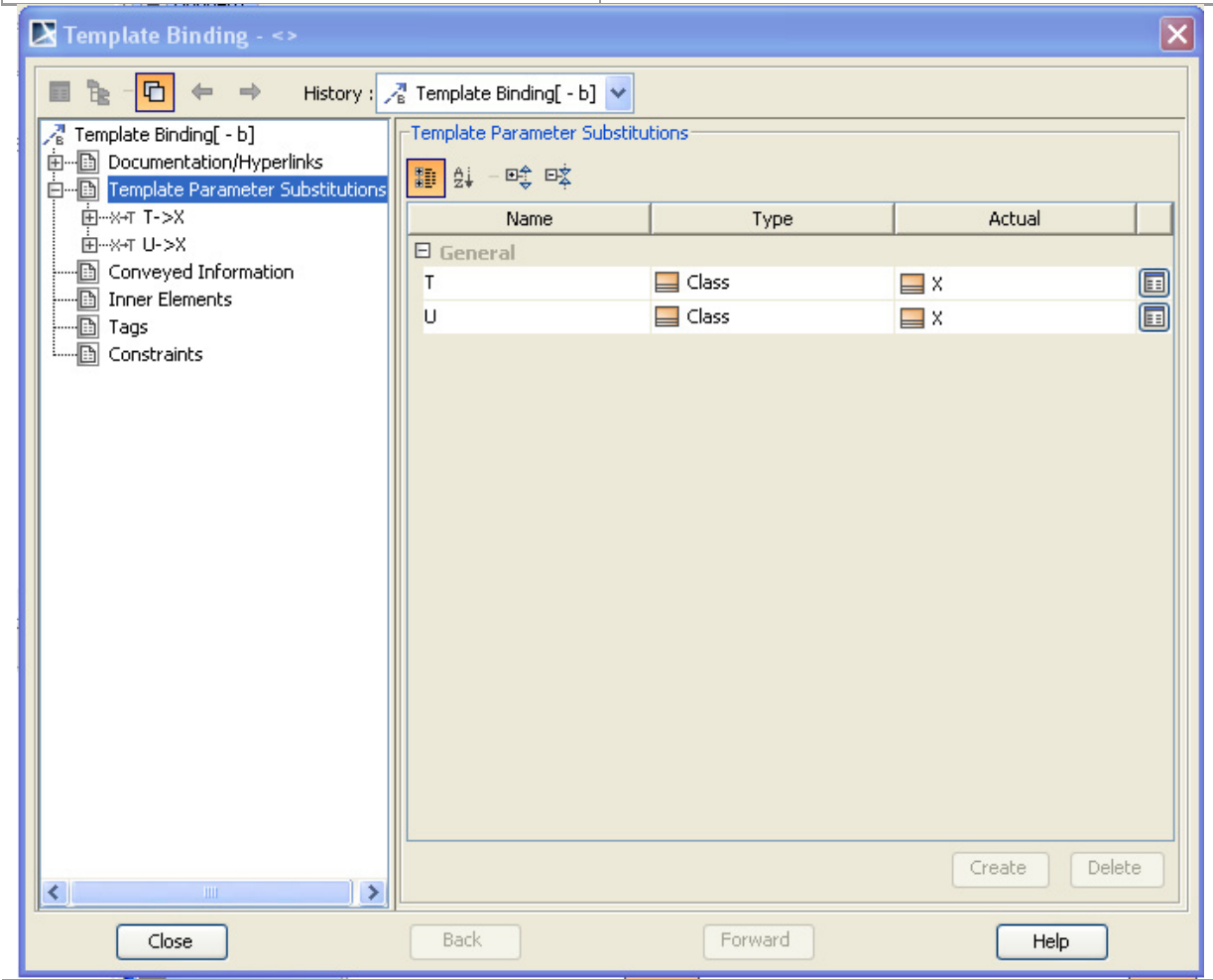
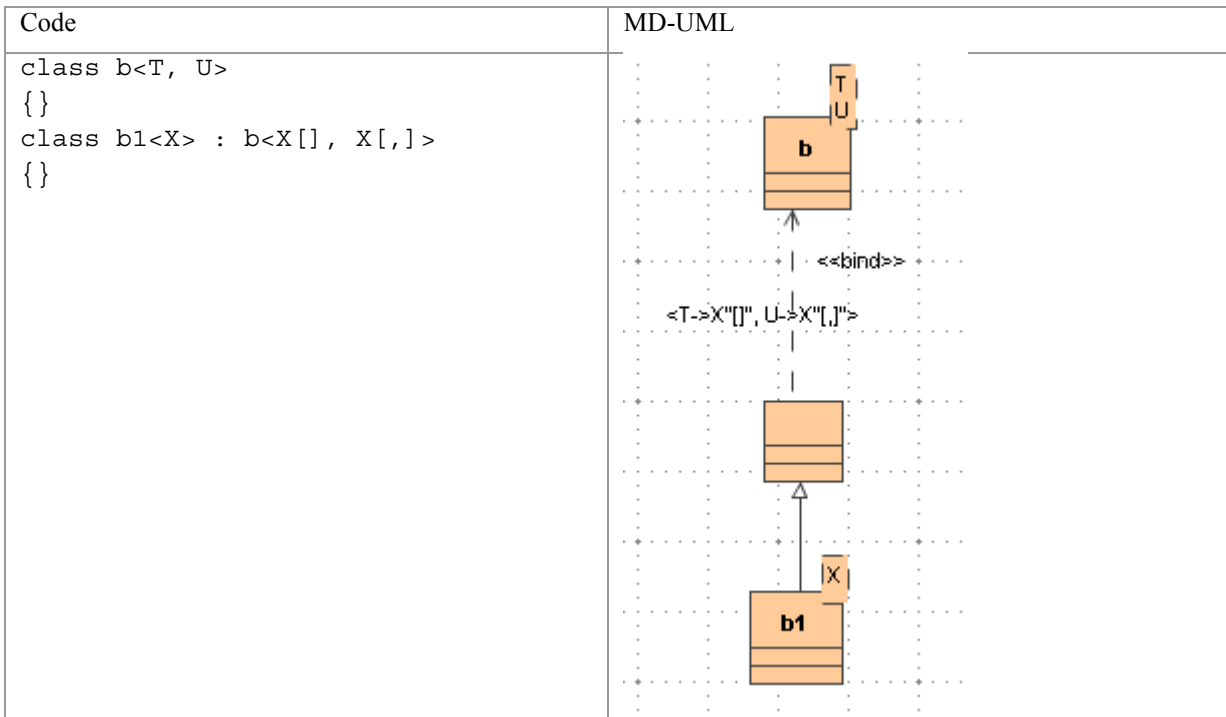


## Generic Overloading

Code	MD-UML
<pre> class X&lt;T&gt; {     public static explicit operator X&lt;T&gt;(T value)     { return null; }     public static implicit operator T(X&lt;T&gt; x)     { return x.item; }     public static explicit operator XXX&lt;int&gt;(X&lt;T&gt; x)     { return null; }     public static explicit operator X&lt;T&gt;(XXX&lt;int&gt; x)     { return null; }     public static X&lt;T&gt; operator ++(X&lt;T&gt; operand)      { return null; }     public static int operator &gt;&gt;(X&lt;T&gt; i, int c)      { return c; } } </pre>	



### Generic Parent Class



Code	MD-UML
<pre> class b&lt;T, U&gt; { } interface Ib&lt;B&gt; { } class b2&lt;Y&gt; : b&lt;int?[], string[]&gt; { } class c&lt;T&gt; : b&lt;int?, string&gt; { } class d&lt;X, Y&gt; : b&lt;X, Y&gt;, Ib&lt;int&gt; { } </pre>	

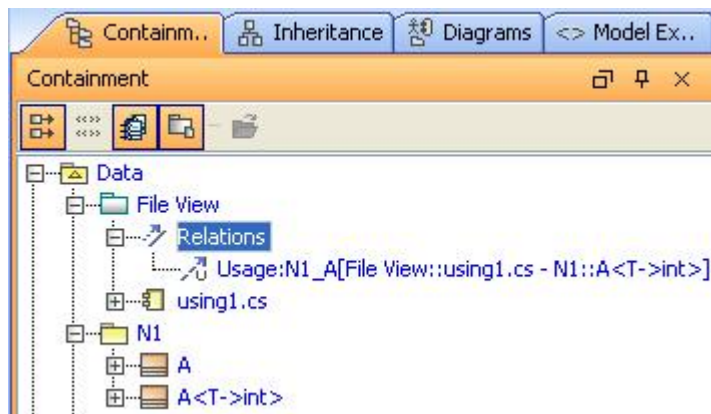
Code	MD-UML
<pre> class b&lt;T, U&gt; { } interface Ib&lt;B&gt; { } class g&lt;U&gt; : b&lt;b&lt;int, object&gt;, U&gt;, Ib&lt;string&gt; { } </pre>	

### Generic Using Alias

For example, using `N1_A = N1.A<int>`, we create template binding for `A<int>` in namespace `N1`, and then we create the Usage dependency from the parent component (in this case it is file component) to the template binding class.

For more information about the mapping of Using Directive, Using Directive Mapping.

Code	MD-UML
<pre>using N1_A = N1.A&lt;int&gt;; namespace N1 { public class A&lt;T&gt; {} } class A { N1_A a; }</pre>	



## Generic Constraints

Generic type and method declarations can optionally specify type parameter constraints by including *type parameter-constraints-clauses*.

*type-parameter-constraints-clauses:*

*type-parameter-constraints-clause*

*type-parameter-constraints-clauses type-parameter-constraints-clause*

*type-parameter-constraints-clause:*

**where** *type-parameter* : *type-parameter-constraints*

*type-parameter-constraints:*

*primary-constraint*

*secondary-constraints*

*constructor-constraint*

*primary-constraint* , *secondary-constraints*

*primary-constraint* , *constructor-constraint*

*secondary-constraints* , *constructor-constraint*

*primary-constraint* , *secondary-constraints* , *constructor-constraint*

*primary-constraint:*

*class-type*

`class`

`struct`

*secondary-constraints:*

*interface-type*

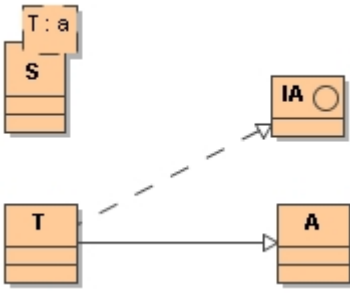
*type-parameter*

*secondary-constraints* , *interface-type*

*secondary-constraints* , *type-parameter*

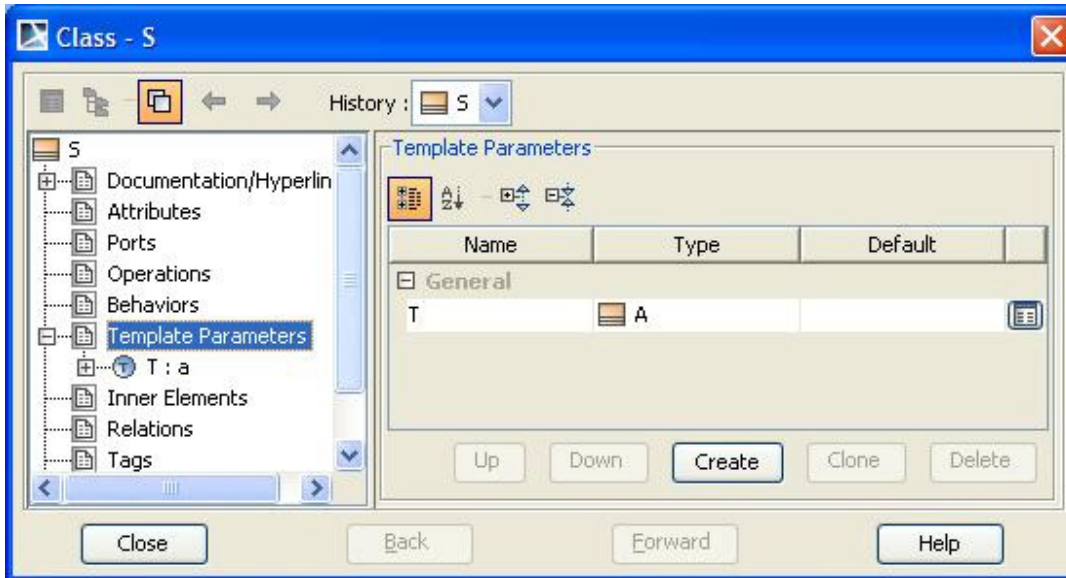
*constructor-constraint:*

`new ( )`

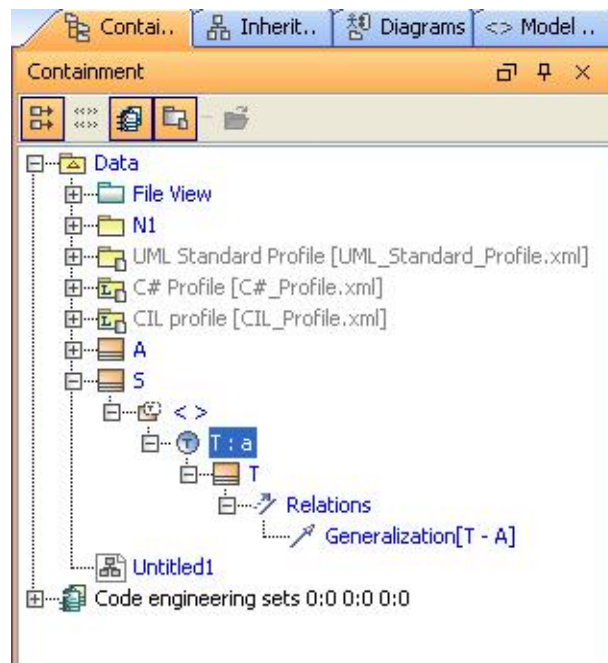
Code	MD-UML
<pre>public class A { } interface IA { } public class S&lt;T&gt; where T: A, IA{ }</pre>	 <p>The MD-UML diagram illustrates the relationships between the code elements. It features four nodes: a class <b>S</b> with a parameter <b>T: a</b>, a class <b>A</b>, an interface <b>IA</b>, and a parameter <b>T</b>. A solid arrow with an open arrowhead points from <b>T</b> to <b>A</b>, indicating that <b>T</b> inherits from <b>A</b>. A dashed arrow with an open arrowhead points from <b>T</b> to <b>IA</b>, indicating a dependency between <b>T</b> and <b>IA</b>. The <b>IA</b> node is represented as an interface with a small circle in its top-right corner.</p>



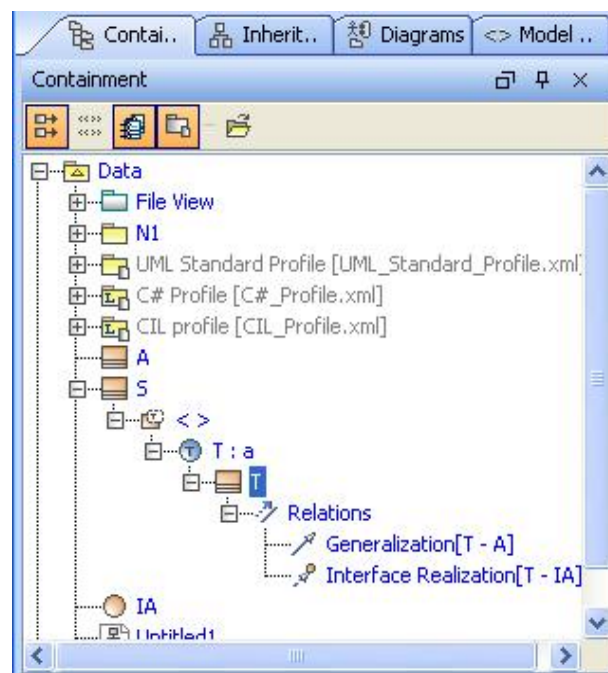
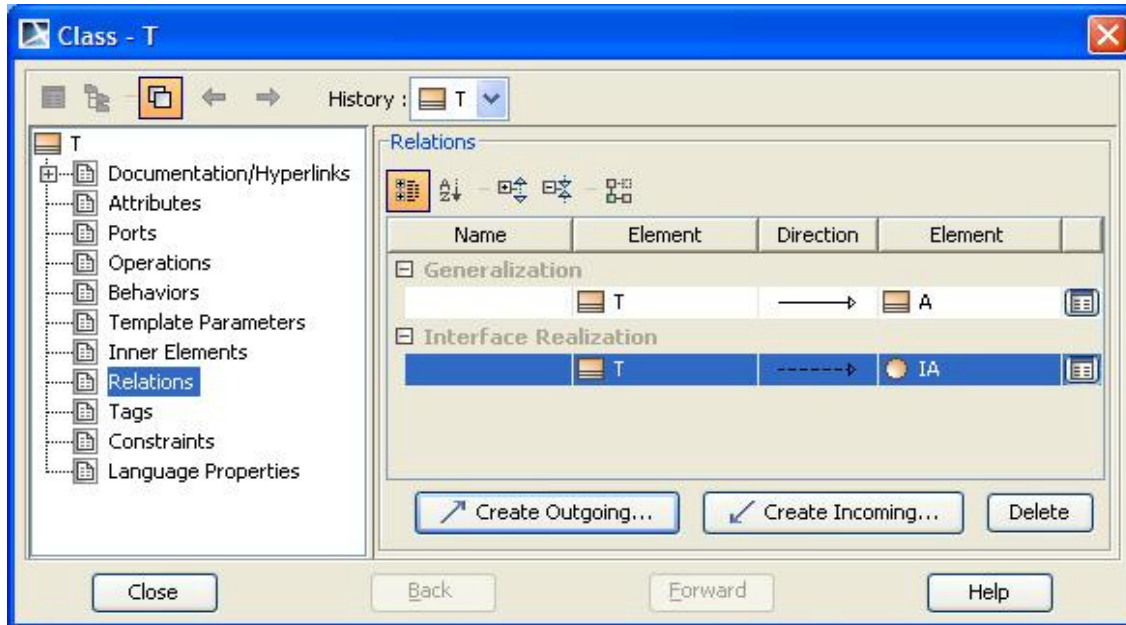
To create the first template constraint for template parameter T, we select the type of template parameter T as class A.



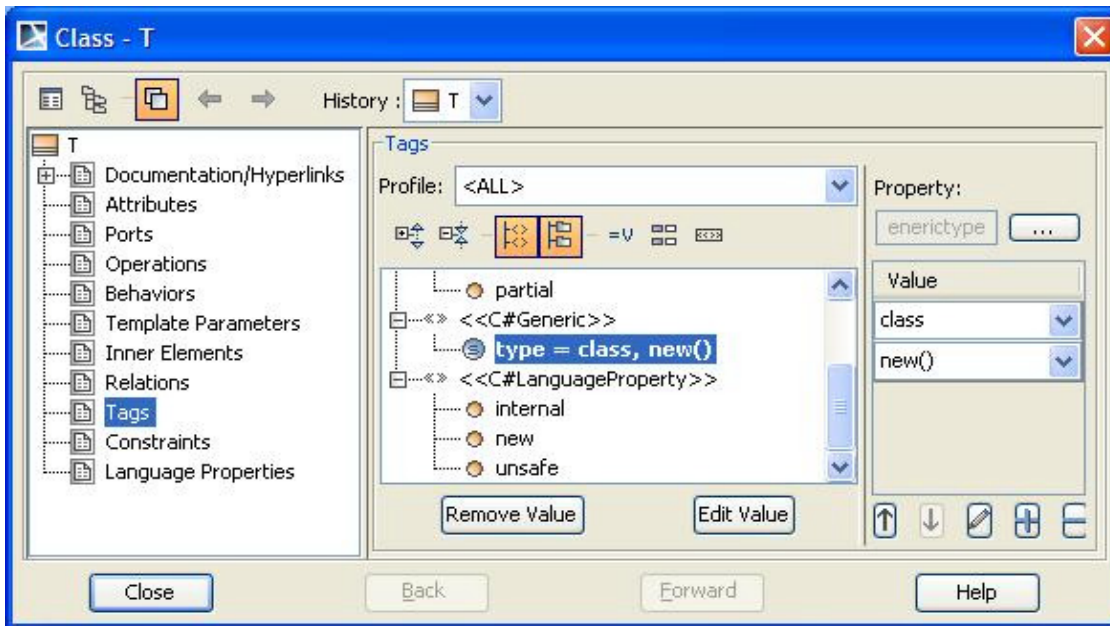
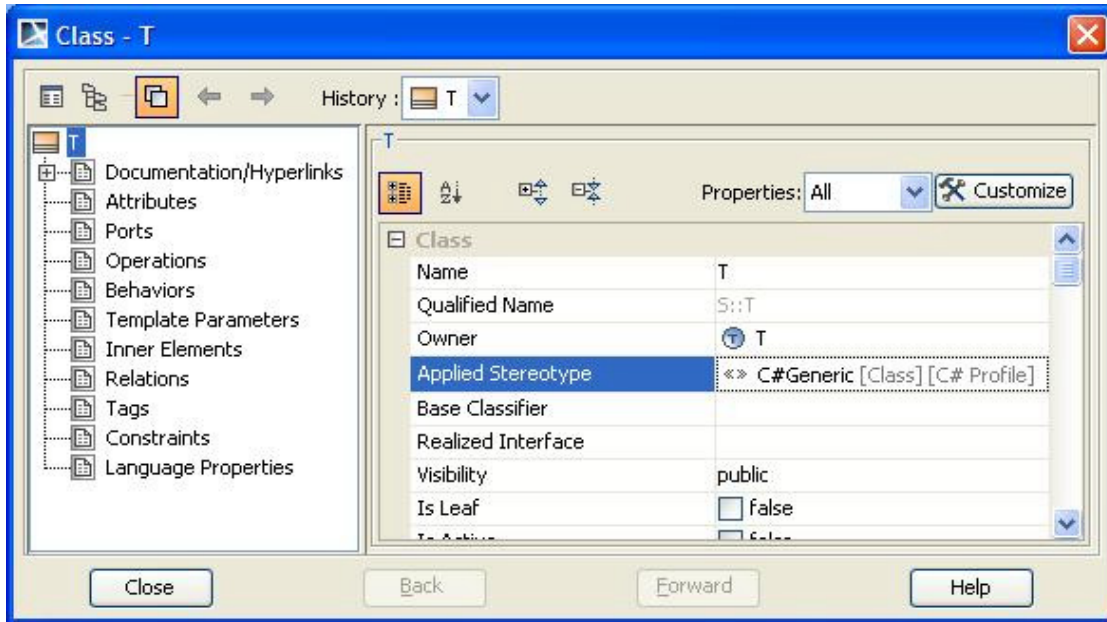
The constraint A is shown as generalization relation of T.



To create the second template constraint for template parameter T, Open the specification of template parameter T, and then create Interface Realization as Outgoing relation to the interface IA.



To create class, struct, and new() constraints, we apply <<C#Generic>> stereotype to the template parameter, and then we create the tag values.



## Anonymous Methods

In versions of C# previous to 2.0, the only way to declare a delegate was to use named methods. C# 2.0 introduces anonymous methods.

Creating anonymous methods is essentially a way to pass a code block as a delegate parameter.

### For example:

```
// Create a delegate instance
delegate void Del(int x);
```

```
// Instantiate the delegate using an anonymous method
```

```
Del d = delegate(int k) { /* ... */};
```

By using anonymous methods, you reduce the coding overhead in instantiating delegates by eliminating the need to create a separate method.

From now, there is no direct mapping for Anonymous Methods but it uses the mapping for delegate method feature.

## Partial Types

A new type modifier, `partial`, is used when defining a type in multiple parts. To ensure compatibility with existing programs, this modifier is different than other modifiers: like `get` and `set`, it is not a keyword, and it must appear immediately before one of the keywords `class`, `struct`, or `interface`.

