



CODE ENGINEERING AND DATABASE

version 14.0
user's guide

No Magic, Inc.
September, 2007

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 1998-2007 by No Magic, Inc.

CONTENTS

INTRODUCTION 9

Overview 9

Code Engineering Sets 10

Generating Code 15

 Code Generation for Set 15

 Code Generation for Model Element 17

Reverse 18

Global options for Code Engineering 21

 Code engineering options for all sets in your project 21

 Java Documentation Properties dialog box 27

 Round Trip 28

 Type Mapping Table 29

Files of Properties 29

JAVA CODE ENGINEERING 31

Introduction 31

 Abbreviations 31

 References 31

 Java support in MagicDraw 31

Java Mapping to UML 32

 Java Profile 32

 Java referenced types 34

 Mapping to UML rules 34

Java CE Properties 66

 Java Reverse Properties 66

 Java Language Options 68

Method Implementation Reverse 72

 Sequence Diagram from Java Source Wizard 72

CONTENTS

C++ CODE ENGINEERING 77

Abbreviations 77

References 77

C++ ANSI Profile 77

Data type 77

Stereotype 81

Mapping 91

Class 91

Base class definition 91

Class member variable 93

Class member function 93

Class constructor/destructor 94

Variable 94

Variable modifiers 97

Variable extern 99

Variable default value 99

Const Volatile qualified type 101

Function 101

Function variable-length parameter list 102

void parameter 103

Register parameter 104

Function modifiers 105

Function pointer 106

Function operator 107

Exception 107

Visibility 108

Static members 109

Friend declaration 110

Struct 111

Union 112

Enumeration 112

Typedef 113

Namespace 115

Global functions and variables 115

Class definition 116

CONTENTS

Class Template Definition	116
Function Template Definition	118
Default template parameter	119
Template instantiation	120
Partial template instantiation	125
Template specialization	127
Forward class declaration	127
Include declaration	130
Conversion from old project version	133
Translation Activity Diagram	133
Language properties	139
Type Modifiers	171
Stereotypes	177
Tag Value	193
Constructor and Destructor name	195
Data type	199
DSL customization	208
Operation and Constructor	208
Attribute	210
Generalization	211
Enumeration literal	212
Namespace	212
Template parameter	213
Profile constraints	213
Operation	214
Constructor	214
Destructor	214
Global	214
Typedef	214
Friend	214
New in MagicDraw 12.1	216
CG Properties Editor	216
Roundtrip on #include statement and forward class declaration	217
Project Option and Code Generation Options	221
New in MagicDraw 14.0	226

CONTENTS

Support C++ dialects	226
CG Properties Editor	227
Tutorial	229
Type Modifier	229
Global Member	229
Typedef	230
Function Pointer	233
Friend	235
How to specify component to generate code to	237
Project constraint	240
Working with QT	246
Microsoft	248
Microsoft Visual C++ Profile	248
C++/CLI Profile	258
C++ Managed Profile	270
C# CODE ENGINEERING	315
C# 2.0 Description	315
Generics	315
Anonymous Methods	348
Partial Types	348
Nullable Types	353
Accessor Declarations	354
Static Class	356
Extern Alias Directive	359
Pragma Directives	362
Fix Size Buffer	363
C# 3.0 Description	367
Extension Methods	367
Lambda Expression Conversion	369
C# Profile	371
Stereotype	373
Data Type	379
Conversion from old project version	380
Translation Activity Diagram	380

CONTENTS

Mapping **386**

- Language Properties Mapping **386**
- C# Properties Customization **390**
- Using Directive Mapping **392**

Constraints **399**

- Mapping Constraints **399**
- UML Constraints **399**
- Translation Constraints **403**

DATABASE ENGINEERING **405**

- Retrieve DB Info dialog box **405**

Forward engineering to DDL script **409**

- Packages **409**
- Classes **410**
- Attributes **411**
- Operations **412**
- Relationship cardinalities **414**
- Inheritance **416**
- Not supported UML constructs **417**

Reverse engineering for DDL script **418**

- Database **418**
- Schema **419**
- Table **419**
- Column **420**
- Constraint **421**

Unnamed constraint representation as a stereotype of an attribute **426**

- Index **427**
- Trigger **428**
- View **429**

DDL dialects **431**

- Standard SQL2 **431**
- Cloudscape **432**
- Oracle Oracle8 **432**

Stereotypes for MagicDraw constructs **432**

Properties of code engineering set for DDL **433**

CONTENTS

Properties for DDL script reverse engineering and generation	434
Supported SQL statements	438
Tips	440
Short representation for primary key constraint	440
Primary key constraint with overhead info	441
CORBA IDL MAPPING TO UML	443
EJB 2.0 - UML	447
Java Profile	447
EJB Design Profile	449
EJB Deployment Profile	456
Using MagicDraw EJB 2.0	460
Reverse engineering	460
Code generation	461
XML SCHEMA	463
XML Schema Mapping to UML Elements	463
Defined stereotypes	463
attribute	467
element	469
complexType	472
attributeGroup	479
simpleType	480
restriction	482
list	482
union	482
minExclusive	492
maxExclusive	492
minInclusive	493
maxInclusive	494
totalDigits	495
fractionDigits	496
length	497