*class-declaration:*

```
attributesopt class-modifiersopt partialopt class identifier type-parameter-listopt  
class-baseopt type-parameter-constraints-clausesopt class-body ;opt
```

*struct-declaration:*

```
attributesopt struct-modifiersopt partialopt struct identifier type-parameter-listopt  
struct-interfacesopt type-parameter-constraints-clausesopt struct-body ;opt
```

*interface-declaration:*

```
attributesopt interface-modifiersopt partialopt interface identifier type-parameter-listopt  
interface-baseopt type-parameter-constraints-clausesopt interface-body ;opt
```

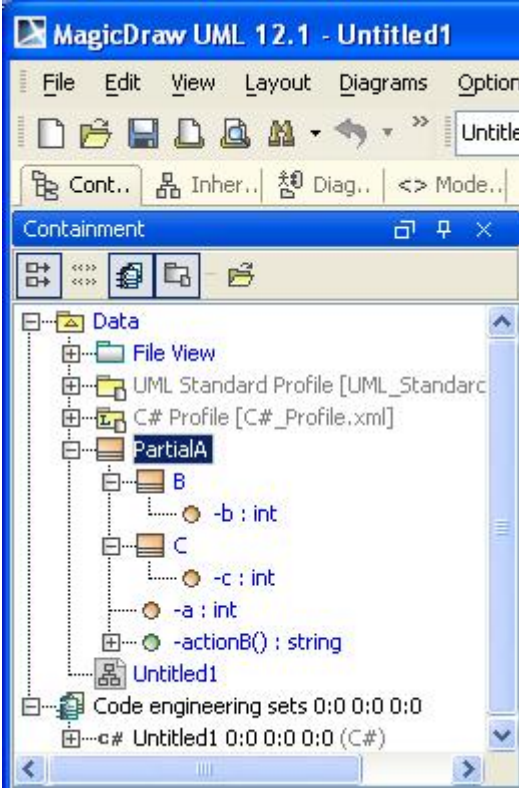
Each part of a partial type declaration must include a `partial` modifier and must be declared in the same namespace as the other parts. The `partial` modifier indicates that additional parts of the type declaration may exist elsewhere, but the existence of such additional parts is not a requirement. It is valid for just a single declaration of a type to include the `partial` modifier.

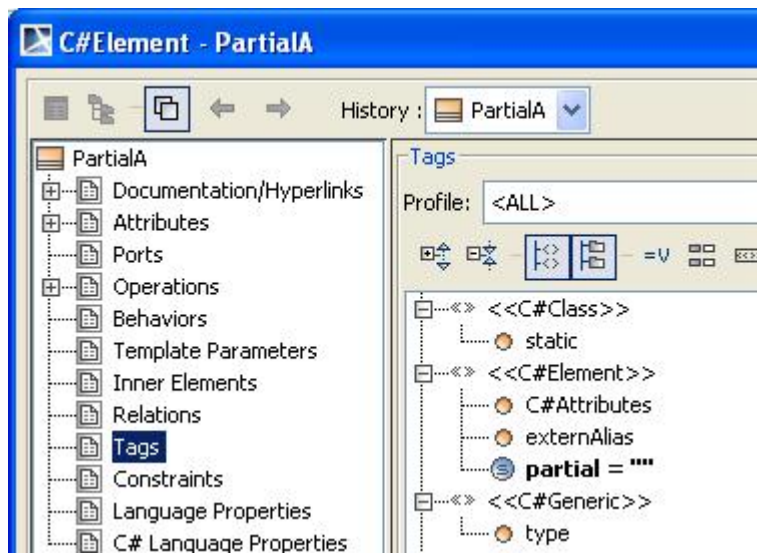
All parts of a partial type must be compiled together such that the parts can be merged at compile-time. Partial types specifically do not allow already compiled types to be extended.

Nested types may be declared in multiple parts by using the `partial` modifier. Typically, the containing type is declared using `partial` as well, and each part of the nested type is declared in a different part of the containing type.

The `partial` modifier is not permitted on delegate or enum declarations.

Code	MD-UML
<pre> //Case #1 //The partial class is written into one class file. public partial class PartialA{     int a;     string actionA()     {     } } public partial class PartialA{     int b;     string actionB()     {     } } </pre>	<div data-bbox="970 353 1158 533" style="border: 1px solid black; padding: 5px; margin: 10px auto; width: fit-content;"> <p style="text-align: center; margin: 0;"><b>PartialA</b></p> <hr style="border: 0; border-top: 1px solid black; margin: 2px 0;"/> <p style="margin: 0;">-a : int -b : int</p> <hr style="border: 0; border-top: 1px solid black; margin: 2px 0;"/> <p style="margin: 0;">-actionA() : string -actionB() : string</p> </div> <p>All child elements will be merged into one Class element.</p>
<pre> //Case #2 //The partial class is written into separate class file. //PartialA1.cs public partial class PartialA{     int a;     string actionA()     {     } }  //PartialA2.cs public partial class PartialA{     int b;     string actionB()     {     } } </pre>	<div data-bbox="970 880 1158 1059" style="border: 1px solid black; padding: 5px; margin: 10px auto; width: fit-content;"> <p style="text-align: center; margin: 0;"><b>PartialA</b></p> <hr style="border: 0; border-top: 1px solid black; margin: 2px 0;"/> <p style="margin: 0;">-a : int -b : int</p> <hr style="border: 0; border-top: 1px solid black; margin: 2px 0;"/> <p style="margin: 0;">-actionA() : string -actionB() : string</p> </div> <p>All child elements will be merged into one Class element, the same as both classes are written into one class file.</p>

Code	MD-UML
<pre> //Case #3 //The partial class with inner class public partial class PartialA{     public class B     {         int b;     } } public partial class PartialA{     int a;     string actionB()     {     }     public class C     {         int c;     } }         </pre>	 <p data-bbox="762 1137 1362 1200">All inner classes have to be located as inner class of the parent class (partial class).</p>

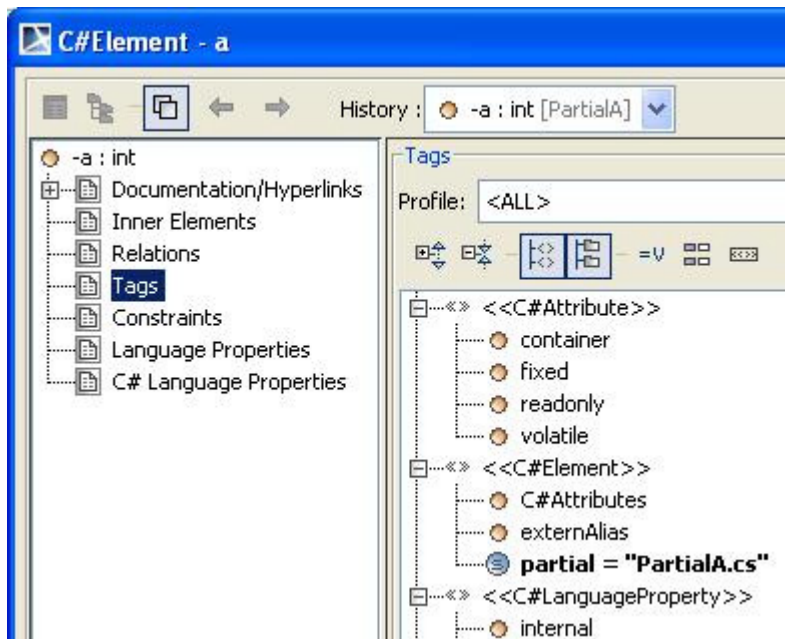


The "partial" tag will have a blank tagged value for Partial Class Element.



Code

MD-UML



The "partial" tag will have a value of the file name that the class belongs to for each child element in partial class element.

## Nullable Types

A nullable type is classified as a value type:

*value-type:*

*struct-type*

*enum-type*

*struct-type:*

*type-name*

*simple-type*

*nullable-type*

*nullable-type:*

*non-nullable-value-type* ?


*non-nullable-value-type:*

*type*

The type specified before the ? modifier in a nullable type is called the **underlying type** of the nullable type. The underlying type of a nullable type can be any non-nullable value type or any type parameter that is constrained to non-nullable value types (that is, any type parameter with a struct constraint). The underlying type of a nullable type cannot be a nullable type or a reference type.

A nullable type can represent all values of its underlying type plus an additional null value.

The syntax `T?` is shorthand for `System.Nullable<T>`, and the two forms can be used interchangeably.

Code	MD-UML
<pre>class class1 {     int? a = null;     System.Nullable a = null; }</pre>	 <pre>classDiagram     class Class1 {         -num : myType "?"     }     style Class1 fill:#f9f9f9,stroke:#ccc,stroke-width:1px</pre>

Add "Nullable" class type to C# profile.

## Accessor Declarations

The syntax for property accessors and indexer accessors is modified to permit an optional *accessor-modifier*:

*get-accessor-declaration*:

*attributes*<sub>opt</sub> *accessor-modifier*<sub>opt</sub> **get** *accessor-body*

*set-accessor-declaration*:

*attributes*<sub>opt</sub> *accessor-modifier*<sub>opt</sub> **set** *accessor-body*

*accessor-modifier*:

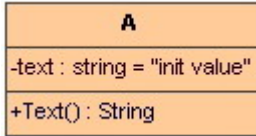
**protected**  
**internal**  
**private**  
**protected internal**  
**internal protected**

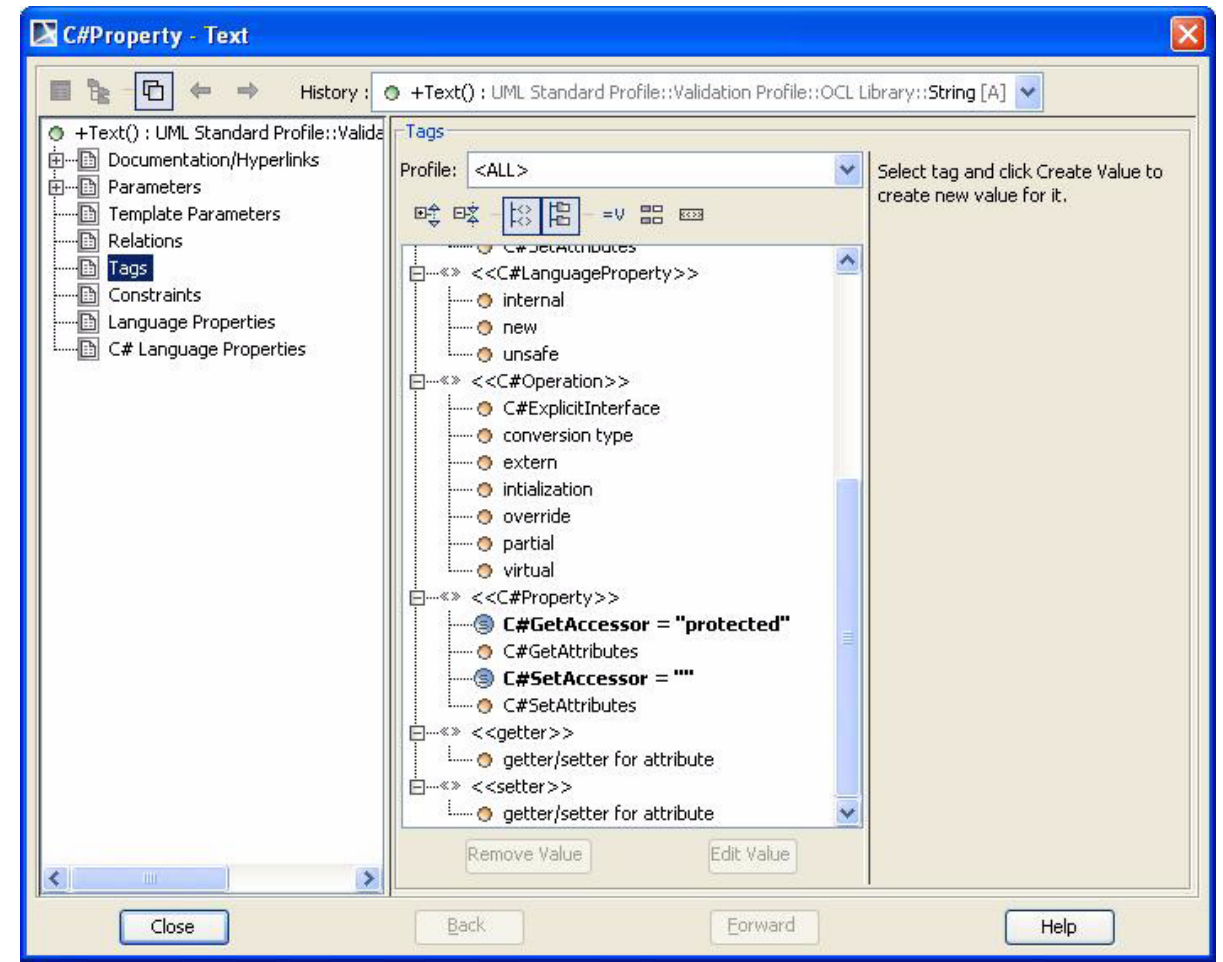
The use of *accessor-modifiers* is governed by the following restrictions:

- An *accessor-modifier* may not be used in an interface or in an explicit interface member implementation.
- For a property or indexer that has no *override* modifier, an *accessor-modifier* is permitted only if the property or indexer has both a **get** and **set** accessor, and then is permitted only on one of those accessors.
- For a property or indexer that includes an *override* modifier, an accessor must match the *accessor-modifier*, if any, of the accessor being overridden.
- The *accessor-modifier* must declare an accessibility that is strictly more restrictive than the declared accessibility of the property or indexer itself. To be precise:
  - If the property or indexer has a declared accessibility of **public**, any *accessor-modifier* may be used.
  - If the property or indexer has a declared accessibility of **protected internal**, the *accessor-modifier* may be either **internal**, **protected**, or **private**.
  - If the property or indexer has a declared accessibility of **internal** or **protected**, the *accessor-modifier* must be **private**.



- If the property or indexer has a declared accessibility of `private`, no *accessor-modifier* may be used.

Code	MD-UML
<pre> Class A { private string text = "init value"; public String Text {     protected get{ return text;}     set{ text = value;} } }         </pre>	



## Static Class

Static classes are classes that are not intended to be instantiated and which contain only static members. When a class declaration includes a static modifier, the class being declared is said to be a static class.

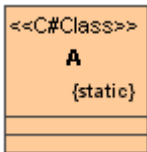
When a class declaration includes a static modifier, the class being declared is said to be a static class.

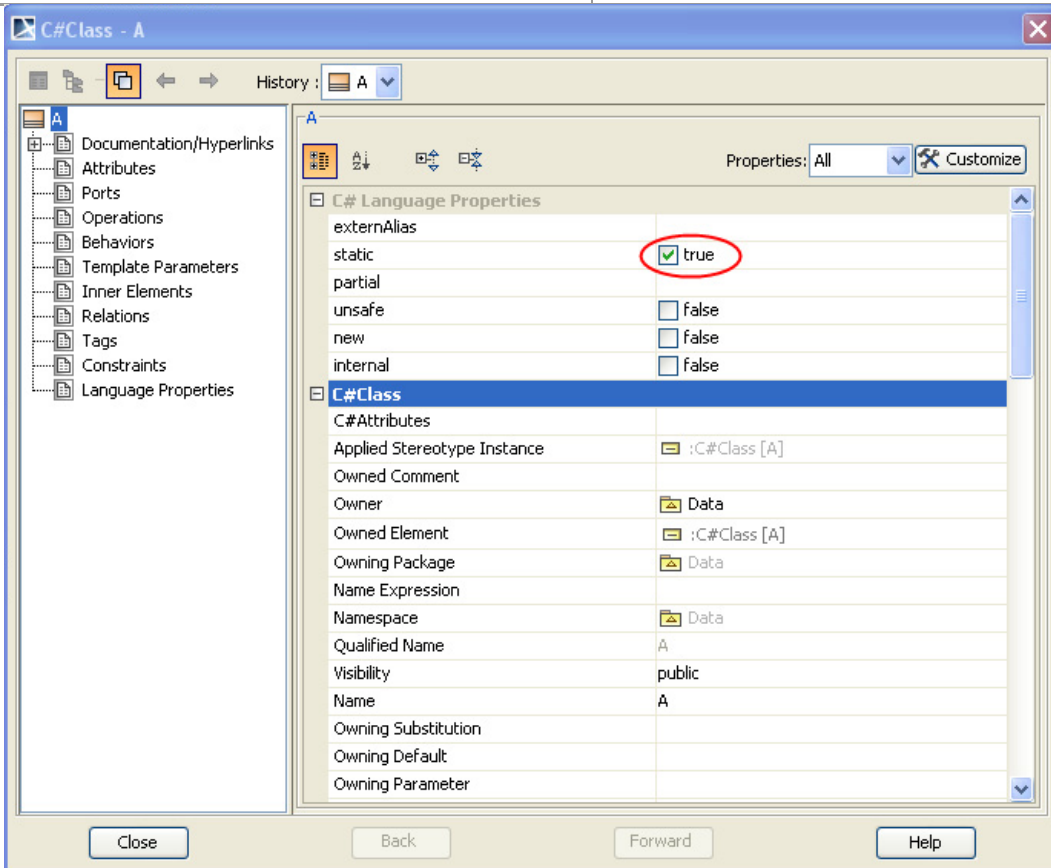
*class-declaration:*

*attributes*<sub>opt</sub> *class-modifiers*<sub>opt</sub> *partial*<sub>opt</sub> *class identifier type-parameter-list*<sub>opt</sub>  
*class-base*<sub>opt</sub> *type-parameter-constraints-clauses*<sub>opt</sub> *class-body* ;<sub>opt</sub>

*class-modifiers:*  
*class-modifier*  
*class-modifiers class-modifier*

*class-modifier:*  
 new  
 public  
 protected  
 internal  
 private  
 abstract  
 sealed  
 static

Code	MD-UML
<pre>static class A {     ... }</pre>	



## Extern Alias Directive

Until now, C# has supported only a single namespace hierarchy into which types from referenced assemblies and the current program are placed. Because of this design, it has not been possible to reference types with the same fully qualified name from different assemblies, a situation that arises when types are independently given the same name, or when a program needs to reference several versions of the same assembly. Extern aliases make it possible to create and reference separate namespace hierarchies in such situations.

An *extern-alias-directive* introduces an identifier that serves as an alias for a namespace hierarchy.

*compilation-unit:*

```
extern-alias-directivesopt using-directivesopt global-attributesopt
    namespace-member-declarationsopt
```

*namespace-body:*

```
{ extern-alias-directivesopt using-directivesopt namespace-member-declarationsopt }
```

*extern-alias-directives:*

```
extern-alias-directive
extern-alias-directives extern-alias-directive
```

*extern-alias-directive:*

```
extern alias identifier ;
```

Consider the following two assemblies:

Assembly a1.dll:

```
namespace N
{
    public class A {}
    public class B {}
}
```

Assembly a2.dll:

```
namespace N
{
    public class B {}
    public class C {}
}
```

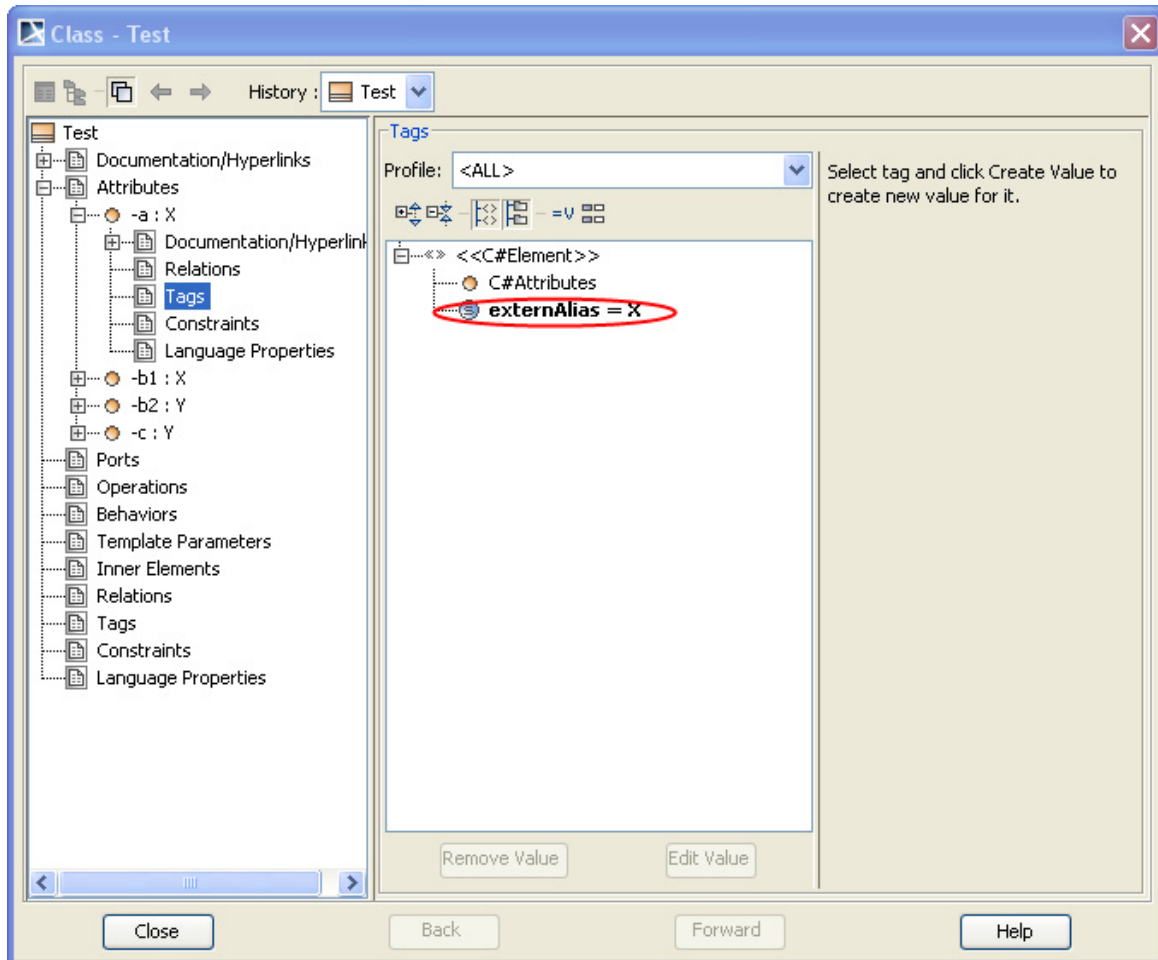
and the following program:

```
class Test
{
    N.A a;
    N.C c;
}
```

The following program declares and uses two extern aliases, X and Y, each of which represent the root of a distinct namespace hierarchy created from the types contained in one or more assemblies.

```
extern alias X;
extern alias Y;
class Test
{
    X::N.A a;
    X::N.B b1;
    Y::N.B b2;
    Y::N.C c;
}
```

Code	MD-UML
<pre>extern alias X; extern alias Y; class Test {     X::N.A a;     X::N.B b1;     Y::N.B b2;     Y::N.C c; }</pre>	<div style="border: 1px solid black; padding: 5px; background-color: #f9f9f9;"> <p style="text-align: center; margin: 0;"><b>Test</b></p> <pre>&lt;&lt;C#Element&gt;&gt;-a : X{externAlias = X} &lt;&lt;C#Element&gt;&gt;-b1 : X{externAlias = X} &lt;&lt;C#Element&gt;&gt;-b2 : Y{externAlias = Y} &lt;&lt;C#Element&gt;&gt;-c : Y{externAlias = Y}</pre> </div>



## Pragma Directives

The `#pragma` preprocessing directive is used to specify optional contextual information to the compiler. The information supplied in a `#pragma` directive will never change program semantics.

*pp-directive:*  
...  
*pp-pragma*

```

pp-pragma:
    whitespaceopt # whitespaceopt pragma whitespace pragma-body pp-new-line

pragma-body:
    pragma-warning-body

```

C# 2.0 provides `#pragma` directives to control compiler warnings. Future versions of the language may include additional `#pragma` directives.

## Pragma Warning

The `#pragma` warning directive is used to disable or restore all or a particular set of warning messages during compilation of the subsequent program text.

```

pragma-warning-body:
    warning whitespace warning-action
    warning whitespace warning-action whitespace warning-list

warning-action:
    disable
    restore

warning-list:
    decimal-digits
    warning-list whitespaceopt , whitespaceopt decimal-digits

```

A `#pragma warning` directive that omits the warning list affects all warnings. A `#pragma warning` directive that includes a warning list affects only those warnings that are specified in the list.

A `#pragma warning disable` directive disables all or the given set of warnings.

A `#pragma warning restore` directive restores all or the given set of warnings to the state that was in effect at the beginning of the compilation unit. Note that if a particular warning was disabled externally, a `#pragma warning restore` will not re-enable that warning.

The following example shows use of `#pragma warning` to temporarily disable the warning reported when obsolete members are referenced.

```

using System;
class Program
{
    [Obsolete]
    static void Foo() {}
    static void Main() {
#pragma warning disable 612
        Foo();
#pragma warning restore 612
    }
}

```

There is no code engineering mapping for Pragma Directives now.

### Fix Size Buffer

Fixed size buffers are used to declare “C style” in-line arrays as members of structs. Fixed size buffers are primarily useful for interfacing with unmanaged APIs. Fixed size buffers are an unsafe feature, and fixed size buffers can only be declared in unsafe contexts.

A fixed size buffer is a member that represents storage for a fixed length buffer of variables of a given type. A fixed size buffer declaration introduces one or more fixed size buffers of a given element type. Fixed size buffers are only permitted in struct declarations and can only occur in unsafe contexts.

struct-member-declaration:

...

fixed-size-buffer-declaration

fixed-size-buffer-declaration:

attributesopt fixed-size-buffer-modifiersopt fixed buffer-element-type  
 fixed-sized-buffer-declarators ;

fixed-size-buffer-modifiers:

fixed-size-buffer-modifier

fixed-sized-buffer-modifier fixed-size-buffer-modifiers

fixed-size-buffer-modifiers:

new

public

protected

internal

private

unsafe

buffer-element-type:

type

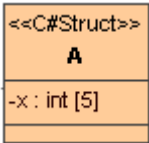
fixed-sized-buffer-declarators:

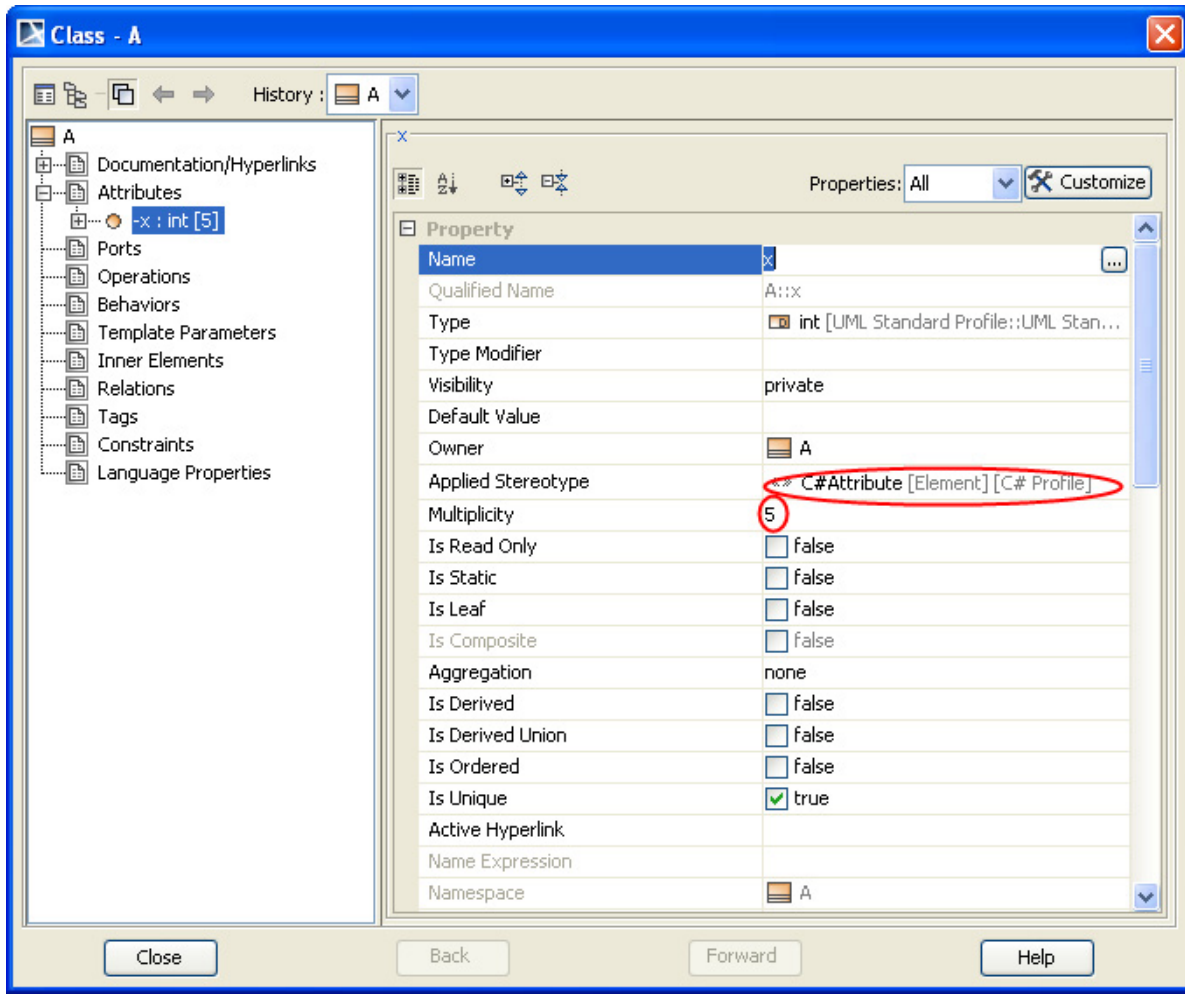
fixed-sized-buffer-declarator

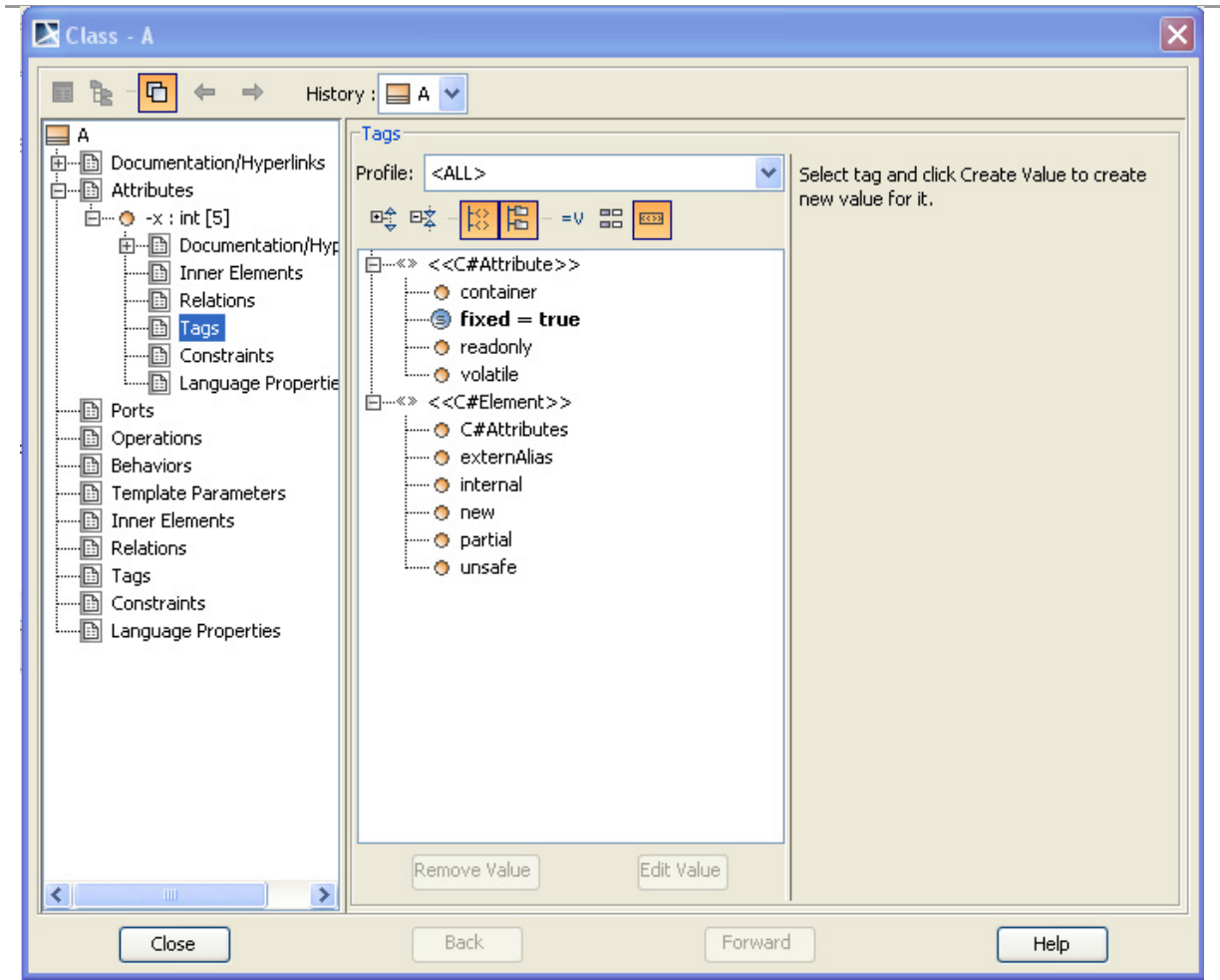
fixed-sized-buffer-declarator fixed-sized-buffer-declarators

fixed-sized-buffer-declarator:

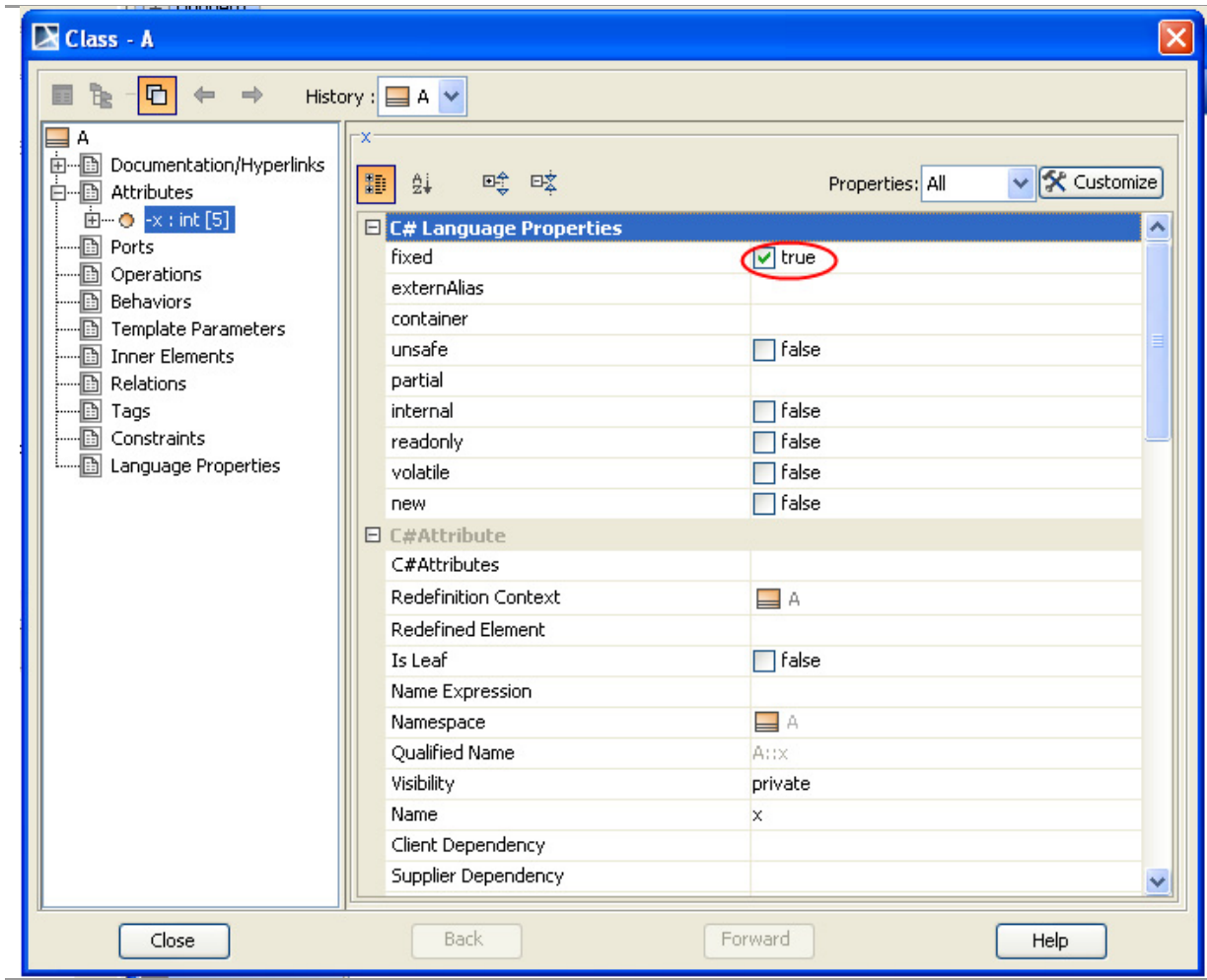
identifier [ const-expression ]

Code	MD-UML
<pre>unsafe struct A {     public fixed int x[5]; }</pre>	 <pre>&lt;&lt;C#Struct&gt;&gt; <b>A</b> -x : int [5]</pre>









## C# 3.0 Description

### Extension Methods

**Extension methods** are static methods that can be invoked using instance method syntax. In effect, extension methods make it possible to extend existing types and constructed types with additional methods.

Extension methods are declared by specifying the keyword `this` as a modifier on the first parameter of the methods. Extension methods can only be declared in static classes.

The sample code :

```
public static class Extensions
{
    public static int ToInt32(this string s) {
        return Int32.Parse(s);
    }

    public static T[] Slice<T>(this T[] source, int index, int count) {
        if (index < 0 || count < 0 || source.Length - index < count)
            throw new ArgumentException();
        T[] result = new T[count];
        Array.Copy(source, index, result, 0, count);
    }
}
```

```

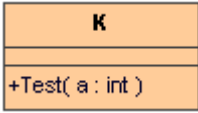
        return result;
    }
}
    
```

It becomes possible to invoke the extension methods in the static class `Extensions` using instance method syntax:

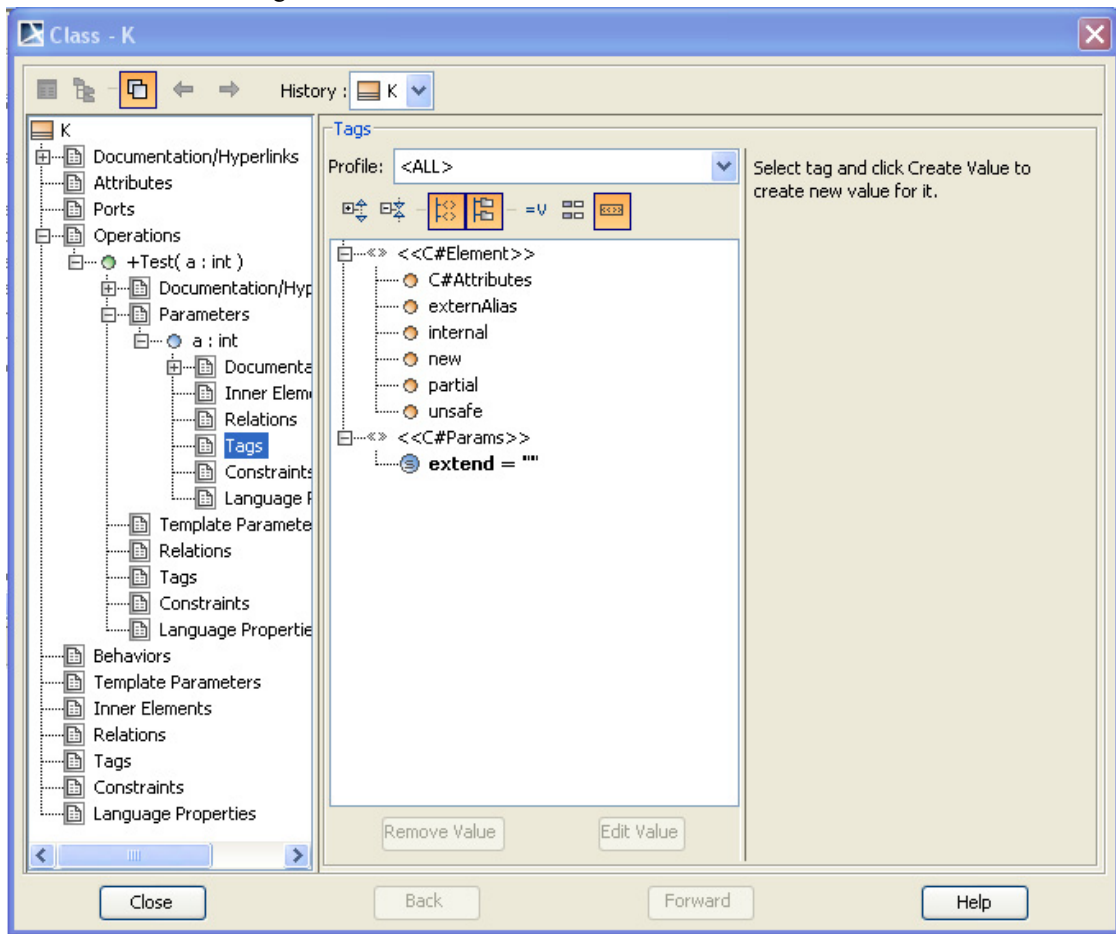
```

string s = "1234";
int i = s.ToInt32();// Same as Extensions.ToInt32(s)

int[] digits = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
int[] a = digits.Slice(4, 3);// Same as Extensions.Slice(digits, 4, 3)
    
```

Code	MD-UML
<pre> class K {     Test(this int a)     {} }                 </pre>	

The value is created to tag “extend” in <<C#Params>>



## Lambda Expression Conversion

A lambda expression is written as a parameter list, followed by the `=>` token, followed by an expression or a statement block.

The parameters of a lambda expression can be explicitly or implicitly typed. In an explicitly typed parameter list, the type of each parameter is explicitly stated. In an implicitly typed parameter list, the types of the parameters are inferred from the context in which the lambda expression occurs.

Some examples of lambda expressions are below:

```
x => x + 1 // Implicitly typed, expression body
x => { return x + 1; } // Implicitly typed, statement body
(int x) => x + 1 // Explicitly typed, expression body
(int x) => { return x + 1; } // Explicitly typed, statement body
(x, y) => x * y // Multiple parameters
() => Console.WriteLine() // No parameters
```

There is no mapping for this feature.

## C# Profile

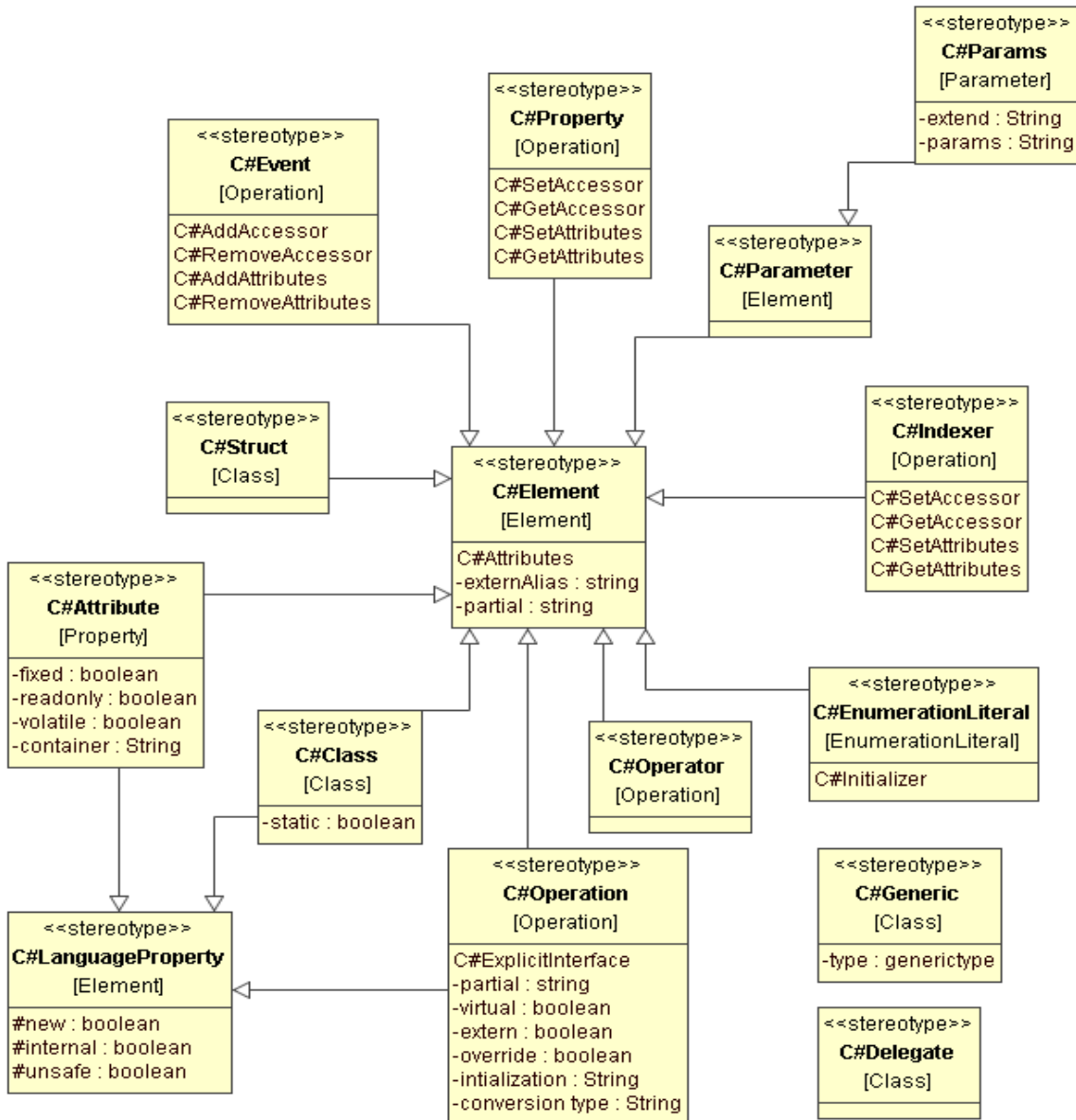


Figure 133 -- C# Profile

The above diagram shows the class diagram of C# profile. The C# profile package contains many stereotype and related tagged value to contain the properties of C# language usage. The new stereotypes and tagged value are shown in the following part.

### Stereotype

C# Profile contains many stereotypes and tagged definitions. When the mapping begins, there will be tagged value applied to tagged definition in stereotype. Since MagicDraw 12.0 the information of language properties is moved to tagged value in stereotype. The information of language properties of:

- Class is moved to <<C#Class>>
- Attribute is moved to <<C#Attribute>>

- Operation is moved to <<C#Operation>>
- The common information of the old language properties (Class, Attribute and Operation) is moved to C#LanguageProperty

This document represents only newly stereotyped and tagged definitions, the details are shown below.

## C#Attribute

<<C#Attribute>> is an invisible stereotype that is used to define C# attribute properties. This is used to store the language properties of the attribute in C#.