CONTENTS

minLength	498
maxLength	499
whiteSpace	500
pattern	501
enumeration	501
unique	503
key	503
keyref	503
selector and field	508
annotation	512
compositors	514
group	518
any and anyAttribute	519
schema	521
notation	523
redefine	524
import	527
include	529
XML schema namespaces	530
XSD FILE CREATION WITH MAGICDRAW	531
 WSDL	533
WSDL Mapping to UML elements	534
Defined stereotypes	534
Definitions	534
Import, namespace	535
Messages	536
Types	537
Port types	539
Bindings	541
Services	543
Ports	545

CONTENTS

TRANSFORMATIONS	549
UML to DDL transformation	550
Type mapping	550
Transformation results	560
DDL to UML transformation	562
Type mapping	562
Transformation results	566
Generic DDL to Oracle DDL transformation	570
Generic DDL to Oracle DDL type mapping	570
UML to XML Schema transformation	579
Type mapping	579
Transformation results	582
XML Schema to UML transformation	585
Type mapping	585
Transformation Results	585

INTRODUCTION

Overview

[View Online Demos](#)

Code Generation

Code Reverse

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	-	+

Language	Professional Edition	Enterprise Edition
CIL	C#	+
CIL Disassembler	C#	+
XML Schema	-	+
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.

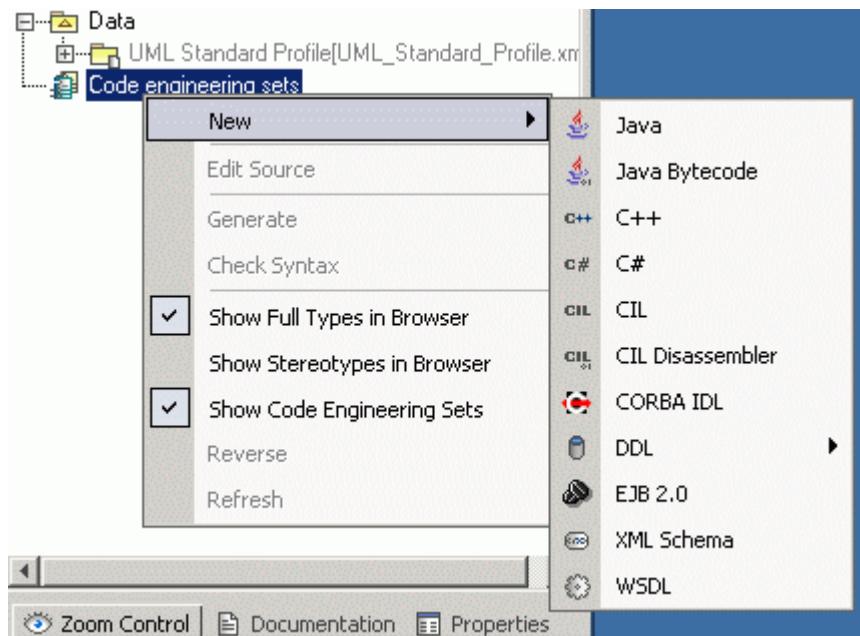


Figure 1 -- Code engineering language options

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.

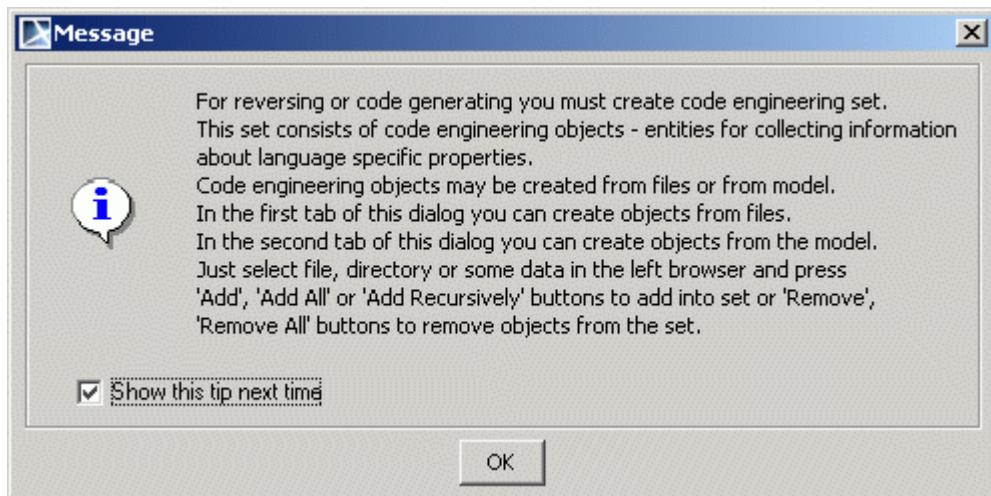


Figure 2 -- Code Engineering Sets tip message box

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.