Name	Meta class	Constraints
C#Attribute	Property	

Tag	Type	Description
container	String	
fixed	Boolean	Represents the usage of Fixed Size Buffer <pre>unsafe struct A {     public <b>fixed</b> int x[5]; }</pre>
readonly	Boolean	Represents the usage of read-only attribute <pre>Class A {     public <b>readonly</b> int x; }</pre>
volatile	Boolean	Represents the usage of volatile attribute <pre>Class A {     public <b>volatile</b> int x; }</pre>

## C#Class

<<C#Class>> is a stereotype that is used to define property of C# class. This is used to store the language properties of the class in C#. Moreover this class is used for store the information about the static class too.

Name	Meta class	Constraints
C#Class	Class	

Tag	Type	Description
Static	boolean	Represents the usage of static class <pre>static Class A {     public int x; }</pre>

### C#Delegate

<<C#Delegate>> is a stereotype that indicates that model class represents C# delegate.

Name	Meta class	Constraints
C#Delegate	Class	

### C#Element

<<C#Element>> is a stereotype that is used to define properties of all element in model.

Name	Meta class	Constraints
C#Element	Element	This stereotype can be applied to all elements such as class, attribute, operation and parameter.

Tag	Type	Description
externAlias	String	Represents the usage of extern alias <pre>extern alias X; class Test {     X::N.A a; }</pre>
partial	String	
C#Attributes	String	Represents C# attributes for element.

### C#EnumerationLiteral

<<C#EnumerationLiteral>> is a stereotype that is used to define enumeration.

Name	Meta class	Constraints
C#EnumerationLiteral	EnumerationLiteral	

Tag	Type	Description
C#Initializer	String	Represents enumeration member's constant value.

### C#Event

<<C#Event>> is a stereotype that is used to indicate that operation represents C# event.

Name	Meta class	Constraints
C#Event	Operation	

Tag	Type	Description
C#AddAccessor	String	Adds <i>add</i> accessor for event
C#AddAttributes	String	Defines C# attributes for <i>add</i> accessor.
C#RemoveAccessor	String	Adds <i>remove</i> accessor for event
C#RemoveAttributes	String	Defines C# attributes for <i>remove</i> accessor.

### C#Generic

<<C#Generic>> is a stereotype that is used to define generic properties.

Name	Meta class	Constraints
C#Generic	Class	

Tag	Type	Description
type	generictype	Defines type for generic constraint parameter, class, struct, and new()

### C#Indexer

<<C#Indexer>> is a stereotype that is used to indicate that operation represents C# indexer.

Name	Meta class	Constraints
C#Indexer	Operation	

Tag	Type	Description
C#GetAccessor	String	Adds <i>get</i> accessor for indexer.
C#GetAttributes	String	Defines C# attributes for <i>get</i> accessor.
C#SetAccessor	String	Adds <i>set</i> accessor for indexer.
C#SetAttributes	String	Defines C# attributes for <i>set</i> accessor.

### C#LanguageProperty

<<C#LanguageProperty>> is the parent of <<C#Class>>, <<C#Attribute>>, and <<C#Operation>>, So the <<C#Class>>, <<C#Attribute>> and <<C#Operation>> also have it's tag definition.

Name	Meta class	Constraints
C#LanguageProperty	Element	

Tag	Type	Description
internal	boolean	
new	boolean	

Tag	Type	Description
unsafe	boolean	Represents the usage of unsafe element <pre>unsafe struct A {     public fixed int x[5]; }</pre>

### C#Operation

<<C#Operation>> is a stereotype which is used to define properties of the operation. This is used to store the language properties of the operation in C#.

Name	Meta class	Constraints
C#Operation	Operation	

Tag	Type	Description
conversion type	String	
extern	boolean	Represent the usage of extern operation Class A <pre>{     public extern int x(); }</pre>
initialization	String	
override	boolean	
partial	String	
virtual	boolean	
C#ExplicitInterface	Classifier	Defines C# explicit interface for explicit interface member implementation.

### C#Operator

<<C#Operator>> is a stereotype that is used to indicate that the operation represents C# operator.

Name	Meta class	Constraints
C#Operator	Operation	

### C#Parameter

<<C#Parameter>> is a stereotype that is used to indicate that the element represents C#Parameter.

Name	Meta class	Constraints
C#Parameter	Element	

### C#Params

<<C#Params>> is a stereotype that is used to indicate parameter arrays.

Name	Meta class	Constraints
C#Params	Parameter	

Tag	Type	Description
extend	String	
params	String	



## C#Property

<<C#Property>> is a stereotype that is used to indicate that the operation represents C# property.

Name	Meta class	Constraints
C#Property	Operation	

Tag	Type	Description
C#GetAccessor	String	Adds <i>get</i> accessor for indexer.
C#GetAttributes	String	Defines C# attributes for <i>get</i> accessor.
C#SetAccessor	String	Adds <i>set</i> accessor for indexer.
C#SetAttributes	String	Defines C# attributes for <i>set</i> accessor.

## C#Struct

<<C#Struct>> is a stereotype that is used to indicate that the model class represents C# structure.

Name	Meta class	Constraints
C#Struct	Class	

## C#Using

<<C#Using>> is a stereotype that is used to indicate that the usage dependency is C# Using directive.

Name	Meta class	Constraints
C#Using	Association, Realization, Usage	The clients of dependency are Component, and Namespace.

## Data Type

In addition to stereotype and tagged value, C# datatype should be created in the profile. The following diagram shows the data type of C# profile.

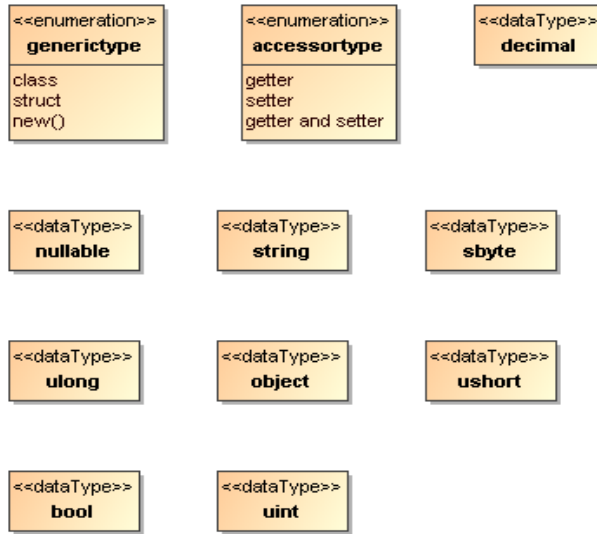


Figure 134 -- Data Type of C# Profile

## Conversion from old project version

This chapter describes the project converter of MD project version less than 11.6.

### Translation Activity Diagram

There are projects that use C# language properties, which need to be translated with version of MagicDraw project less than or equal to 11.6.

## Open local project

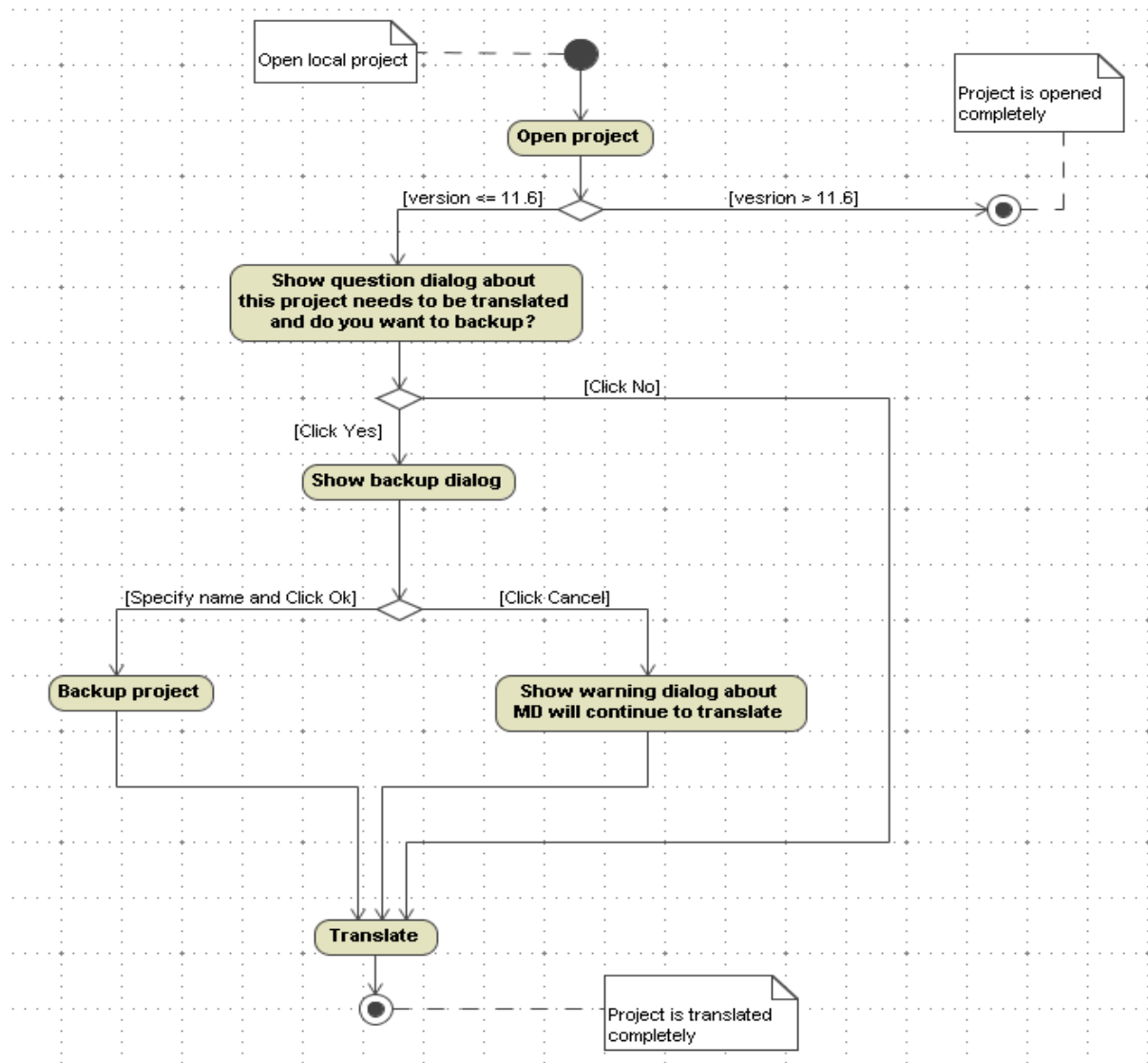


Figure 135 -- Open local project Activity Diagram

### Open teamwork project

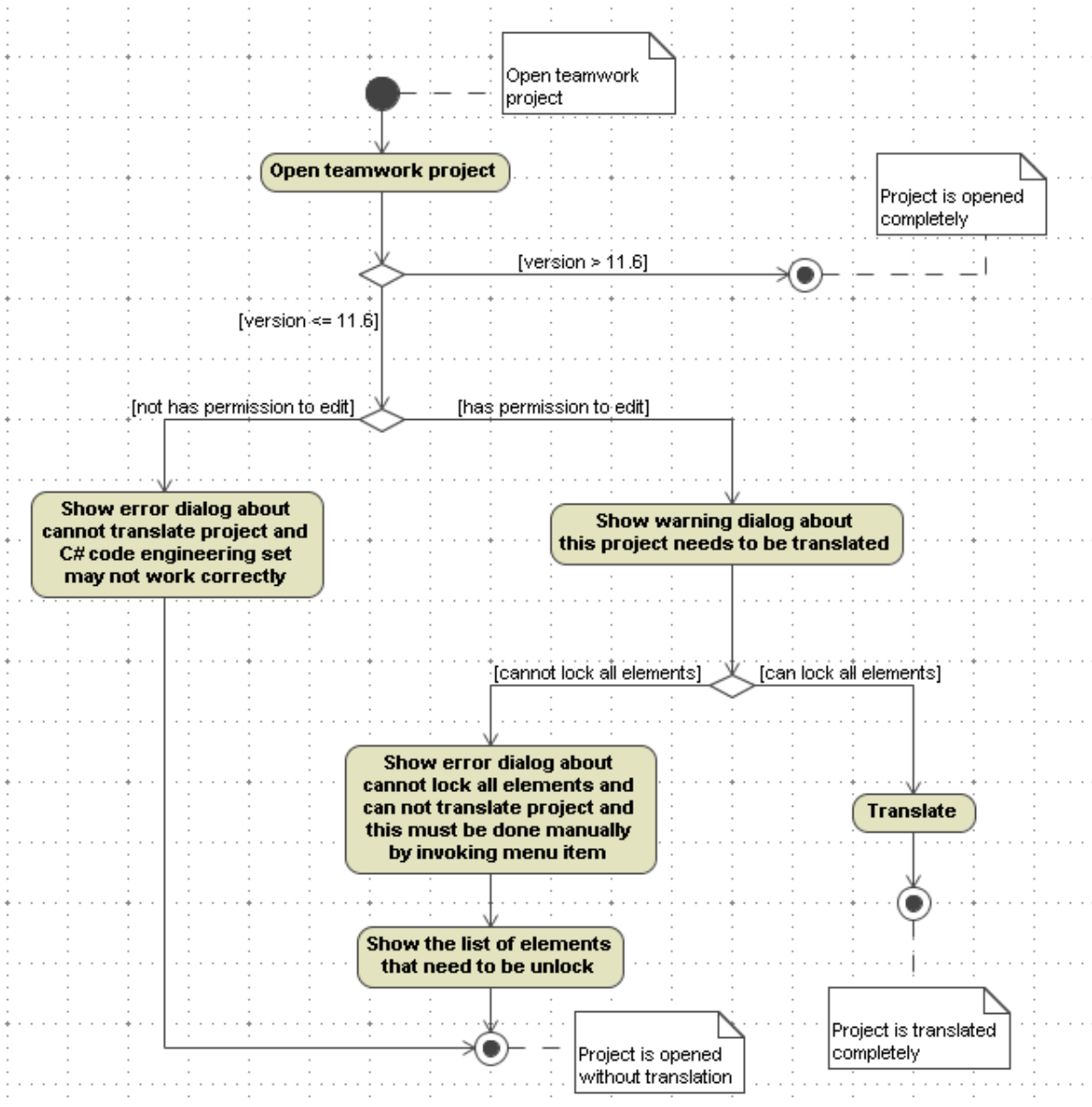


Figure 136 -- Open teamwork project Activity Diagram

### Import MagicDraw project

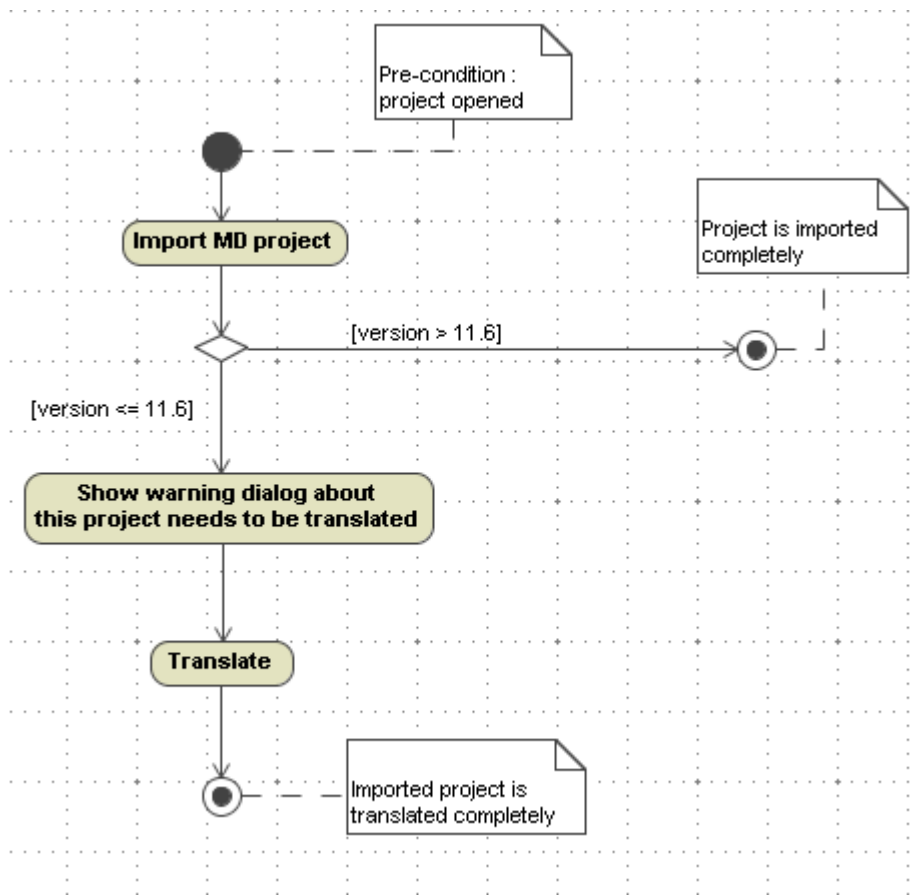


Figure 137 -- Import MagicDraw project Activity Diagram

## Use module

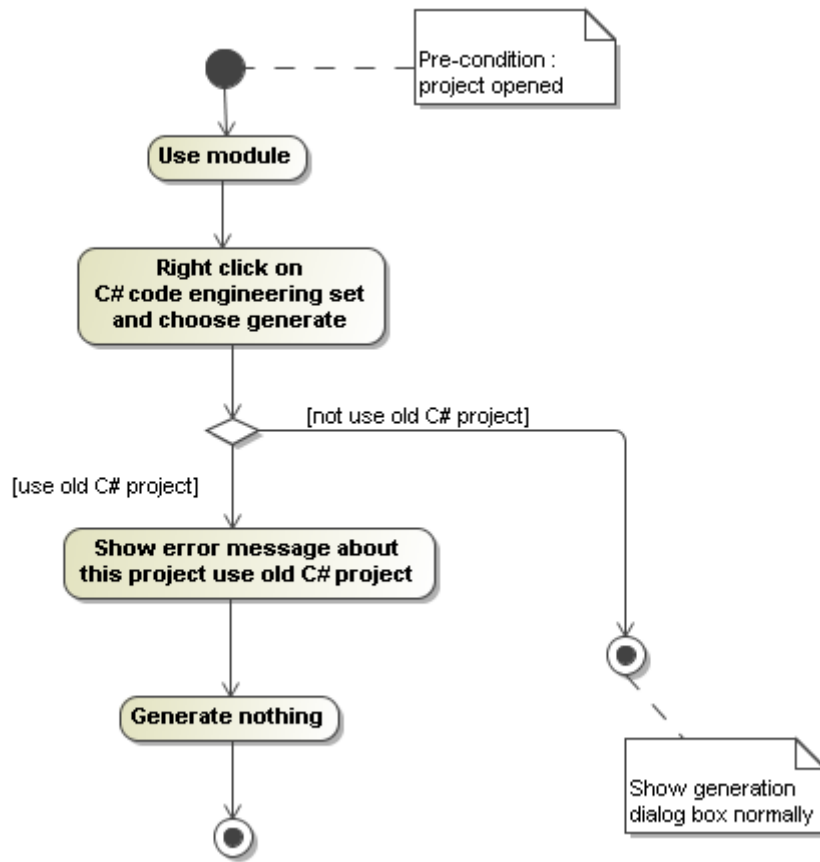


Figure 138 -- Use module Activity Diagram

## Update C++ Language Properties and Profiles

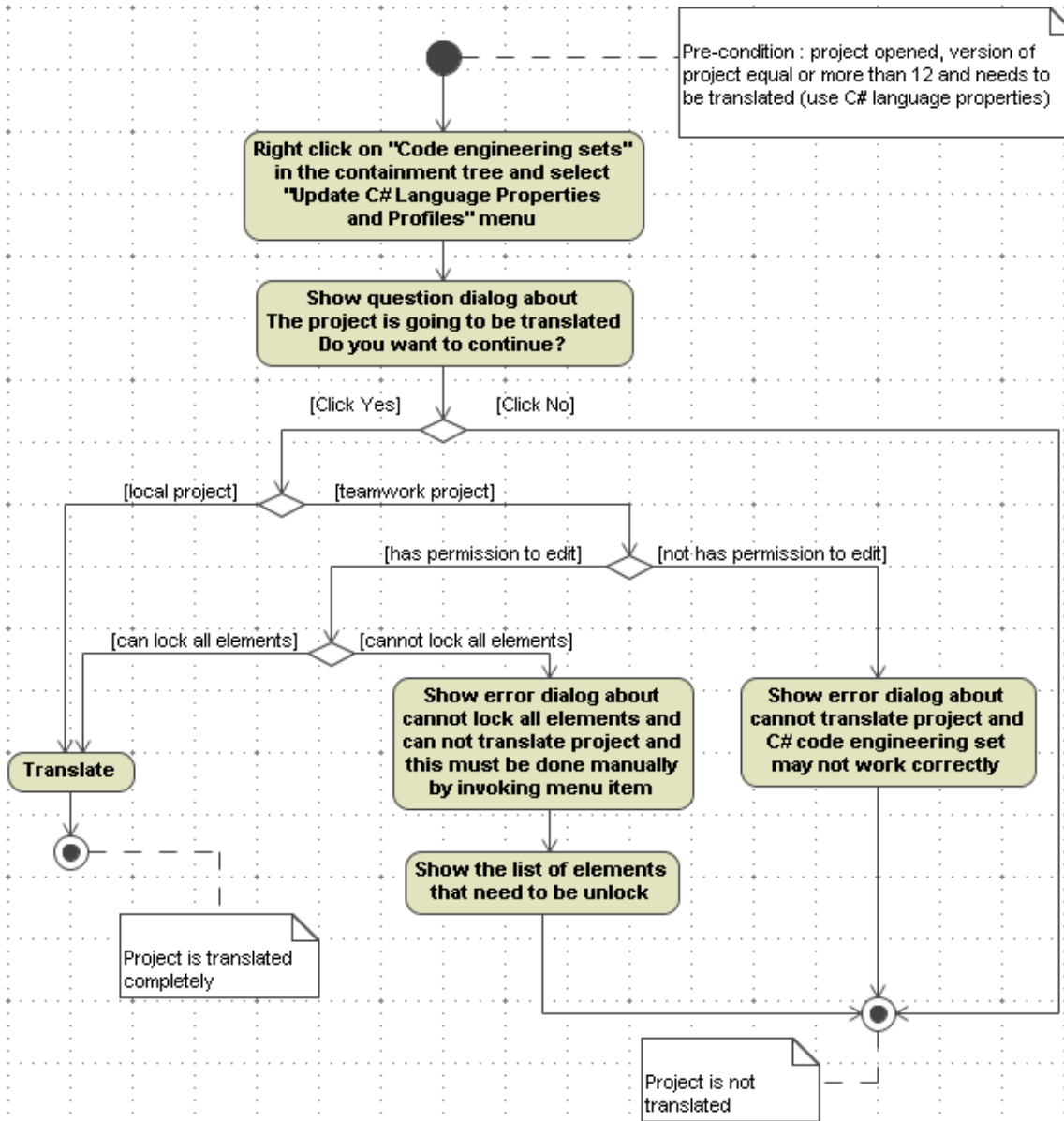


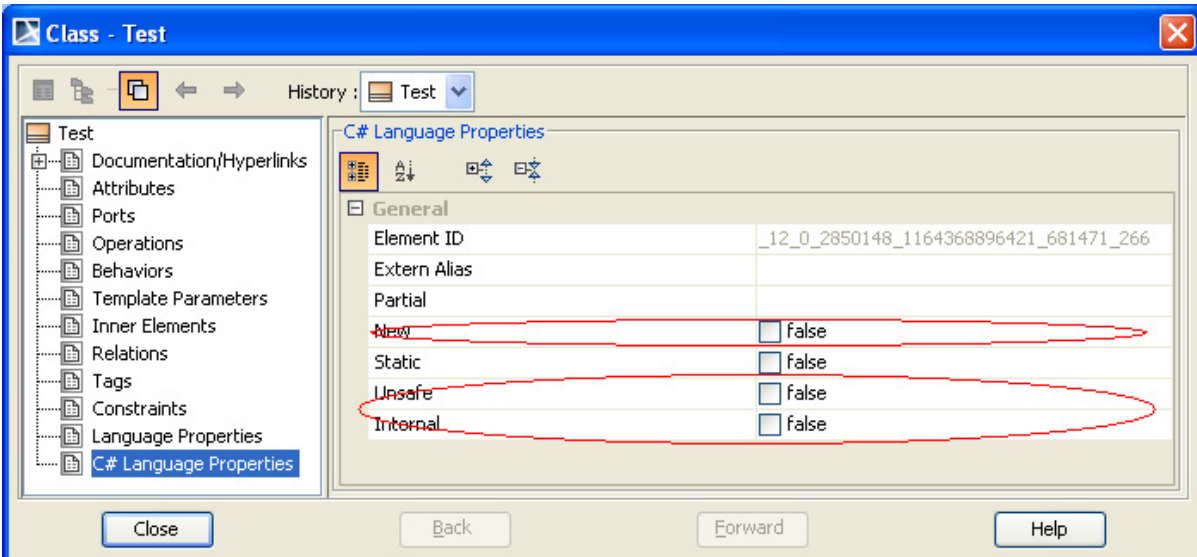
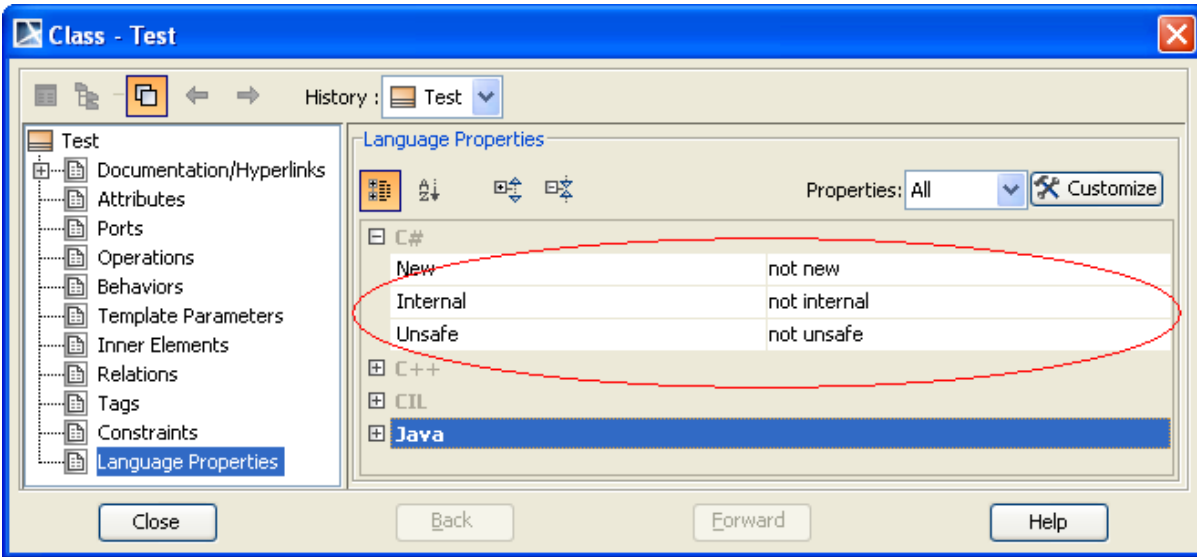
Figure 139 -- Update C# Language Properties and Profiles Activity Diagram

## Mapping

### Language Properties Mapping

Until MD version 11.6, language properties are stored in a specific format, since MD version 12 language properties are moved to stereotype tag value and using DSL to customize to C# Language Properties. (The language properties will move to C# Language Properties)

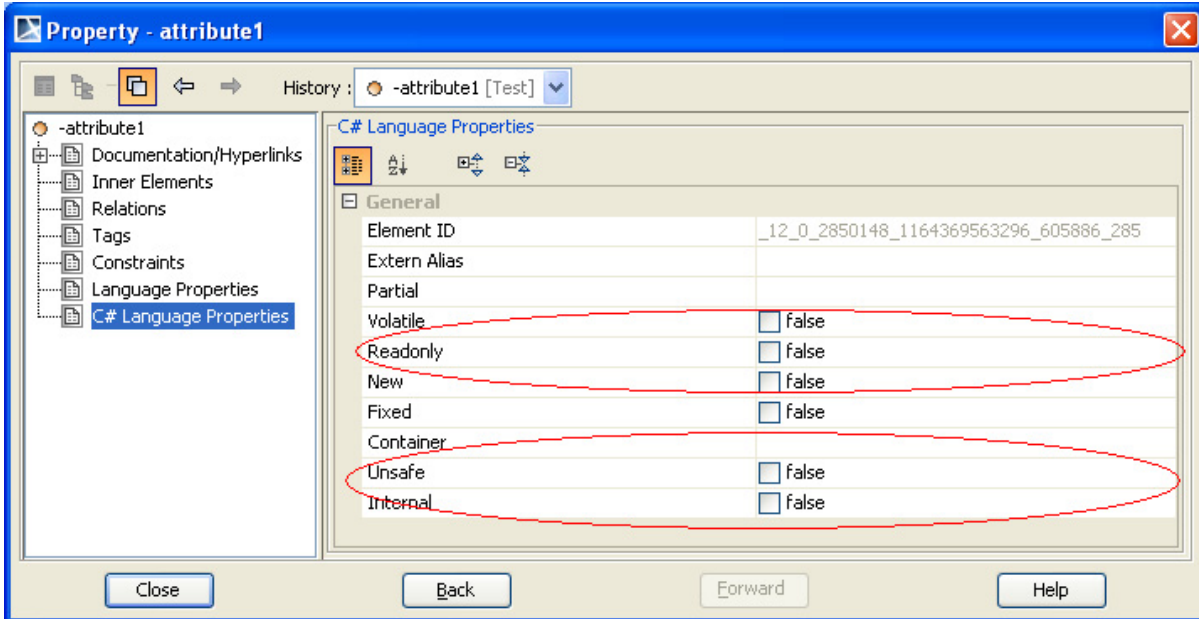
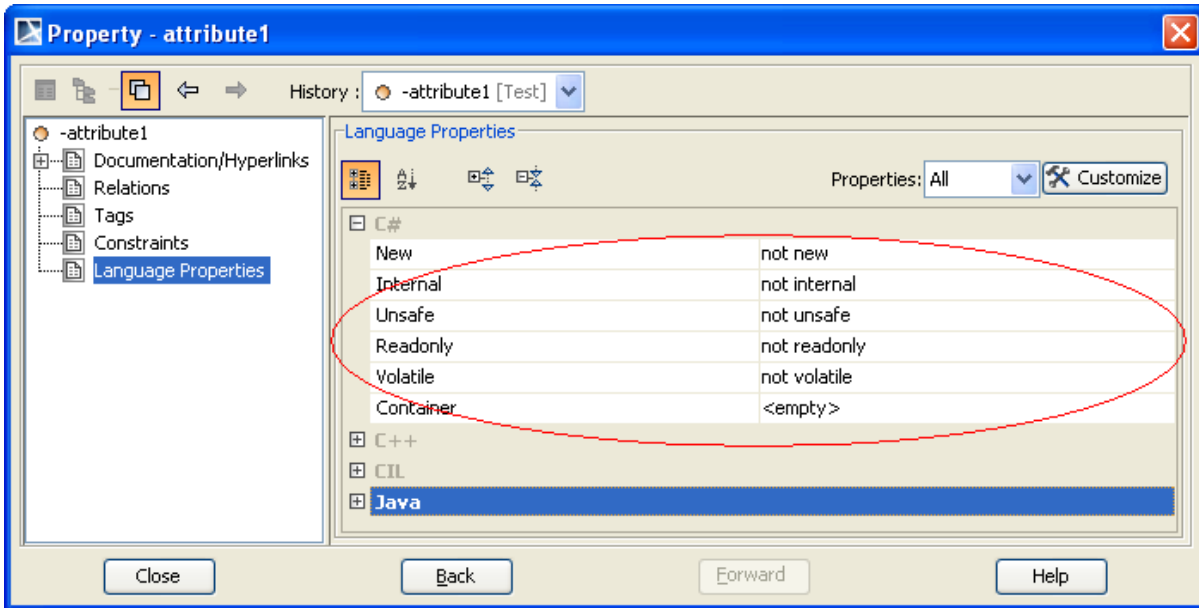
**Class**



Old Value	Translation
New	Mapped to tag "new" of <<C#LanguageProperty>> (The usage for this is to apply <<C#Class>> and set value to tag "new")
Internal	Mapped to tag "internal" of <<C#LanguageProperty>> (The usage for this is to apply <<C#Class>> and set value to tag "internal")
Unsafe	Mapped to tag "unsafe" of <<C#LanguageProperty>> (The usage for this is to apply <<C#Class>> and set value to tag "unsafe")

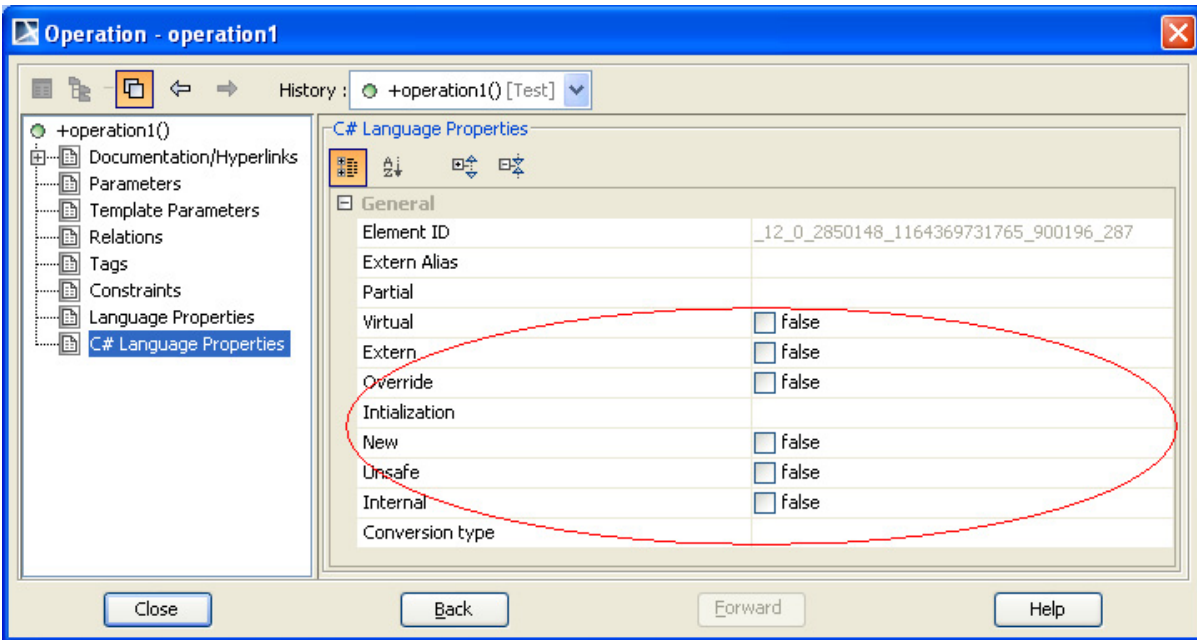
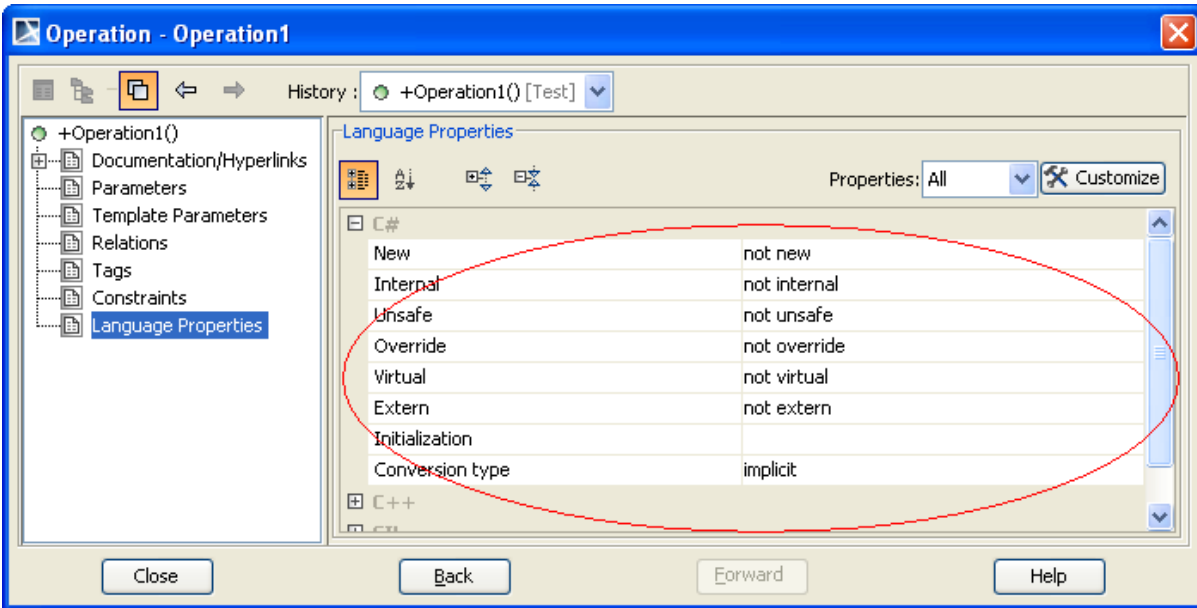


## Attribute



Old Value	Translation
New	Mapped to tag "new" of <<C#LanguageProperty>> (The usage for this is to apply <<C#Attribute>> and set value to tag "new")
Internal	Mapped to tag "internal" of <<C#LanguageProperty>> (The usage for this is to apply <<C#Attribute>> and set value to tag "internal")
Unsafe	Mapped to tag "unsafe" of <<C#LanguageProperty>> (The usage for this is to apply <<C#Attribute>> and set value to tag "unsafe")
Readonly	Mapped to "readonly" of <<C#Attribute>>
Volatile	Mapped to "volatile" of <<C#Attribute>>
Container	Mapped to "container" of <<C#Attribute>>

## Operation



Old Value	Translation
New	Mapped to tag "new" of <<C#LanguageProperty>> (The usage for this is to apply <<C#Operation>> and set value to tag "new")
Internal	Mapped to tag "internal" of <<C#LanguageProperty>> (The usage for this is to apply <<C#Operation>> and set value to tag "internal")
Unsafe	Mapped to tag "unsafe" of <<C#LanguageProperty>> (The usage for this is to apply <<C#Operation>> and set value to tag "unsafe")
Override	Mapped to tag "override" of <<C#Operation>>
Virtual	Mapped to tag "virtual" of <<C#Operation>>
Extern	Mapped to tag "extern" of <<C#Operation>>
Initialization	Mapped to tag "initialization" of <<C#Operation>>
Conversion type	Mapped to tag "conversion type" of <<C#Operation>>

## C# Properties Customization

DSL Customization is adopted to hide UML mapping, so extensions in MagicDraw should look like they are standard elements. DSL properties should appear in specifications as regular properties, not tags.

So we disable the extension C# language properties and then move them to tagged value in stereotype of C# profile and DSL properties. The DSL properties will conform to tagged value of stereotype.

For an old project containing old C# language properties, after performing open or import, they will be translated to the appropriate stereotype and tagged value.

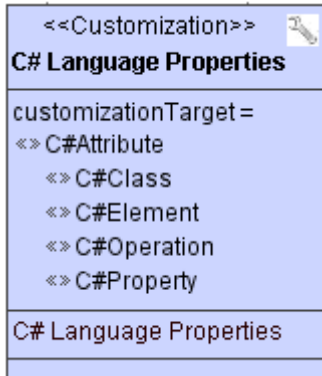


Figure 140 -- Customization Class

All DSL specific custom rules should be stored as tag values in Classes, marked with <<Customization>> stereotype.

This stereotype contains special tags that should be interpreted by “MD Customization Engine” in special ways, to enable many possible customizations in MD GUI and behavior.

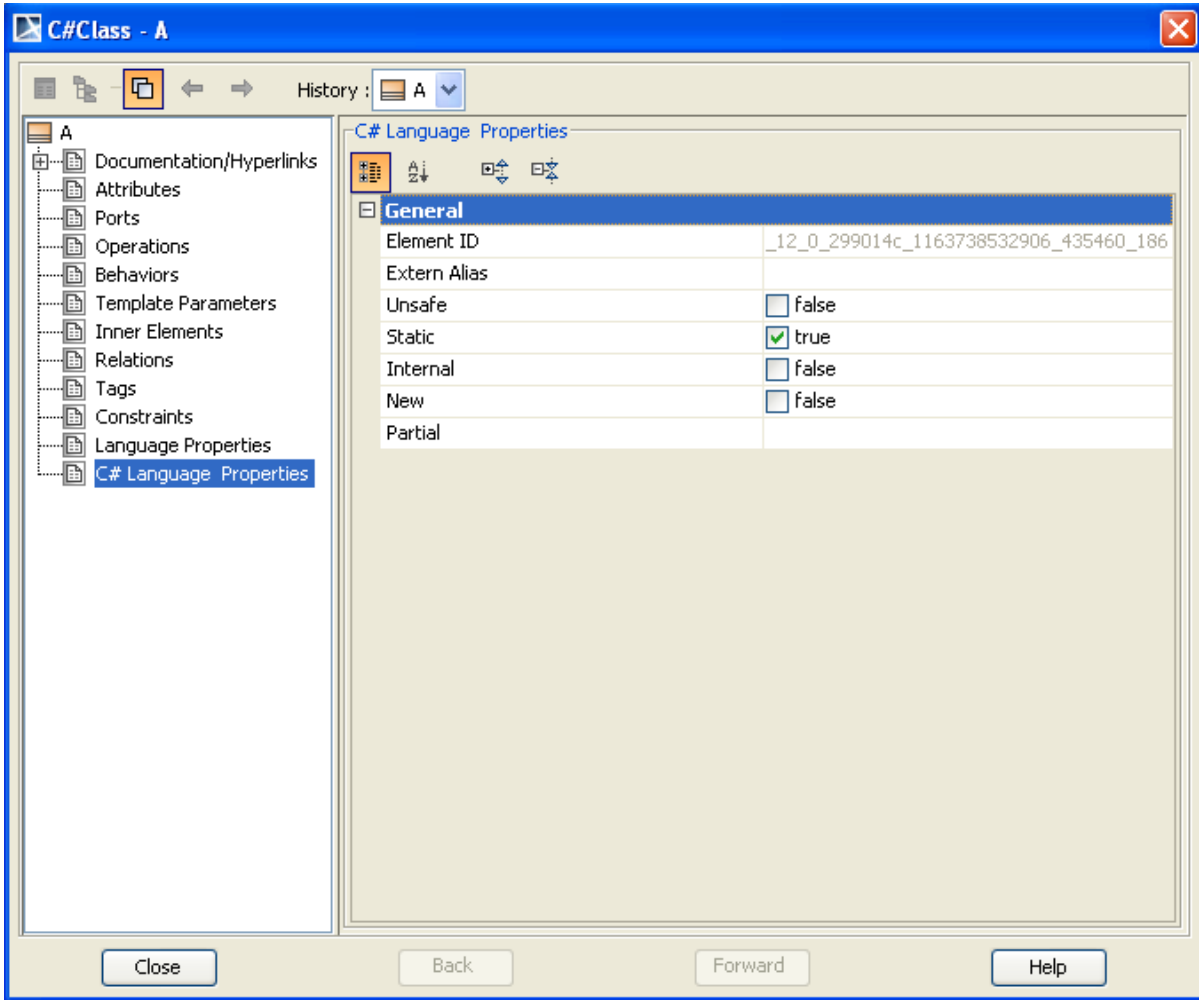
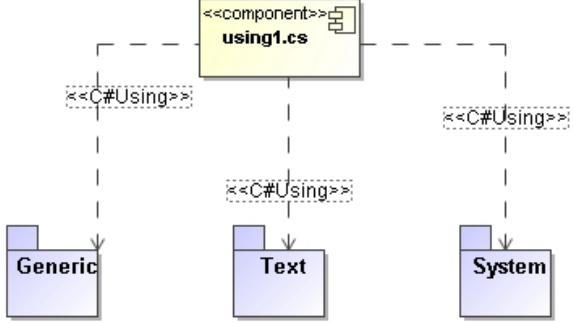


Figure 141 -- DSL Properties

## Using Directive Mapping

In MagicDraw version 12.1, the C# Using directive is mapped to the model as usage dependency with <<C#Using>> stereotype.

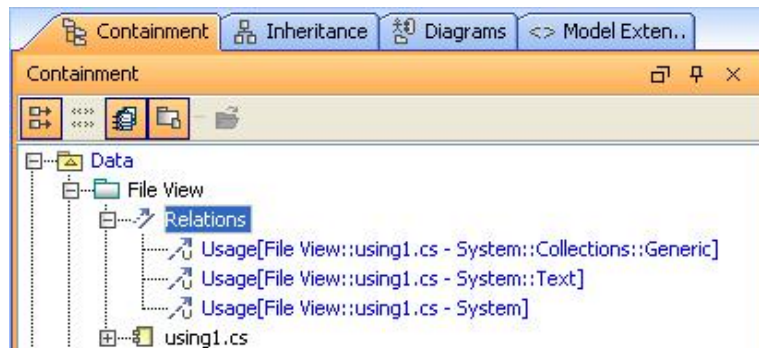
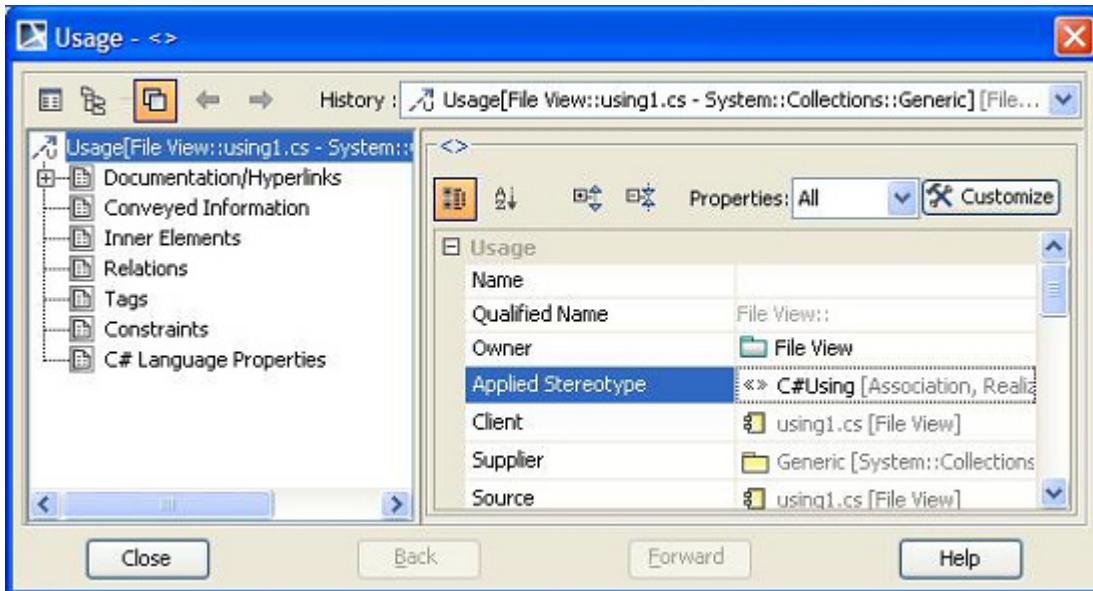
The following example shows the mapping of C# Using Namespace. The usage dependency for C# Using namespace declaration that is not in the namespace will be created under File View component.

Code	MD-UML
<pre>using System; using System.Collections.Generic; using System.Text; namespace n0 { class using1 {} }</pre>	

Open usage dependency specification

apply <<C#Using>> stereotype.