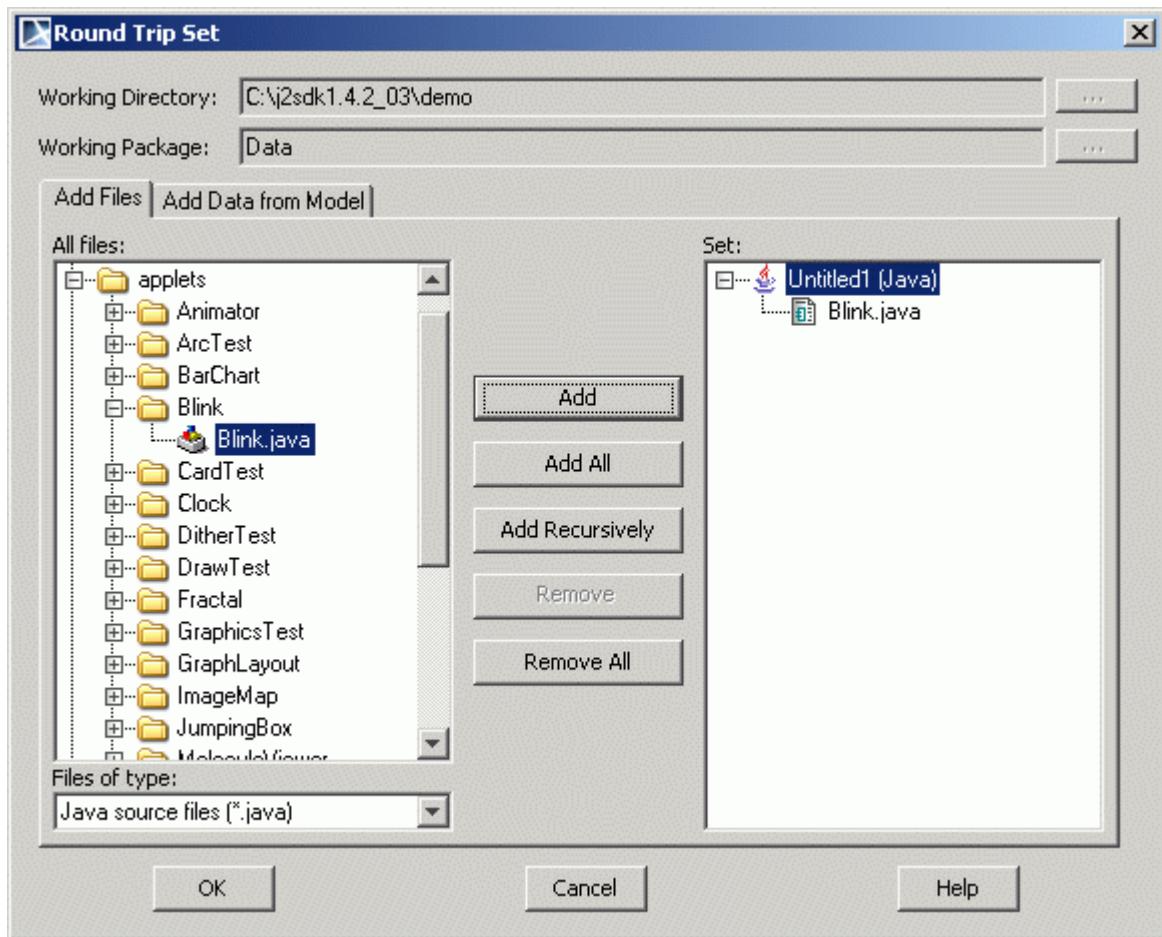


Figure 3 -- Round Trip Set dialog box. Add files tab

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
All files	Helps you find directories with the source files for the set.
Files of type	Contains possible file name extensions for the chosen language.