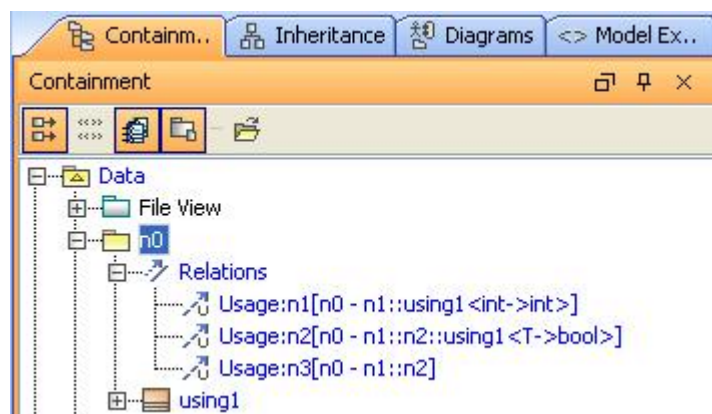
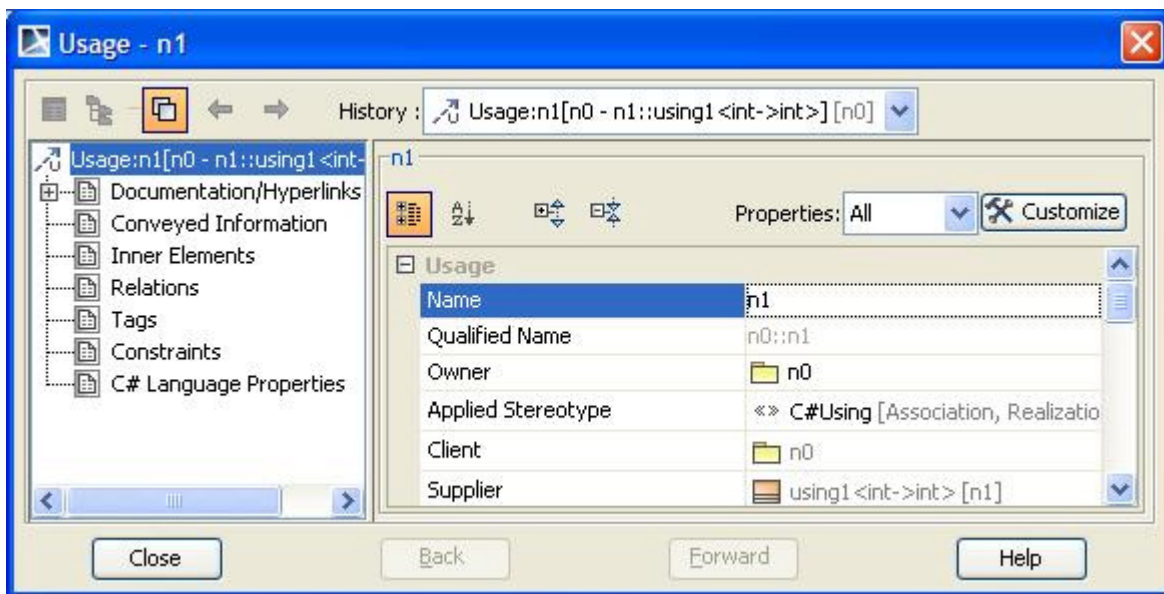
For using namespace, leave the name empty.




The following example shows the mapping of C# Using alias. The usage dependency for C# Using alias declared in the namespace will be created under that namespace.

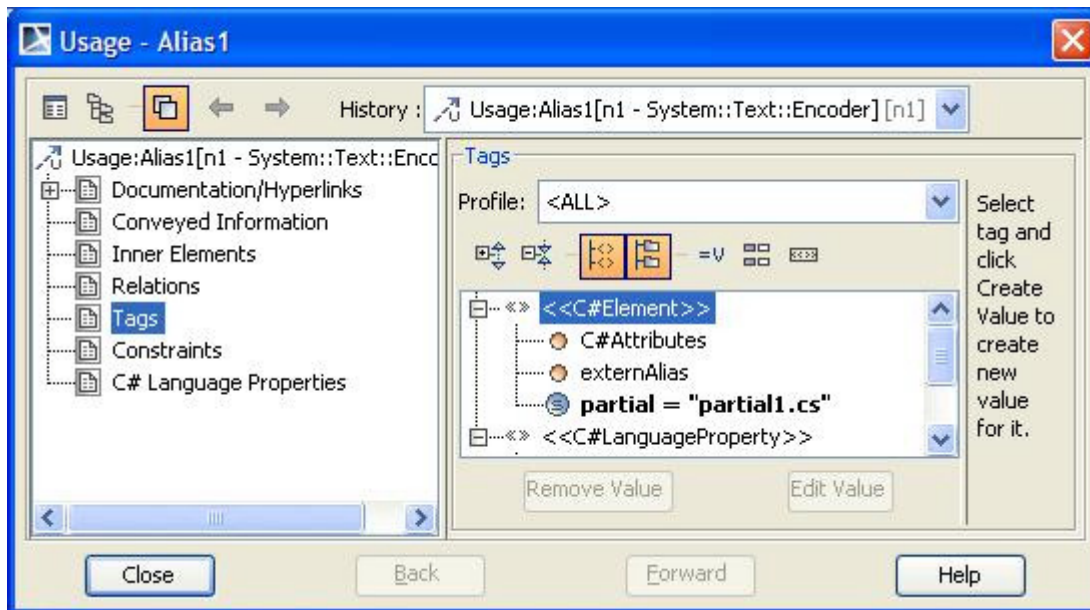
Code	MD-UML
<pre> namespace n0{     using n1 = n1.using1&lt;int&gt;;     using n2 = n1.n2.using1&lt;bool&gt;;     using n3 = n1.n2;     class using1 : n2 { n1 a;     n2 b; n3.using1&lt;float&gt; c;     }} namespace n1{     namespace n2 {         class using1&lt;T&gt; { }     }     class using1&lt;T&gt; { } } </pre>	


Open usage dependency specification  
 Apply <<C#Using>> stereotype.  
 For using alias, enter the alias name.



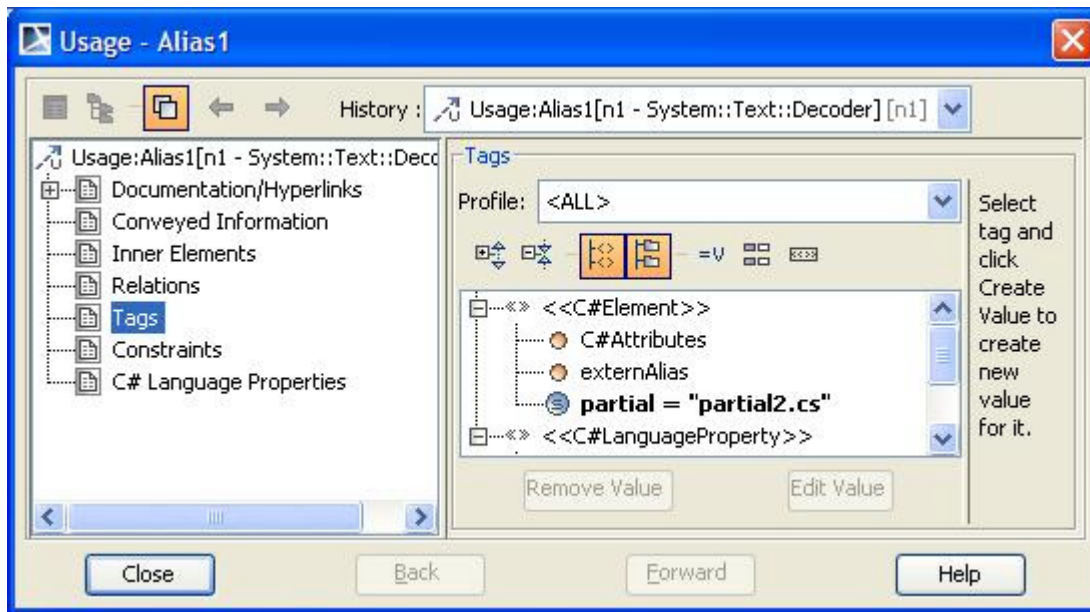
For mapping C# Using directive in the C# Partial feature, we have to add file component name into Partial Tag value of usage link specification.

Code	MD-UML
<pre>//partial1.cs namespace n1 {     using LL = System.Text.Encoder;     partial class using1     {         LL a;     } }</pre>	



<pre>//partial2.cs namespace n1 {     using LL = System.Text.Decoder;     partial class using1     {         LL b;     } }</pre>	
--	--





## Constraints

### Mapping Constraints

#### No Mapping Cases

This section will show some cases which have no mapping in C# reverse engineering.

### UML Constraints

#### Multiple Generic Class

In C# syntax, we can declare classes with the same name but different generic type parameters in the same package. For example;

```
namespace N {  
    class A { .. }  
    class A<T> { .. }  
    class A<T, U> { .. }  
}
```

The user cannot create the model for these three classes in MagicDraw manually. They can only be created in C# reverse engineering.



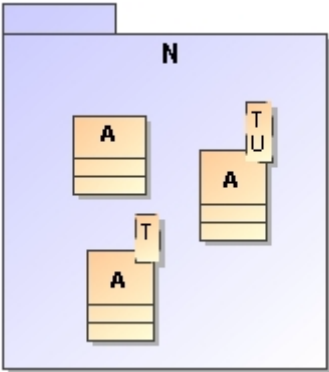


Figure 142 -- Generic Classes

How to manually create class A in the model, and Figure 12 shows the error message when trying to create class A.

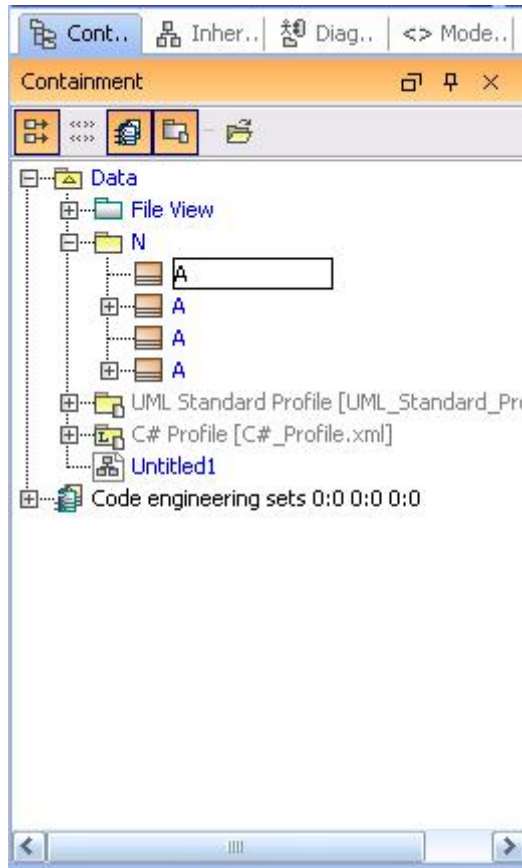


Figure 143 -- Creating Class A in The Model

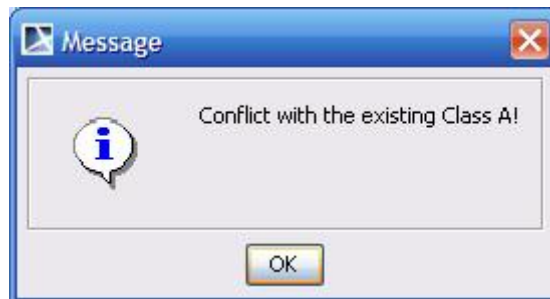


Figure 144 -- Conflict Message Dialog

## Code Generation for Partial Class

Generate the code without the round trip feature. Separated partial classes and all its child elements will be generated into only one class code and one class file as the example below.

Code used for reverse engineer	Code after generation
<pre>//Case #1 //The partial class is written into one class file.  public partial class PartialA{     int a;     string actionA()     {     } } public partial class PartialA{     int b;     string actionB()     {     } }</pre>	<pre>public partial class PartialA{     int a;      int b;     string actionA( )     {return ;}     string actionB( )     {return ;} }</pre>
<pre>//Case #2 //The partial class is written into separate class file.  //PartialA1.cs public partial class PartialA{     int a;     string actionA()     {     } }  //PartialA2.cs public partial class PartialA{     int b;     string actionB()     {     } }</pre>	<pre>public partial class PartialA{     int a;      int b;     string actionA( )     {return ;}     string actionB( )     {return ;} }</pre>

Code used for reverse engineer	Code after generation
<pre>//Case #3 //The partial class with inner class  public partial class PartialA{     public class B     {         int b;     } } public partial class PartialA{     int a;     string actionB()     {     }     public class C     {         int c;     } } }</pre>	<pre>public partial class PartialA{     int a;     string actionB( )     {return ;}     public class B     {int b;}     public class C     {int c;} }</pre>

## Translation Constraints

### Property Translation

As you may know that after version 12.1 MD C# will translate C#Property from operation to attribute. After the translation is finished, the message window will show the following message:

Orphaned proxy found in module "C#\_Profile.xml". Use search functionality with option "Orphaned proxies only" to locate them.

This message occurred because there is no Operation as a metaclass of C#Property (changing of profile), but this will not affect the functionality of reverse engineering and translation.

After the project has been saved and reopened, the message will not be appear any more.

# CORBA IDL MAPPING TO UML

**NOTE:** This functionality is available in Enterprise edition only.

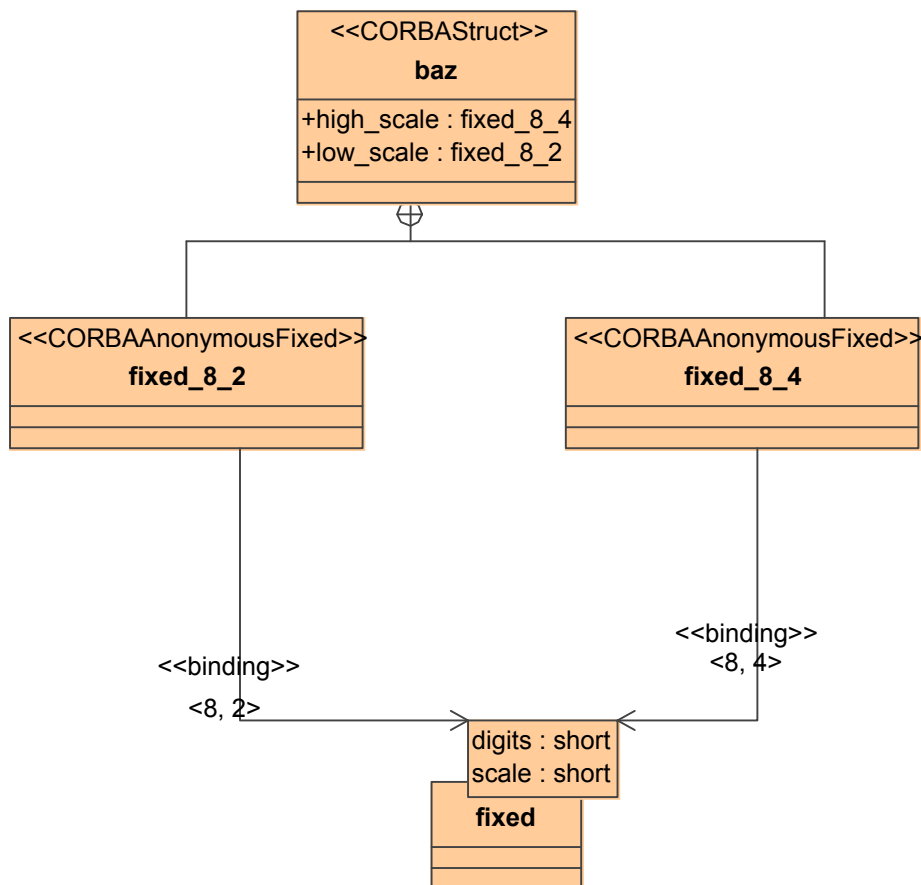
CORBA IDL mapping to UML is based on UML™ Profile for CORBA™ Specification, Version 1.0, April 2002. [http://www.omg.org/technology/documents/formal/profile\\_corba.htm](http://www.omg.org/technology/documents/formal/profile_corba.htm).

Differences between UML Profile for CORBA specification and mapping in MagicDraw is listed below.








- Constraints defined in UML Profile for CORBA specification are not checked in MagicDraw.
- Stereotype CORBAAnonymousFixed is introduced. It is used to represent anonymous fixed types. Fixed types without names are mapped to inner classes with stereotype CORBAAnonymousFixed. These classes are bound to CORBA::fixed classes.IDL Code:

```
struct baz
{
  fixed <8, 4> high_scale;
  fixed <8, 2> low_scale;
};
```

This code is mapped to the following diagram:



MagicDraw presents a CORBA IDL diagram, which simplifies the creation of standard CORBA IDL elements. The following elements are available in the CORBA IDL diagram:

Button	Shortcut key	Model Element
	M	<b>CORBA IDL Module</b>
	I	<b>CORBA IDL Interface</b>
	V	<b>CORBA IDL Value</b>
	SHIFT+P	<b>Class by Pattern</b>
	G	<b>Generalization</b>
		<b>Truncatable Generalization</b>
		<b>Value Support Generalization</b>
		<b>CORBA IDL Association</b>
	I	<b>Interface</b>

## **NEW!** CORBA Interface Implementation

You can select either the UML Interface or the UML Class as a base element for CORBA Interface. The default element as a base element for CORBA Interfaces is as follows:

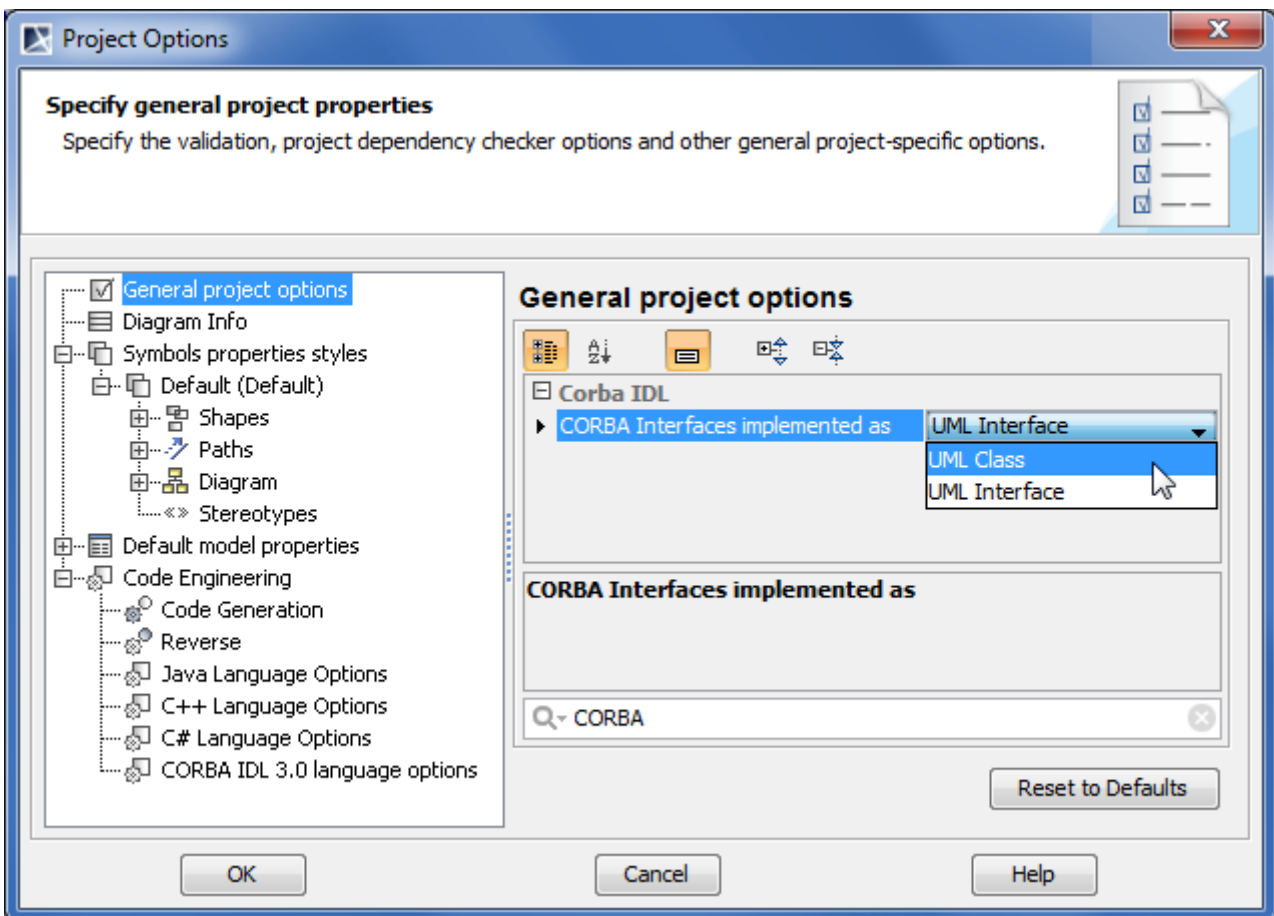
- UML Class. A class as a base element for CORBA Interfaces is in projects created on earlier than MagicDraw version 17.0.1. The CORBA Interfaces are modeled as classes in these projects.

- UML Interface. An interface as a base element for CORBA Interfaces is in projects created on MagicDraw version 17.0.1 and later. The CORBA Interfaces are modeled as interfaces in these projects.

**IMPORTANT!** The generalization relationship can be used only between CORBA Interfaces based on the same element. That is, you can use the generalization relationship between CORBA Interfaces based on a class or between CORBA Interfaces based on an interface but not between CORBA Interface based on a class and CORBA Interface based on an interface.

## To set a base element for CORBA Interfaces

1. On the **Options** menu, click **Project**. The **Project Options** dialog opens.
2. In the Tab tree, select **General project options**.
3. In the **General project options** list, click the **CORBA Interfaces implemented as** value cell. The list of available values opens.
4. Do one of the following:
  - Select **UML Class**, to set a class as the base element for the CORBA Interface.
  - Select **UML Interface**, to set an interface as the base element for the CORBA Interface.
5. Click **OK** when you are done.



## To change a base element for CORBA Interfaces

1. Select the CORBA Interface element in the Model Browser or its symbol on the diagram pane.

2. On the shortcut menu of the selected CORBA Interface, point to **Refactor > Convert To**, and then do the following:
  - Click **Class**, if an interface is the current base element for CORBA Interface.
  - Click **Interface**, if a class is the current base element for CORBA Interface.

**IMPORTANT!** Convert selected CORBA Interface only to the class or interface element. Conversion to other elements changes the CORBA Interface element to the selected element but not the element CORBA Interface is based on.

For more information about refactoring, see “Refactoring” in “[MagicDraw UserManual.pdf](#)”.



# EJB 2.0 - UML

In MagicDraw, EJB-UML mapping is based on UML Profiles for EJB. The UML Profiles for EJB is a mapping from Unified Modeling Language (UML) to the Enterprise JavaBeans (EJB) architecture. The mapping specifies the standard representation for elements of the EJB architecture in UML models.

UML Profiles for EJB consist of Java, EJB Design, and EJB Deployment profiles. The Java Profile defines the subset of Java constructs that are needed to support the EJB Profiles. EJB Design Profile defines how to model EJB applications using the interfaces and the implementation classes of an enterprise bean. The EJB Deployment Profile defines mapping of EJB components contained in EJB-JAR archives and deployment on Application Servers to UML.

## Java Profile

This section describes stereotypes and tagged values introduced in the Java Profile

### Stereotypes.

Stereotype	Applies To	Definition
«JAR»	Artifact, Component	Specializes the standard UML Stereotype «file». Indicates that the Artifact or Component represents a JAR.
«JARInclude»	Dependency	Indicates that the supplier of the Dependency, a JAR , Java .class file, or other file, is included “by value” ( <i>i.e.</i> , copied into) the JAR that is the client of the Dependency
«JARReference»	Usage	Indicates that the supplier is a model element that is not contained in the archive but is referenced by it in the META-INF/MANIFEST.MF file
«JavaSourceFile»	«file»	Specializes the standard UML Stereotype «file». Indicates that the Artifact describes a Java Source File. <b>NOTE</b> The compiled “*.class” file is not modeled by this profile.