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

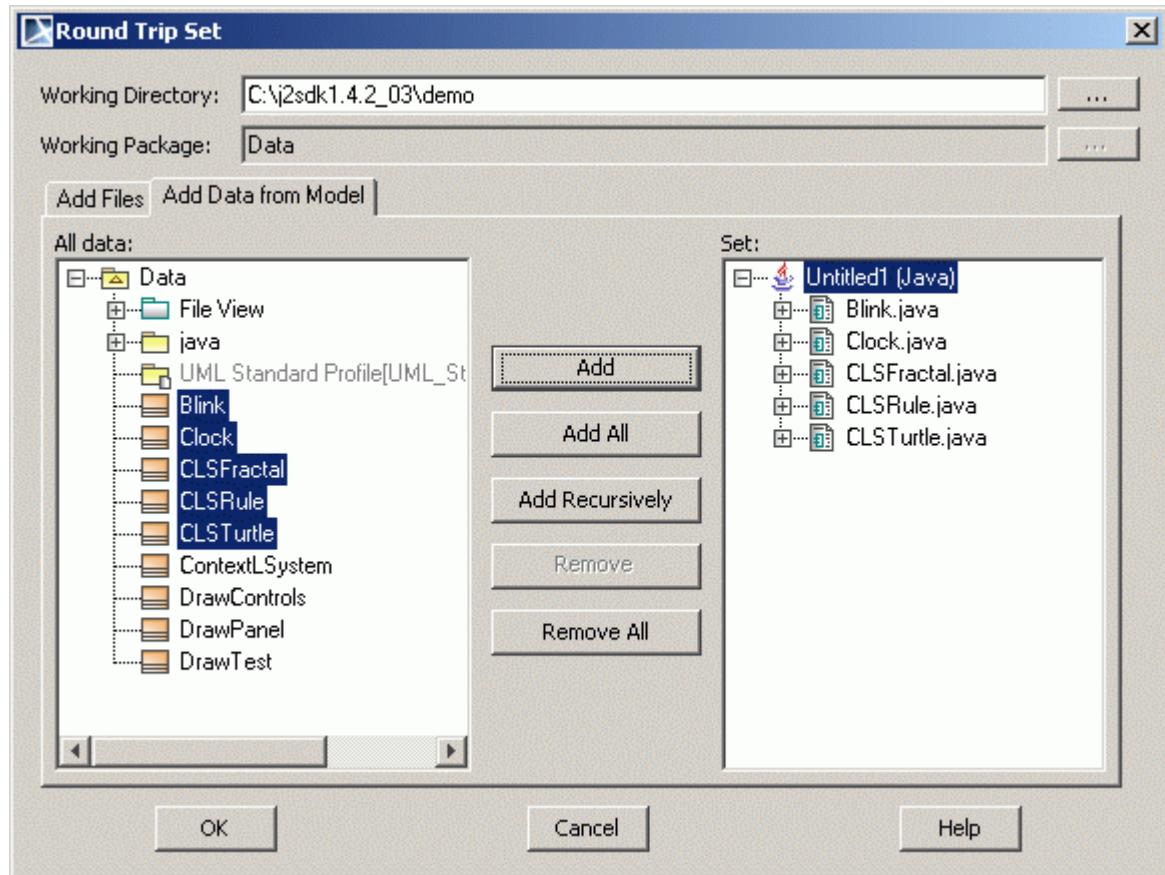


Figure 4 -- Round Trip Set dialog box. Add data from model tab

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:

Add	The selected file in the All Files or All Data list is added to the set.
-----	--

Add All	All files in the opened or selected directory are added to the set.
Add Recursively	All files in the selected directory and its subdirectories are added to the set.
Remove	Removes the selected entity from the set.
Remove All	Removes all entities from the set.

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 10.

- 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.

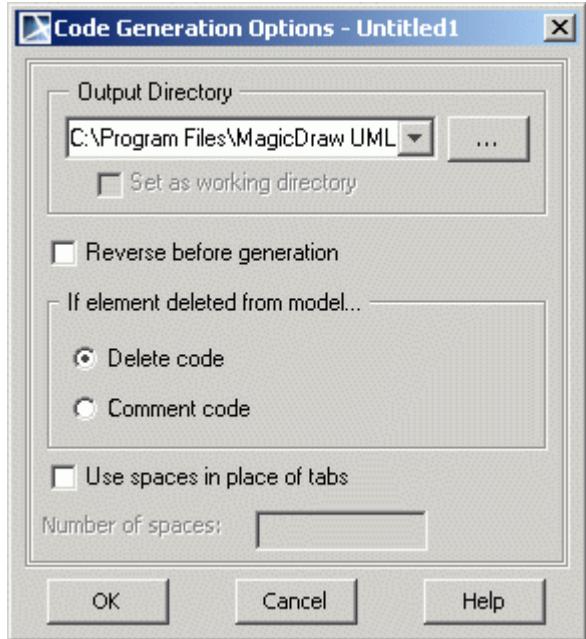


Figure 5 -- Code Generation Options dialog box.

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
Output Directory	Type the directory where the generated files will be saved.
'...'	The Set Output Directory dialog box appears. Select the directory for the generated files.
Set as Working Directory	The output directory is set as a working directory and files are saved to the working directory.
Reverse before generation	Changes your model according to changes in the existing code. WARNING: Exercise caution when selecting the Reverse before generation check box. If the model differs from the code, all differences in the model will be discarded. In such cases, you will lose some of your work.

Box name	Function
If element deleted from model	To influence the structure of generated code, click one of the following option buttons: <ul style="list-style-type: none"> ● Delete code. The representation of deleted entities will be deleted from the code file. ● Comment code. Deleted entities will be commented in the code file.
Use spaces in place of tabs	When selected, spaces (instead of tabs) will be written to the code file.
Number of spaces	Specify the number of spaces to be written.
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 10)

- 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 its shortcut menu.

Models can be reversed without creating a set.

To reverse a model without creating a set

1. From the **Tools** menu, choose **Quick Reverse**. The **Round Trip Set** dialog box appears.

NOTE:

Quick Reverse is available only in Professional and Enterprise editions.

2. Select the files from the **Round Trip Set** dialog box, **Add Files** tab.

INTRODUCTION

Reverse

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

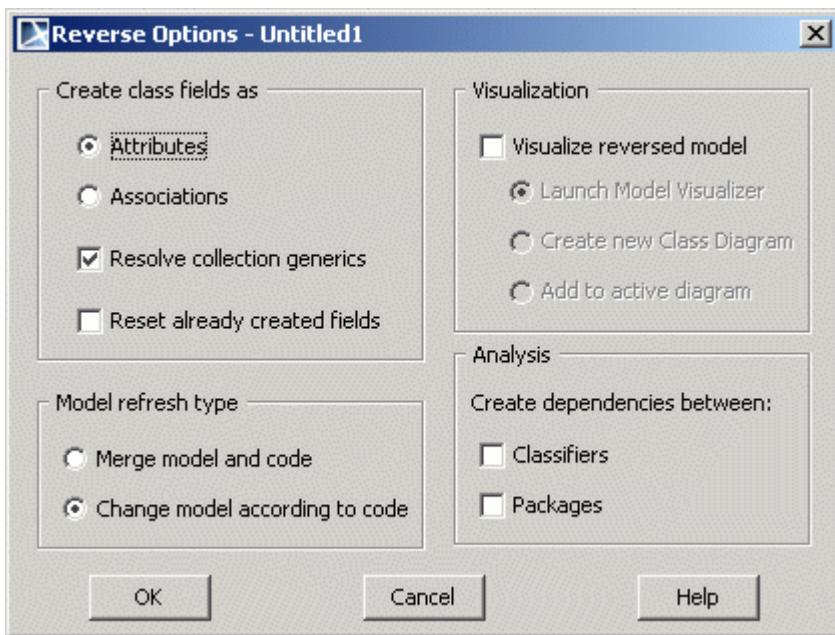


Figure 6 -- Reverse options dialog box

Element name	Function
CREATE CLASS FIELDS AS	
Attributes	Class fields are represented in model as attributes.
Associations	Class fields are represented in model as association ends.
Resolve collection generics	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: <code>:java.util.List<\$\$type\$\$></code> where \$\$type\$\$ replaced to value of "type" property when code is generated.
Reset already created fields	Select this option if you want to keep already created UML representation (attribute or association) for class fields.
MODEL REFRESH TYPE	

Element name	Function
Merge model and code	The model elements are updated by code. Elements that do not exist in the code will not be removed from the model.
Change model according to code	Model will be created strictly by code. Entities in the model that do not match entities in the code will be discarded.
VIZUALIZATION	
Visualize reversed model	Classes that are created while reversing can be added to a diagrams.
Launch Model Visualizer	After reversing, the Model Visualizer dialog box appears. It will assist you in creating a class diagram or sequence diagram (Java only) for newly created entities.
Create new class diagram	After reversing, the Create Diagram dialog box appears. Create a new diagram where the created entities will be added.
Add to active diagram	After reversing, all created entities will be added to the current opened diagram.
ANALYSIS - create dependencies between	
Classifiers	Dependencies between classes will be analyzed and created.
Packages	Dependencies only between packages will be created.
OK	Saves changes and exits the dialog box.
Cancel	Exits dialog box without saving changes.
Help	Displays MagicDraw Help.

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.

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.