### Tagged Values

Tagged Values of the Java Profile are included in current implementation of MagicDraw but are not used in EJB 2.0 code engineering.

Tagged Value	Applies To	Definition
JavaStrictfp	Class	A Boolean value indicating whether or not the Java Class is FP-strict
JavaStatic	Class	A Boolean value indicating whether or not the Java Class is static.
JavaVolatile	Attribute or AssociationEnd	A Boolean value indicating whether or not the Java Field is volatile.
JavaDimensions	Attribute, AssociationEnd or Parameter	An Integer indicating the number of array dimensions declared by the Java Field or Parameter.

Tagged Value	Applies To	Definition
<b>JavaCollection</b>	Attribute or AssociationEnd	A String containing the name of the Java Collection Type used to implement an Attribute or Association End with complex multiplicity.
<b>JavaNative</b>	Operation or Method	A Boolean value indicating whether or not the Operation or Method is native.
<b>JavaThrows</b>	Operation or Method	A comma-delimited list of names of Java Exception Classes.
<b>JavaFinal</b>	Parameter	A Boolean value indicating whether or not the Parameter is final.

## EJB Design Profile

This section describes stereotypes and tagged values introduced in the EJB Design Profile.

## Stereotypes

Stereotype	Applies To	Definition
«EJBBusiness»	Operation	Indicates that the Operation represents an <i>instance-level</i> business method, a method that supports the “business logic” of the EJB. Contrast with <i>class-level</i> business methods on home interfaces, which are stereotyped «EJBHome»
«EJBCmpField»	Attribute, AssociationEnd	Indicates that the Attribute or Association End represents a container-managed field for an EJB Entity Bean with container-managed persistence
«EJBCmrMethod»	Dependency	A UML Dependency with the «EJBRelationshipRole» Association End as the client and the abstract get/set Methods of the relationship as the suppliers. This connects the EJB Relationship to the pair of container managed relationship methods, and allows the decoupling of the role name to the name of the CMR fields. (EJB 2.0 only)
«EJBCreate»	Operation	Indicates that the Operation represents an EJB Create Method.
«EJBEnterpriseBean»	Class	Specializes the standard UML Stereotype «implementationClass». An abstract Class that represents an EJB Enterprise Bean. The stereotype is applied to the implementation class for the EJB.
«EJBEntityBean»	Class	Indicates that the Class represents an EJB Entity Bean. Specializes «EJBEnterpriseBean».
«EJBFinder»	Operation	Indicates that the Operation represents an EJB Finder Method.
«EJBHome»	Operation	Indicates that the Operation represents an EJB Home Method, <i>either</i> local or remote, which is a <i>class-level</i> “business” method, as opposed to a “create”, “finder”, <i>etc.</i> method. Contrast with «EJBBusiness».
«EJBLocalMethod»	Operation	Indicates that the Operation represents a method that is exposed on <i>either</i> the local or local-home interface. The former case is assumed if the EJB Business stereotype is also present. (EJB 2.0 only)
«EJBLocalReference»	Dependency	A stereotyped Dependency representing an EJB Local Reference, where the client is an EJB-JAR and the supplier is an EJB Enterprise Bean. (EJB 2.0 only)
«EJBMessageDrivenBean»	Class	Indicates that the Class represents an EJB Message Driven Bean. Specializes «EJBEnterpriseBean». (EJB 2.0 only)
«EJBPrimaryKey»	Usage	Indicates that the supplier of the Usage represents the EJB Primary Key Class for the EJB Enterprise Bean represented by the client.
«EJBPrimaryKeyField»	Attribute, AssociationEnd	Specializes «EJBCmpField». Indicates that the Attribute or Association End is the primary key field for an EJB Entity Bean with container-managed persistence.

Stereotype	Applies To	Definition
«EJBRealizeHome»	Abstraction	Indicates that the supplier of the Abstraction represents an EJB Remote Home Interface for the EJB Enterprise Bean Class represented by the client.
«EJBRealizeLocal»	Abstraction	Indicates that the supplier of the Abstraction represents an EJB Local Interface for the EJB Enterprise Bean Class represented by the client. (EJB 2.0 only)
«EJBRealizeLocalHome»	Abstraction	Indicates that the supplier of the Abstraction represents an EJB Local Home Interface for the EJB Enterprise Bean Class represented by the client. (EJB 2.0 only)
«EJBRealizeRemote»	Abstraction	Indicates that the supplier of the Abstraction represents an EJB Remote Interface for the EJB Enterprise Bean Class represented by the client.
«EJBReference»	Dependency	A stereotyped Dependency representing an EJB (Remote) Reference, where the client is an EJB-JAR and the supplier is an EJB Enterprise Bean.
«EJBRelationship»	Association	Indicates that the EJB Entity Bean at the supplier Association End represents a container-managed relationship field for the client EJB Entity Bean. (EJB 2.0)
«EJBRelationshipRole»	AssociationEnd	Indicates the Association End of an «EJBRelationship».
«EJBRemoteMethod»	Operation	Indicates that the Operation represents a method that is exposed on <i>either</i> the remote or remote-home interface. The former case is assumed if the EJB Business stereotype is also present.
«EJBRoleNameRef»	Actor	The name of security role <i>reference</i> used programmatically in a bean's source code and mapped to an EJB Role Name in the Deployment Model.
«EJBRoleLink»	Dependency	Indicates a Dependency between a client EJB Role Name Reference (Design Model) and a supplier EJB Role Name (Deployment Model).
«EJBSessionBean»	Class	Indicates that the Class represents an EJB Session Bean. Specializes «EJBEnterpriseBean».
«EJBSelect»	Operation	An Operation that is a select method in an EJB Entity Bean. (EJB 2.0 only)
«EJBSecurityRoleRef»	Association	Indicates a Dependency between an EJB client, and an EJB Role Name Reference supplier.

## Tagged Values

Tagged Value	Applies To	Definition
<b>EJBAbstractSchemaName</b>	Class «EJBEntityBean»	A String representing the abstract name used for a schema associated with a CMP 2.0 entity bean.
<b>EJBAcknowledgeMode</b>	Class «EJBMessageDrivenBean»	An enumeration with values Auto Acknowledge or Dups OK Acknowledge indicating the type of message acknowledgment used for the onMessage message of a message-driven bean that uses bean-managed transaction demarcation.
<b>EJBComponentInterface</b>	Dependency «EJBReference» or «EJBLocalReference»	The name of the EJB Enterprise Bean's "Component" Interface (local or remote).
<b>EJBCmpVersion</b>	Class «EJBEntityBean»	An enumeration with values 1.x or 2.x. Indicates the type of CMP used by the EJB Entity Bean
<b>EJBCmrFieldType</b>	AssociationEnd «EJBRelationshipRole»	Type expression property of the supplier UML Association End «EJBRelationshipRole».
<b>EJBDisplayName</b>	Class «EJBEnterpriseBean»	A String with the name for the EJB to be displayed by tools
<b>EJBEnvEntries</b>	Class «EJBEnterpriseBean»	A string of XML tags, designating the environment entries used by the EJB Enterprise Bean.
<b>EJBHomeInterface</b>	Dependency «EJBReference» or «EJBLocalReference»	The name of the EJB Enterprise Bean's Home Interface (local or remote).
<b>EJBLink</b>	Dependency «EJBReference» or «EJBLocalReference»	The name of the referenced EJB Enterprise Bean, if it is in the same Archive or another archive in the same J2EE Application Unit, or a path to the bean.
<b>EJBMessageDrivenDestination</b>	Class «EJBMessageDrivenBean»	An XML tag, designating the type of destination and the durability used by the EJB Message Driven Bean.
<b>EJBMessageSelector</b>	Class «EJBMessageDrivenBean»	A String specifying the JMS message selector to be used in determining which messages a message-driven bean is to receive.

Tagged Value	Applies To	Definition
<b>EJBNameInJAR</b>	Class «EJBEnterpriseBean»	The name used for the EJB Enterprise Bean in the EJB-JAR. Defaults to the name of the EJB Remote Interface.
<b>EJBPersistenceType</b>	Class «EJBEntityBean»	An enumeration with values Bean or Container. Indicates whether the persistence of the EJB Entity Bean is managed by the EJB Entity Bean or by its container, respectively.
<b>EJBQueryString</b>	Operation «EJBFinder» or «EJBSelect»	The EJB QL statement corresponding to the method. It is ignored for BMP Entity Beans
<b>EJBReentrant</b>	Class «EJBEntityBean»	A Boolean value indicating whether or not the EJB Entity Bean can be called re-entrantly.
<b>EJBRefName</b>	Dependency «EJBReference» or «EJBLocalReference»	The name of the EJB Enterprise Bean referenced.
<b>EJBRefType</b>	Dependency «EJBReference» or «EJBLocalReference»	The type of referenced EJB Enterprise Bean, one of the strings Entity or Session.
<b>EJBRelationshipRoleDescription</b>	AssociationEnd «EJBRelationshipRole»	A String specifying a description an EJB relationship role.
<b>EJBRelationshipRoleName</b>	AssociationEnd «EJBRelationshipRole»	The name of an EJB relationship role.
<b>EJBResources</b>	Class «EJBEnterpriseBean»	A string of XML tags designating the resource factories used by the EJB Enterprise Bean.
<b>EJBResourcesEnv</b>	Class «EJBEnterpriseBean»	A string of XML tags designating the environment entries used by the EJB Enterprise Bean.
<b>EJBResultTypeMapping</b>	Operation «EJBFinder» or «EJBSelect»	The allowed values are the literal strings Local or Remote. If the optional corresponding <result-type-mapping> tag is not to be used in the deployment descriptor, then the value is empty.
<b>EJBSessionType</b>	«EJBSessionBean»	Stateful or Stateless. Indicates whether or not the EJB Session Bean maintains state.

---

Tagged Value	Applies To	Definition
<b>EJBTransType</b>	Class «EJBSessionBean» or Class «EJBMessageDrivenBean»	An enumeration with values Bean or Container. Indicates whether the transactions of the EJB Session Bean or EJB Message Driven Bean are managed by the bean or by its container, respectively.
<b>EJBVersion</b>	Class «EJBEnterpriseBean»	An enumeration with values 1.1 or 2.0. Indicates the EJB version of the EJB Enterprise Bean

## EJB Deployment Profile

### Stereotypes

Stereotype	Applies To	Definition
«EJB-JAR»	Artifact, Component	Specializes the Stereotype «JAR». Indicates that the Artifact represents an EJB JAR.
«EJBClientJAR»	Usage	Indicates that the client of the Usage represents an ejb-client-jar for the EJB-JAR represented by the supplier of the Usage.
«EJBContainerTransaction»	Actor	Indicates the type of transaction used for associated Methods. It is combined with a transaction type stereotype, such as «EJBRequired». (See the following Table.) Container transactions are specified using one or more dependencies, stereotyped «EJBMethod», from an EJB Enterprise Bean client to an EJB Container Transaction Actor supplier
«EJBEnterpriseBeanDeployment»	Component	Indicates that the Component represents an EJB Enterprise Bean. It resides in an EJB-JAR. It has tagged values for overriding default settings in the corresponding EJB Enterprise Bean Class. For example, these tagged values can be used to create a new EJB by specifying different EJB Home and EJB Remote interfaces for a shared EJB Enterprise Bean Class.
«EJBEntityBeanDeployment»	Component	Specializes the standard Stereotype «EJBEnterpriseBean». Indicates that the Component represents an EJB Entity Bean.
«EJBExcludeList»	Actor	Indicates an Actor that represents an Exclude List. An exclude list is specified using one or more dependencies, stereotyped «EJBMethod», from an EJB Enterprise Bean client to an EJB Exclude List Actor supplier
«EJBMessageDrivenBeanDeployment»	Component	Specializes the standard Stereotype «EJBEnterpriseBean». Indicates that the Component represents an EJB Message Driven Bean.
«EJBMethod»	Dependency	Indicates a Dependency between an EJB Component client and an EJB Method Permission (or EJB Container Transaction or EJBExcludeList) supplier that completes a Method Permission (or Container Transaction or Exclude List) specification. It contains a method specification, either “style 1, 2, or 3”, according to the EJB 2.0 specification.
«EJBMethodPermission»	Actor	Indicates an Actor that represents a Method Permission. It contains a tag value indicating whether or not the method is “unchecked”. One or more method permissions are specified using one or more dependencies, stereotyped «EJBMethod», from an EJB Enterprise Bean to an EJB Method Permission Actor



Stereotype	Applies To	Definition
«EJBOverrideHome»	«reside»	Indicates a relationship that, for the client EJB Enterprise Bean Component, the supplier UML Interface overrides the Home Interface that is defined by the EJB Enterprise Bean. This is a technique for constructing new EJBs while reusing EJB Enterprise Bean Classes.
«EJBOverrideLocal»	«reside»	Indicates a relationship that, for the client EJB Enterprise Bean Component, the supplier UML Interface overrides the Local Interface that is defined by the EJB Enterprise Bean. This is a technique for constructing new EJBs while reusing EJB Enterprise Bean Classes.
«EJBOverrideLocalHome»	«reside»	Indicates a relationship that, for the client EJB Enterprise Bean Component, the supplier UML Interface overrides the Local Home Interface that is defined by the EJB Enterprise Bean. This is a technique for constructing new EJBs while reusing EJB Enterprise Bean Classes.
«EJBOverridePrimaryKey»	«reside»	Indicates a relationship that, for the client EJB Enterprise Bean Component, the supplier Java Class overrides the EJB Primary Key Class that is defined by the EJB Enterprise Bean. This is a technique for constructing new EJBs while reusing EJB Enterprise Bean Classes.
«EJBOverrideRemote»	«reside»	Indicates a relationship that, for the client EJB Enterprise Bean Component, the supplier UML Interface overrides the Remote Interface that is defined by the EJB Enterprise Bean. This is a technique for constructing new EJBs while reusing EJB Enterprise Bean Classes.
«EJBRoleName»	Actor	Indicates the name of a security role used in the definitions of Method Permissions, <i>etc.</i>
«EJBRunAsIdentity»	Dependency	Indicates a Dependency between an EJB Role Name supplier and an EJB Component client that completes a Run As specification, which names a role under which methods of the EJB are to be executed.
«EJBSecurityRole»	Association	Indicates a Dependency between an EJB JAR client and an EJB Role Name supplier. It defines the role as part of the archive.
«EJBSessionBeanDeployment»	Component	Specializes the standard Stereotype «EJBEnterpriseBeanDeployment». Indicates that the Component represents an EJB Session Bean.

The «EJBContainerTransaction» Actor model elements always receive a second stereotype, which identifies the *type* of transaction.

Transaction Type	Description
«EJBMandatory»	Transactions are mandatory for this method
«EJBNever»	Transactions are never used for this method
«EJBNotSupported»	Transactions aren't supported for this method
«EJBRequired»	Transactions are required for this method

Transaction Type	Description
«EJBRequiresNew»	A new transaction is required for this method
«EJBSupports»	Transactions are supported for this method

## Tagged Values

Tagged Value	Applies To	Definition
<b>EJBDisplayName</b>	Artifact «EJB-JAR»	A String with the name for the Archive to be displayed by tools
<b>EJBMethodDescriptor</b>	Dependency «EJBMethod»	A String with the subset of XML required to specify a complete deployment descriptor <method> tag for a method or set of methods, using “Style 1, 2, or 3” syntax, as defined by the EJB 2.0 Specification.
<b>EJBUnchecked</b>	Actor «EJBMethodPermission»	A Boolean that when “true”, indicates that permission to invoke the specified methods should not be checked before invocation. Default is “false”
<b>EJBUseCallerIdentity</b>	Component «EJBEnterpriseBean»	A Boolean indicating whether or not to use the caller’s identity when invoking the bean methods.

## Using MagicDraw EJB 2.0

### Reverse engineering

1. Create EJB 2.0 code engineering set.

**NOTE** If the project does not contain profiles required for EJB code engineering, you will be asked whether you would like to import the EJB 2.0 template. It is recommended to select **Yes**.

2. Add java files and EJB deployment descriptor file that are part of concrete EJB. EJB deployment descriptor file ejb-jar.xml must be in META-INF directory.
3. Reverse those files.

After the reverse, you will get the following result:

Model elements representing the code will be created. Stereotypes and tagged values representing EJB tags will be applied to appropriate model elements.

### Code generation

We will create a simple example and we will go step by step from the beginning to the generated code. As an example we will take a reservation system for a theater ticket reservation. It will consist of single session enterprise bean ReservationBean. You may find this example in the <MagicDraw installation directory>, samples folder.

1. Create a new MagicDraw project from EJB 2.0 template.
2. Create a new class diagram.

3. Create an interface Reservation that extends javax::ejb:EJBObject interface in the class diagram.
4. Create operation bookSeats (playName : java::lang::String, userName : java::lang::String, quantity : java::lang::String) in the bean's class. The operation should throw java::rmi::RemoteException.  
To specify which exceptions the method throws, open the **Operation Specification** dialog box, click the **Language Properties** button and in the **CG Properties Editor** dialog box, **Java** tab specify the **Throws Exceptions** property (java.rmi.RemoteException).
5. Create a class with name ReservationBean. The class will be an enterprise session bean.
6. Make the class realizes SessionBean interface from javax.ejb package.
7. Open the ReservationBean's specification and add EJBSessionBean stereotype to the class and EJBNameInJAR tagged value - ReservationEJB.
8. Create operation bookSeats( playName : java::lang::String, userName : java::lang::String, quantity : java::lang::String ) in the bean's class. You can copy the operation from the Reservation interface (select the operation in the Reservation interface press Ctrl button and drag the operation to bean's class).
9. Create an interface ReservationHome that extends javax::ejb::EJBHome interface and has operation public Reservation create() throws CreateException, RemoteException.
10. Create abstraction links from the bean's class to the Reservation and Reservation Home interfaces. On the abstraction between ReservationBean and Reservation add stereotype EJBRealizeHome. On the abstraction between ReservationBean and ReservationHome interface add stereotype EJBRealizeRemote.
11. Create a new EJB 2.0 code engineering set.
12. In the containment tree select ReservationBean, Reservation and ReservationHome and drag them to the set.
13. In the ComponentView select ReservationBean.java component, open specification for it and add stereotype EJBSessionBeanDeployment.
14. In the Component View package create new component ejb-jar.jar. Open specification and add stereotype EJB-JAR to it.
15. Create a new Implementation Diagram.
16. Drag ReservationBean.java and ejb-jar.jar components to the diagram.
17. Create dependency link from the ReservationBean.java component to ejb-jar.jar component and add stereotype implement to the dependency.
18. Drag ejb-jar.jar component from the in the Containment Tree from Component View to the EJB 2.0 code engineering set.
19. Generate the code.

# WSDL

**NOTE:** This functionality is available in Enterprise edition only.

Reference <http://www.w3.org/TR/2001/NOTE-wsdl-20010315>

WSDL is an XML format for describing network services as a set of endpoints operating on messages containing either document-oriented or procedure-oriented information. The operations and messages are described abstractly, and then bound to a concrete network protocol and message format to define an endpoint. Related concrete endpoints are combined into abstract endpoints (services).

MagicDraw UML supports WSDL code engineering: code generation, reverse and syntax checking. WSDL diagram is dedicated for WSDL modeling.

WSDL Profile and XML Schema Profile are used in WSDL code engineering.

WSDL Services are defined using six major elements:

- **types**, which provides data type definitions used to describe the messages exchanged.
- **message**, which represents an abstract definition of the data being transmitted. A message consists of logical parts, each of which is associated with a definition within some type system.
- **portType**, which is a set of abstract operations. Each operation refers to an input message and output messages.
- **binding**, which specifies concrete protocol and data format specifications for the operations and messages defined by a particular portType.
- **port**, which specifies an address for a binding, thus defining a single communication endpoint.
- **service**, which is used to aggregate a set of related ports.

## WSDL Mapping to UML elements

### Defined stereotypes

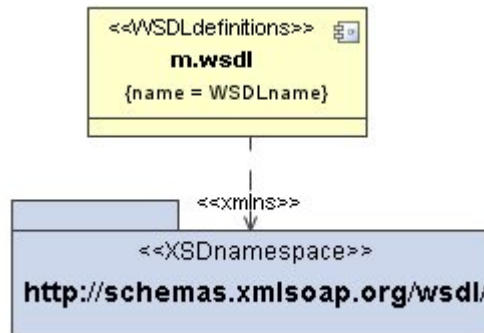
Element	Stereotype name	Applies to	Defined TagDefinitions	Details
Definition	WSDLdefinitions	Component	extension - string name - string targetNamespace - string	
Message	WSDLmessage	Class		
Port Type	WSDLporttype	interface		
Binding	WSDLbinding	Class	extension - string	
Port	WSDLport	Instance Specification	extension - string	
Service	WSDLservice	Component	extension - string	
Type	WSDLtypes	Component	extension - string	
	WSDLimport	ElementImport, PackageImport		
	xmlns	ElementImport, PackageImport		From the XML Schema Profile
	XSDnamespace	Package		From the XML Schema Profile
	WSDLresponse	Parameter	extension - string	
	WSDLoperation	Operation	extension - string	
	WSDLpart	Property	typing Attribute - string	
	WSDLfault	Parameter	extension - string	
	WSDLrequest	Parameter	extension - string	

### Definitions

A WSDL document is simply a set of definitions. There is a definitions element at the root, and definitions inside.

Example:

```
<definitions name="WSDLname" xmlns="http://schemas.xmlsoap.org/wsdl/">
```

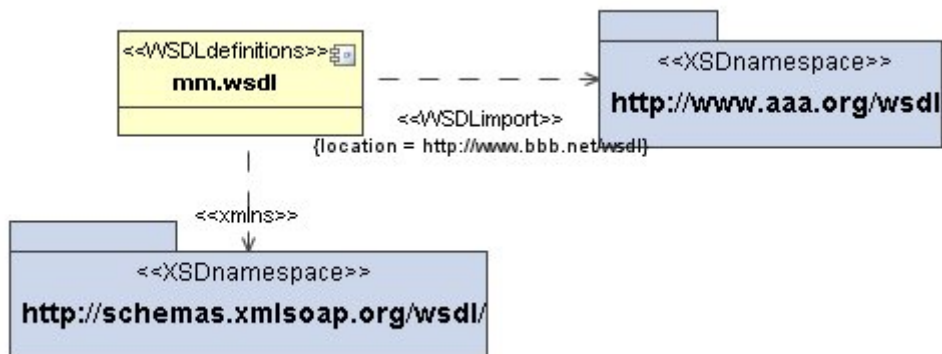
**Reversed UML model example:**

## Import, namespace

Example:

```
<definitions xmlns="http://schemas.xmlsoap.org/wsdl/">
  <import location="http://www.bbb.net/wsdl" namespace="http://www.aaa.org/
wsdl"/>
</definitions>
```

Reversed UML model example:



## Messages

**Messages** consist of one or more logical parts. Each part is associated with a type from some type system using a message-typing attribute. The set of message-typing attributes is extensible.

WSDL defines several such message-typing attributes for use with XSD:

- **element**. Refers to an XSD element using a QName.
- **type**. Refers to an XSD simpleType or complexType using a QName.

Other message-typing attributes may be defined as long as they use a namespace different from that of WSDL. Binding extensibility elements may also use message-typing attributes.