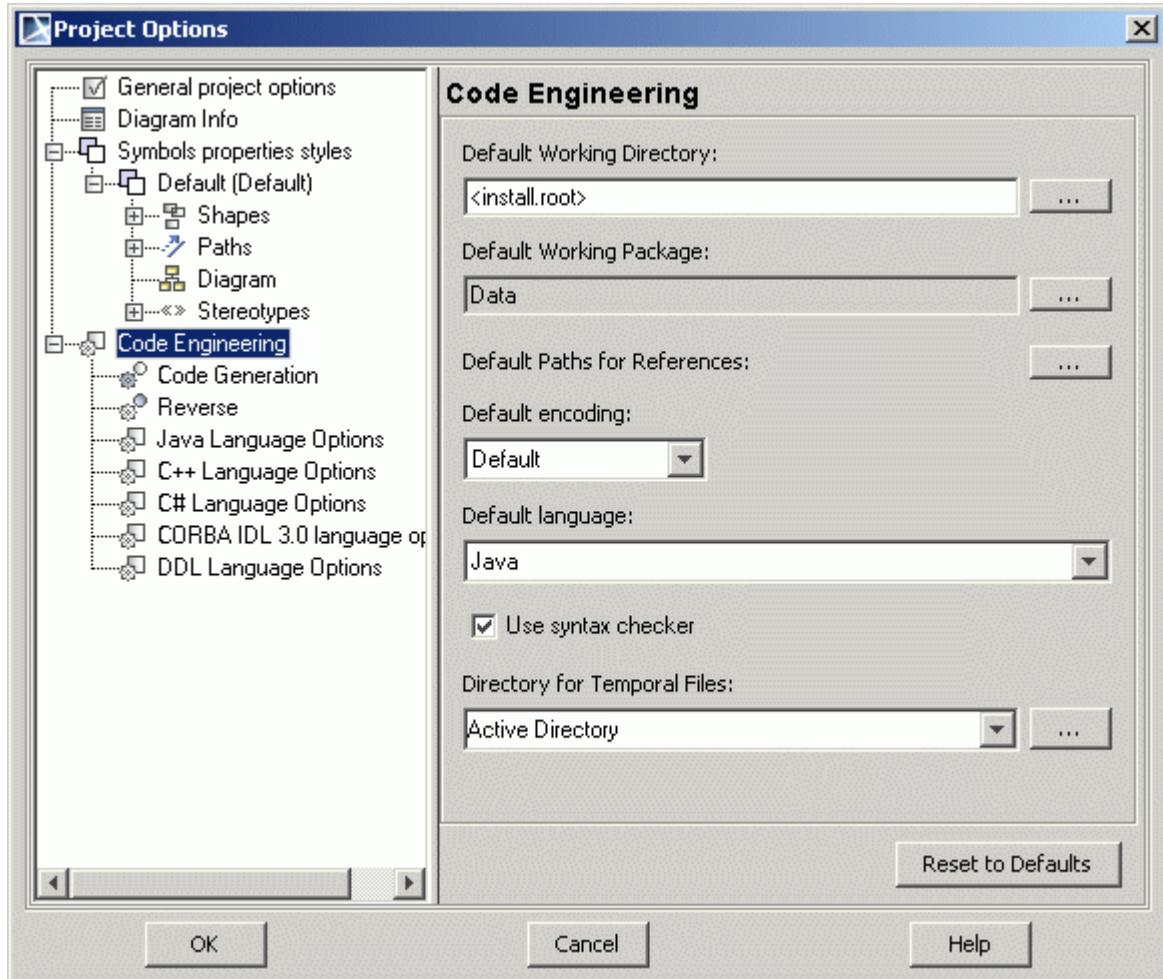


Figure 7 -- Project Options dialog box

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:

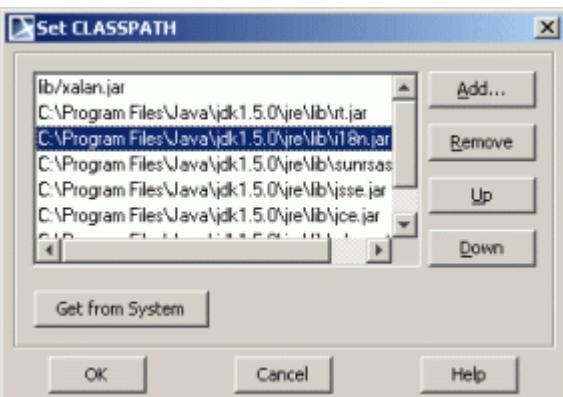
- **Styles** – expands the tree hierarchy of all the styles defined within the project. You may use as many of these styles as you wish. See MagicDraw main User's Manual, working with Projects Section.
- **Code engineering** – these options are found on the right side of the Project options dialog box:
 - **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
Code generation	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 “Generating Code” on page 15).
Reverse	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 18).

INTRODUCTION

Global options for Code Engineering

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

Tab name	Element name	Function
	Use CLASSPATH	The '...' button is activated. Search a classpath for importing sentences generation in the Set classpath dialog box. 
		Click the Get from System button to get CLASSPATH variable defined by operating system or click the Add button and select the classpath directory in the Add Classpath dialog box.
	Java Source	Available choices 1.4 or 5.0
	Header	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.
C++ Language Options Set the generated code style for C++ programming languages.	Generate opening bracket in new line	Opens a bracket in the newly generated line.
	Generate spaces	Spaces inside an assignment and other operators are generated.
	Generate empty documentation	Comment brackets are placed in your code, unless class in the model has no documentation.
	Generate methods body into class	Select check box to generate methods body into class.
	Documentation Style	Two styles are available for documentation.

INTRODUCTION

Global options for Code Engineering

Tab name	Element name	Function
	Use include path	Click the '...' button and then specify the path for the includes in the Set Include Path dialog box.
	Use explicit macros	Select check box. The '...' button is activated, click it and in the C++ Macros dialog box use a set of predefined macros.
	Header	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.
CORBA IDL 3.0 Language Options	Generate documentation	Includes the documentation of an element in the comment.
	Generate opening bracket in new line	Opens a bracket in the new line generating.
	Generate spaces	Spaces inside an assignment and other operators are generated.
	Generate empty documentation	Comment brackets are placed in your code, unless class in the model has no documentation.
	Generate imports	Generation of "import" statements for classes that are referenced in the generated class.
	Generate pre-processor directives	Generates pre-processors directives.
	Documentation Style	Three styles are available for documentation.
	Header “...”	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.
	Set Include Path	Specify the path for the "includes". Click the "..." button to open the Select Folder dialog box.
DDL Language Options	Generate opening bracket in new line	Opens a bracket in the new line generating.

Tab name	Element name	Function
	Generate spaces	Spaces inside an assignment and other operators are generated.
	Generate documentation	Comment brackets are placed in your code, unless class in the model has no documentation.
	Header	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.
C# Language Options Set the generated code style for C# programming languages.	Generate opening bracket in new line	Opens a bracket in the newly generated line.
	Generate spaces	Generates spaces inside an assignment and other operators.
	Generate empty documentation	Comment brackets are placed in your code, unless class in the model has no documentation.
	Generate required "using" directives	Automatic generation of "using" directives. This option facilitates the usage of namespaces and types defined in other namespaces.
	Concatenate namespace names	If not selected namespace names are separated into several lines. e.g. namespace A { namespace B {
	Documentation: <ul style="list-style-type: none">● Processor● Style	<ul style="list-style-type: none">● Use C# XMI processor then generates c# xmi documentation for commenting the code.● Select one of the listed comment styles.
	Header	Adds the specific header to all your code files. Click the '...' button and type header text in the Header dialog box. You may also define \$DATE, \$AUTHOR, and \$TIME in the header.
	Conditional Symbols	Add the conditional symbols, which can not be recognized and should be skipped during reverse. Click the '...' button and add conditional symbols in the Define Conditional Symbols 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.

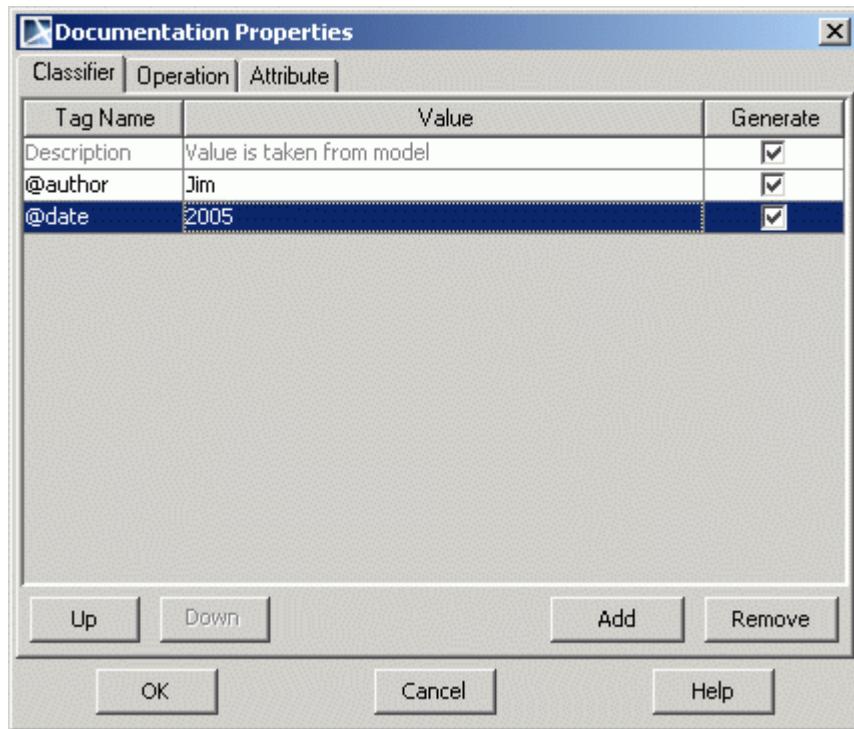


Figure 9 -- Documentation Properties dialog box

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.

Box Name	Function
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.

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.

Supported language	File of Properties
JAVA	java.prop
C++	C++.prop
CORBA IDL	idl.prop
JAVABYTECODE	javabytecode.prop
DDL	ddl.prop
CIL	cil.prop

Supported language	File of Properties
CIL Disassembler	cil disassembler.prop
C#	c#.prop
EJB	ejb.prop
EJB 2.0	ejb20.prop
IDL	idl.prop
XML Schema	xmleschema.prop
WSDL	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 32 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 66 introduces specific Java options.

Abbreviations

UML	Unified Modeling Language
JLS	Java Language Specification
CE	Code Engineering
CES	Code Engineering Set
JVM	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.

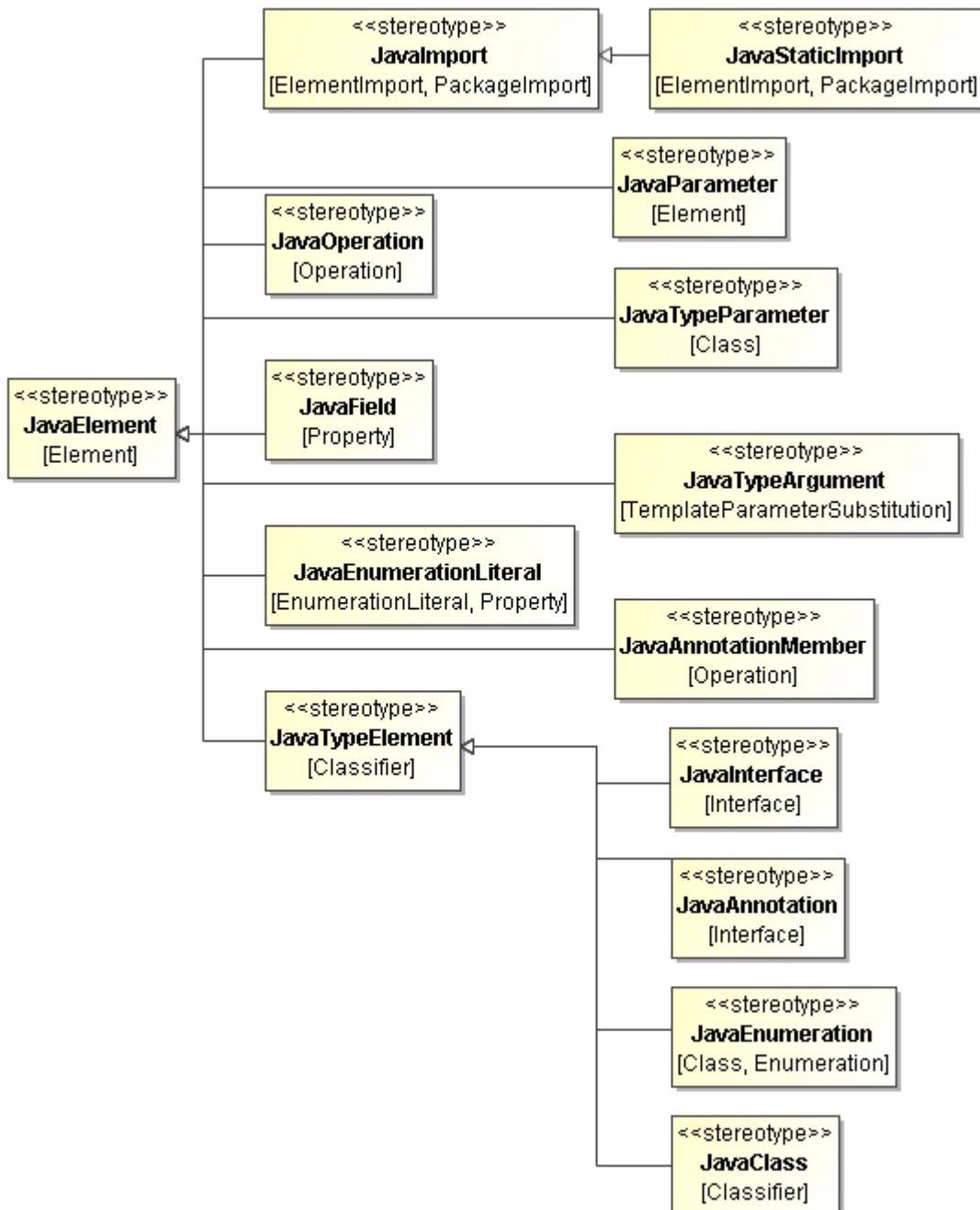


Figure 1 -- Stereotypes

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.

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.

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.

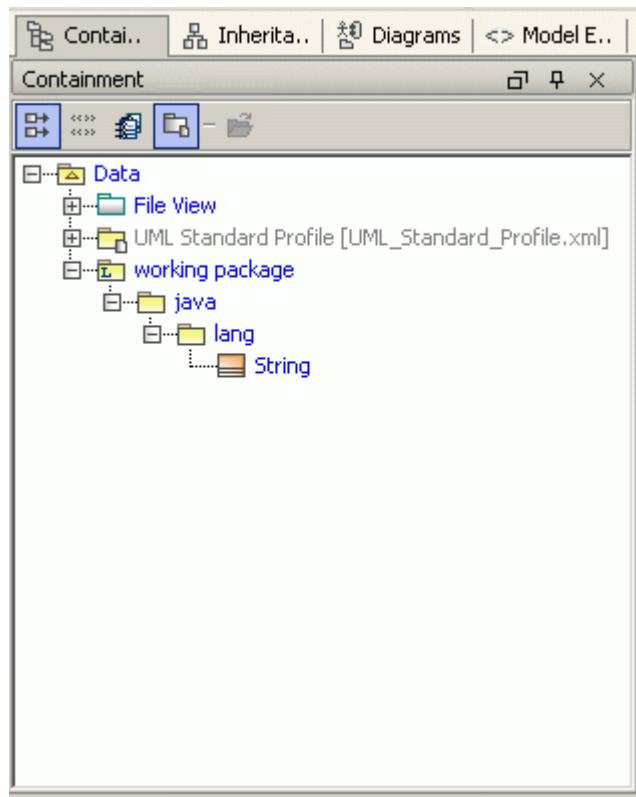


Figure 2 -- Package Structure

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
Class documentation	UML Class Documentation
Class extends clause	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 Leaf" 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

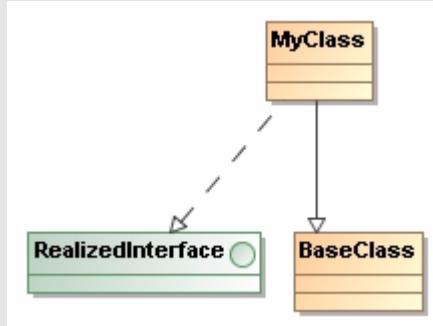


Figure 3 -- Class diagram