Example:

```
<definitions name="StockQuote"
targetNamespace="http://example.com/stockquote.wsdl"
xmlns:tns="http://example.com/stockquote.wsdl">
```

```

xmlns:xsd1="http://example.com/stockquote.xsd"
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns="http://schemas.xmlsoap.org/wsdl/"

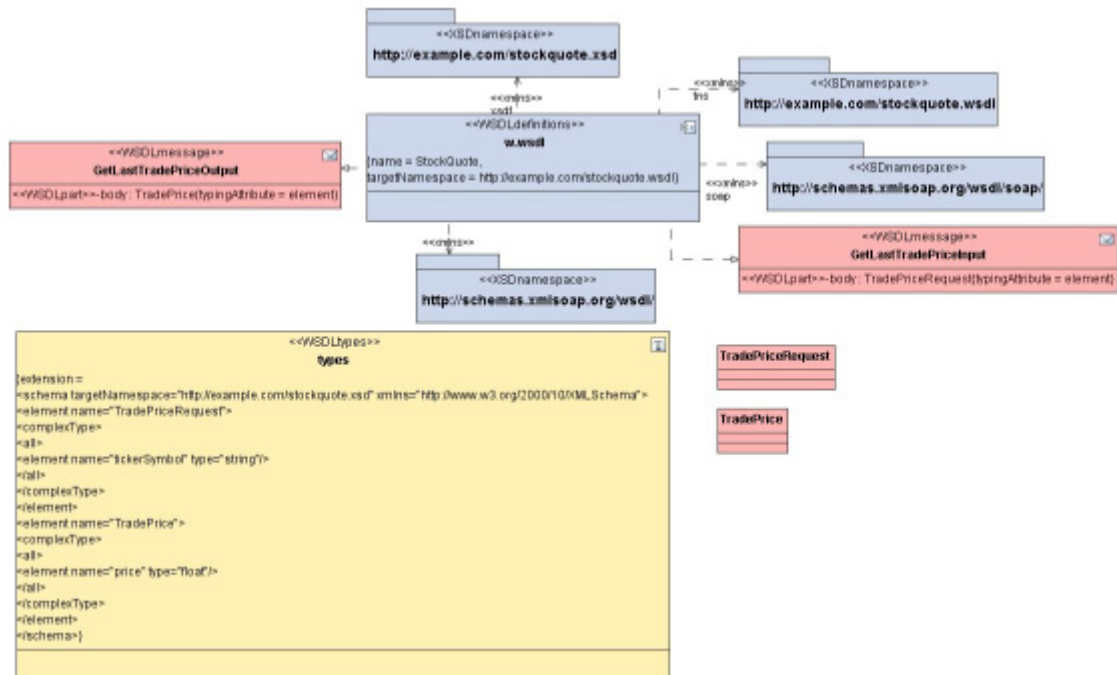
<types>
  <schema targetNamespace="http://example.com/stockquote.xsd"
    xmlns="http://www.w3.org/2000/10/XMLSchema">
    <element name="TradePriceRequest">
      <complexType>
        <all>
          <element name="tickerSymbol" type="string"/>
        </all>
      </complexType>
    </element>
    <element name="TradePrice">
      <complexType>
        <all>
          <element name="price" type="float"/>
        </all>
      </complexType>
    </element>
  </schema>
</types>

<message name="GetLastTradePriceInput">
  <part name="body" element="xsd1:TradePriceRequest"/>
</message>

<message name="GetLastTradePriceOutput">
  <part name="body" element="xsd1:TradePrice"/>
</message>
</definitions>

```

Reversed UML model example:



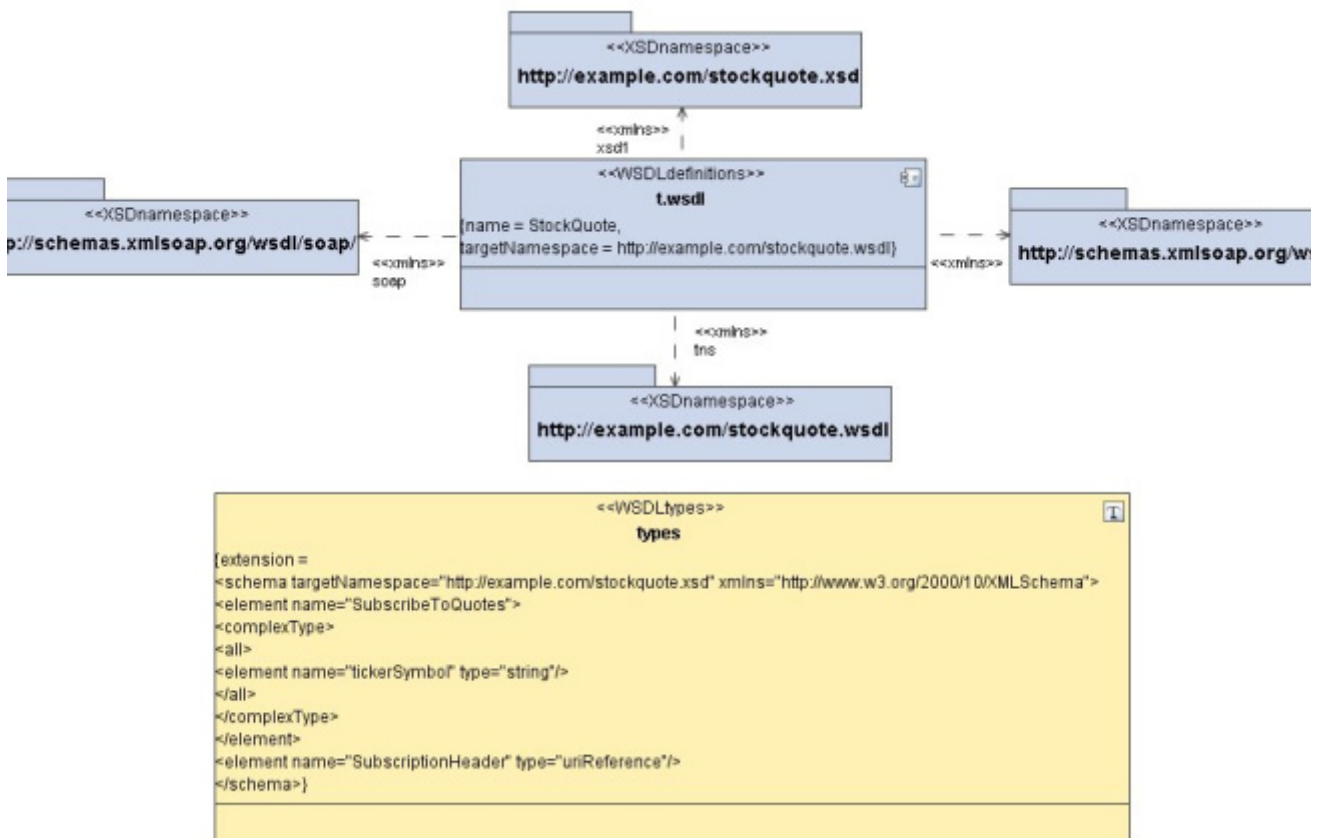
## Types

The **types** element encloses data type definitions that are relevant for the exchanged messages. For maximum interoperability and platform neutrality, WSDL prefers the use of XSD as the canonical type system, and treats it as the intrinsic type system.

Example:

```
<definitions name="StockQuote"
  targetNamespace="http://example.com/stockquote.wsdl"
  xmlns:tns="http://example.com/stockquote.wsdl"
  xmlns:xsd1="http://example.com/stockquote.xsd"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns="http://schemas.xmlsoap.org/wsdl/">
  <types>
    <schema targetNamespace="http://example.com/stockquote.xsd"
      xmlns="http://www.w3.org/2000/10/XMLSchema">
      <element name="SubscribeToQuotes">
        <complexType>
          <all>
            <element name="tickerSymbol" type="string"/>
          </all>
        </complexType>
      </element>
      <element name="SubscriptionHeader" type="uriReference"/>
    </schema>
  </types>
</definitions>
```

Reversed UML model example:





## Port types

A **port type** is a named set of abstract operations and the abstract messages involved.

The port type **name** attribute provides a unique name among all port types defined within in the enclosing WSDL document.

An operation is named via the **name** attribute.

WSDL has four transmission primitives that an endpoint can support:

- **One-way.** The endpoint receives a message.
- **Request-response.** The endpoint receives a message, and sends a correlated message.
- **Solicit-response.** The endpoint sends a message, and receives a correlated message.
- **Notification.** The endpoint sends a message.

WSDL refers to these primitives as **operations**. Although request/response or solicit/response can be modeled abstractly using two one-way messages, it is useful to model these as primitive operation types.

Example:

```
<definitions name="StockQuote"
  targetNamespace="http://example.com/stockquote.wsdl"
  xmlns:tns="http://example.com/stockquote.wsdl"
  xmlns:xsd="http://example.com/stockquote.xsd"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns="http://schemas.xmlsoap.org/wsdl/">

  <message name="SubscribeToQuotes">
    <part name="body" element="xsd:SubscribeToQuotes"/>
    <part name="subscribeheader" element="xsd:SubscriptionHeader"/>
  </message>

  <portType name="StockQuotePortType">
    <operation name="SubscribeToQuotes">
      <input message="tns:SubscribeToQuotes"/>
    </operation>
  </portType>

  <binding name="StockQuoteSoap" type="tns:StockQuotePortType">
    <soap:binding style="document" transport="http://example.com/smtp"/>
    <operation name="SubscribeToQuotes">
      <input message="tns:SubscribeToQuotes">
        <soap:body parts="body" use="literal"/>
        <soap:header message="tns:SubscribeToQuotes" part="subscribeheader"
use="literal"/>
      </input>
    </operation>
  </binding>

  <service name="StockQuoteService">
    <port name="StockQuotePort" binding="tns:StockQuoteSoap">
      <soap:address location="mailto:subscribe@example.com"/>
    </port>
  </service>

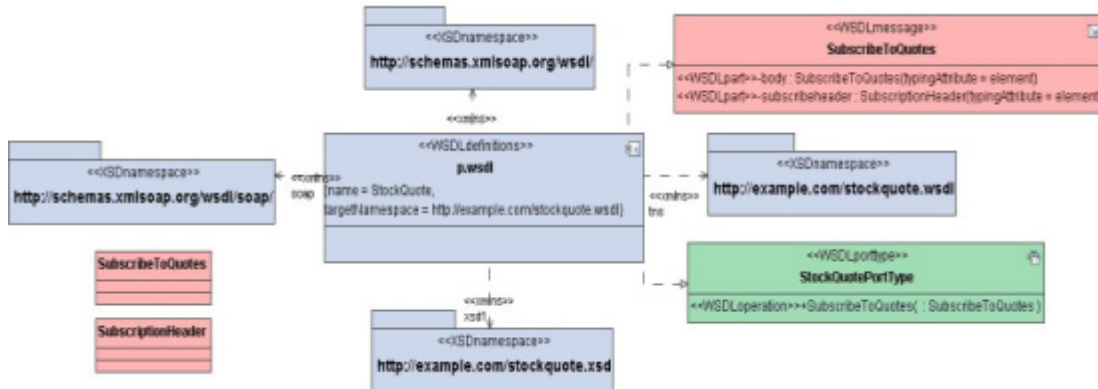
  <types>
    <schema targetNamespace="http://example.com/stockquote.xsd"
      xmlns="http://www.w3.org/2000/10/XMLSchema">
      <element name="SubscribeToQuotes">
        <complexType>
```

```

        <all>
          <element name="tickerSymbol" type="string"/>
        </all>
      </complexType>
    </element>
    <element name="SubscriptionHeader" type="uriReference"/>
  </schema>
</types>
</definitions>

```

Reversed UML model example:



## Bindings

A **binding** defines message format and protocol details for operations and messages defined by a particular portType. There may be any number of bindings for a given portType.

The **name** attribute provides a unique name among all bindings defined within in the enclosing WSDL document.

A binding references the portType that it binds using the **type** attribute. Binding extensibility elements are used to specify the concrete grammar for the input, output, and fault messages. Per-operation binding information as well as per-binding information may also be specified.

Example:

```

<definitions name="StockQuote"
  targetNamespace="http://example.com/stockquote.wsdl"
  xmlns:tns="http://example.com/stockquote.wsdl"
  xmlns:xsd="http://www.w3.org/2000/10/XMLSchema"
  xmlns:xsd1="http://example.com/stockquote.xsd"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns="http://schemas.xmlsoap.org/wsdl/">

  <message name="GetTradePriceInput">
    <part name="tickerSymbol" element="xsd:string"/>
    <part name="time" element="xsd:timeInstant"/>
  </message>

  <message name="GetTradePriceOutput">
    <part name="result" type="xsd:float"/>
  </message>

  <portType name="StockQuotePortType">
    <operation name="GetTradePrice">
      <input message="tns:GetTradePriceInput"/>
      <output message="tns:GetTradePriceOutput"/>
    </operation>
  </portType>

```

```

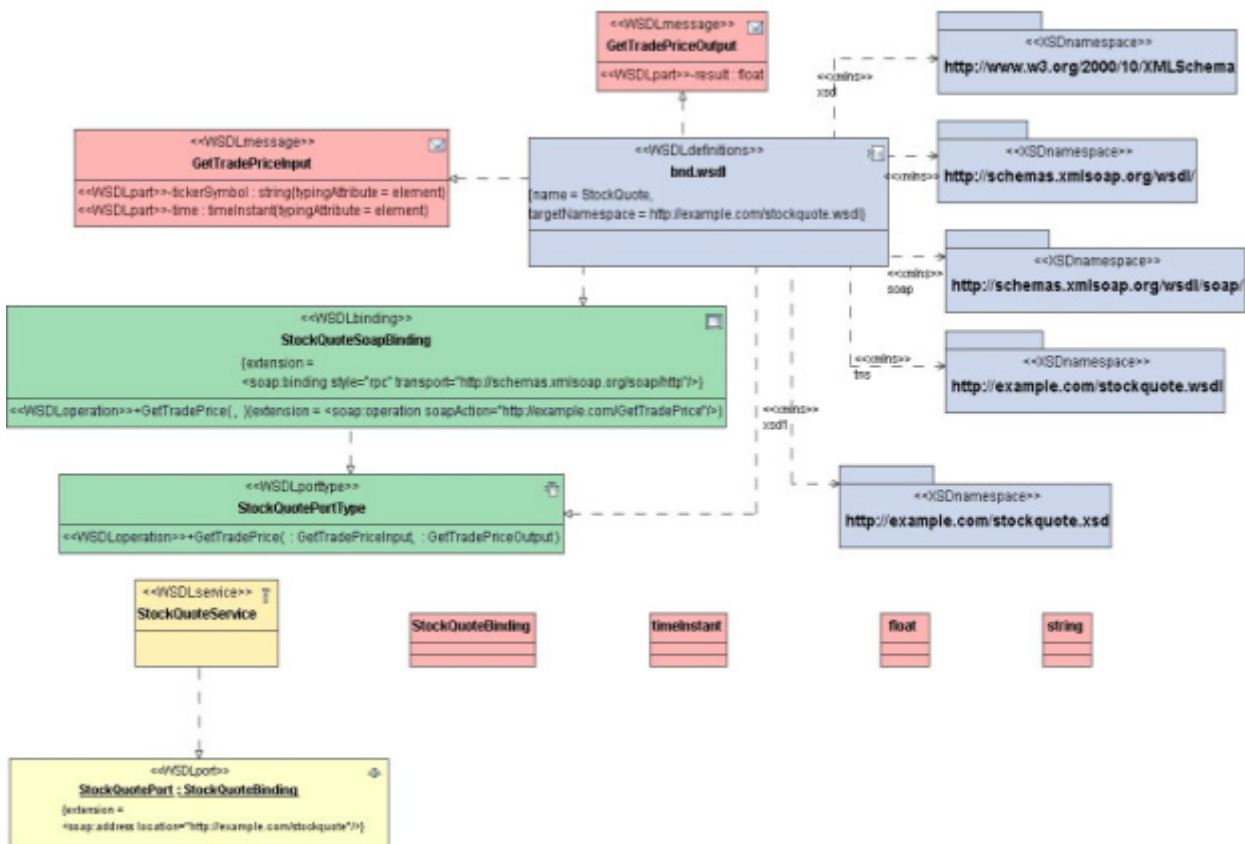
    </operation>
  </portType>

  <binding name="StockQuoteSoapBinding" type="tns:StockQuotePortType">
    <soap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>
  >
    <operation name="GetTradePrice">
      <soap:operation soapAction="http://example.com/GetTradePrice"/>
      <input>
        <soap:body use="encoded" namespace="http://example.com/stockquote"
          encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" />
      </input>
      <output>
        <soap:body use="encoded" namespace="http://example.com/stockquote"
          encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" />
      </output>
    </operation>
  </binding>

  <service name="StockQuoteService">
    <documentation>My first service</documentation>
    <port name="StockQuotePort" binding="tns:StockQuoteBinding">
      <soap:address location="http://example.com/stockquote"/>
    </port>
  </service>
</definitions>

```

Reversed UML model example:



## Services

A **service** groups a set of related ports together.

The **name** attribute provides a unique name among all services defined within in the enclosing WSDL document.

Ports within a service have the following relationship:

- None of the ports communicate with each other (e.g. the output of one port is not the input of another).
- If a service has several ports that share a port type, but employ different bindings or addresses, the ports are alternatives. Each port provides semantically equivalent behavior (within the transport and message format limitations imposed by each binding).
- By examining it's ports, we can determine a service's port types. This allows a consumer of a WSDL document to determine if it wishes to communicate to a particular service based whether or not it supports several port types.

Example:

```
<definitions name="StockQuote"
  targetNamespace="http://example.com/stockquote.wsdl"
  xmlns:tns="http://example.com/stockquote.wsdl"
  xmlns:xsd1="http://example.com/stockquote.xsd"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns="http://schemas.xmlsoap.org/wsdl/">

  <types>
    <schema targetNamespace="http://example.com/stockquote.xsd"
      xmlns="http://www.w3.org/2000/10/XMLSchema">
      <element name="TradePriceRequest">
        <complexType>
          <all>
            <element name="tickerSymbol" type="string"/>
          </all>
        </complexType>
      </element>
      <element name="TradePrice">
        <complexType>
          <all>
            <element name="price" type="float"/>
          </all>
        </complexType>
      </element>
    </schema>
  </types>

  <message name="GetLastTradePriceInput">
    <part name="body" element="xsd1:TradePriceRequest"/>
  </message>

  <message name="GetLastTradePriceOutput">
    <part name="body" element="xsd1:TradePrice"/>
  </message>

  <portType name="StockQuotePortType">
    <operation name="GetLastTradePrice">
      <input message="tns:GetLastTradePriceInput"/>
      <output message="tns:GetLastTradePriceOutput"/>
    </operation>
  </portType>

  <binding name="StockQuoteSoapBinding" type="tns:StockQuotePortType">
```

```

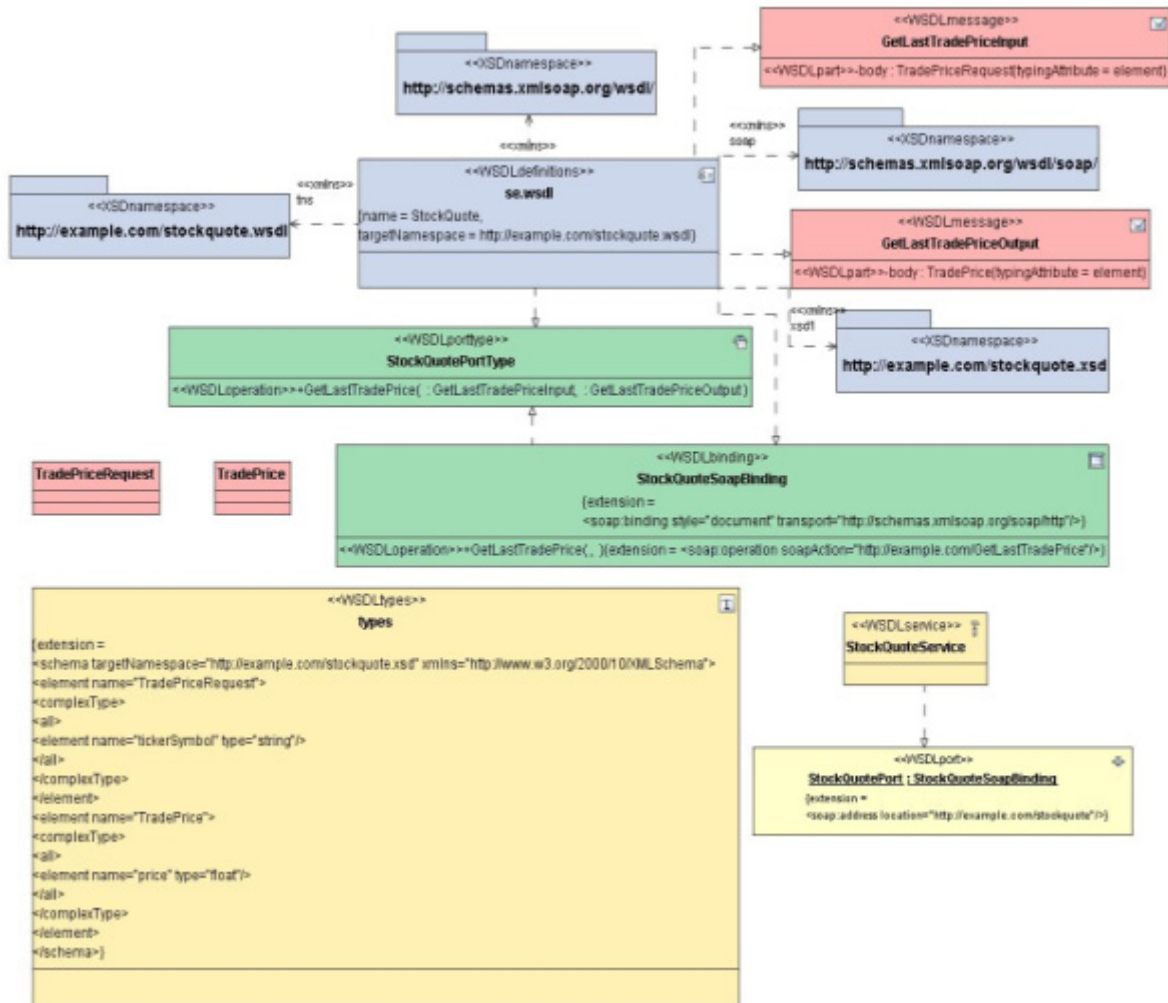
<soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/
http"/>
  <operation name="GetLastTradePrice">
    <soap:operation soapAction="http://example.com/GetLastTradePrice"/>
    <input>
      <soap:body use="literal"/>
    </input>
    <output>
      <soap:body use="literal"/>
    </output>
  </operation>
</binding>

<service name="StockQuoteService">
  <documentation>My first service</documentation>
  <port name="StockQuotePort" binding="tns:StockQuoteSoapBinding">
    <soap:address location="http://example.com/stockquote"/>
  </port>
</service>

</definitions>

```

Reversed UML model example:



## Ports

A port defines an individual endpoint by specifying a single address for a binding.

The **name** attribute provides a unique name among all ports defined within in the enclosing WSDL document.

The **binding** attribute refers to the binding using the linking rules defined by WSDL.

Binding extensibility elements are used to specify the address information for the port.

A port must not specify more than one address.

A port must not specify any binding information other than address information.

Example:

```
<definitions name="HelloService"
  targetNamespace="http://www.ecerami.com/wsdl/HelloService.wsdl"
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:tns="http://www.ecerami.com/wsdl/HelloService.wsdl"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">

  <message name="SayHelloRequest">
    <part name="firstName" type="xsd:string"/>
  </message>
  <message name="SayHelloResponse">
    <part name="greeting" type="xsd:string"/>
  </message>

  <portType name="Hello_PortType">
    <operation name="sayHello">
      <input message="tns:SayHelloRequest"/>
      <output message="tns:SayHelloResponse"/>
    </operation>
  </portType>

  <binding name="Hello_Binding" type="tns:Hello_PortType">
    <soap:binding style="rpc"
      transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="sayHello">
      <soap:operation soapAction="sayHello"/>
      <input>
        <soap:body
          encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
          namespace="urn:examples:helloservice"
          use="encoded"/>
      </input>
      <output>
        <soap:body
          encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
          namespace="urn:examples:helloservice"
          use="encoded"/>
      </output>
    </operation>
  </binding>

  <service name="Hello_Service">
    <documentation>WSDL File for HelloService</documentation>
    <port binding="tns:Hello_Binding" name="Hello_Port">
      <soap:address
        location="http://localhost:8080/soap/servlet/rpcrouter"/>
    </port>
  </service>
</definitions>
```

Reversed UML model example:

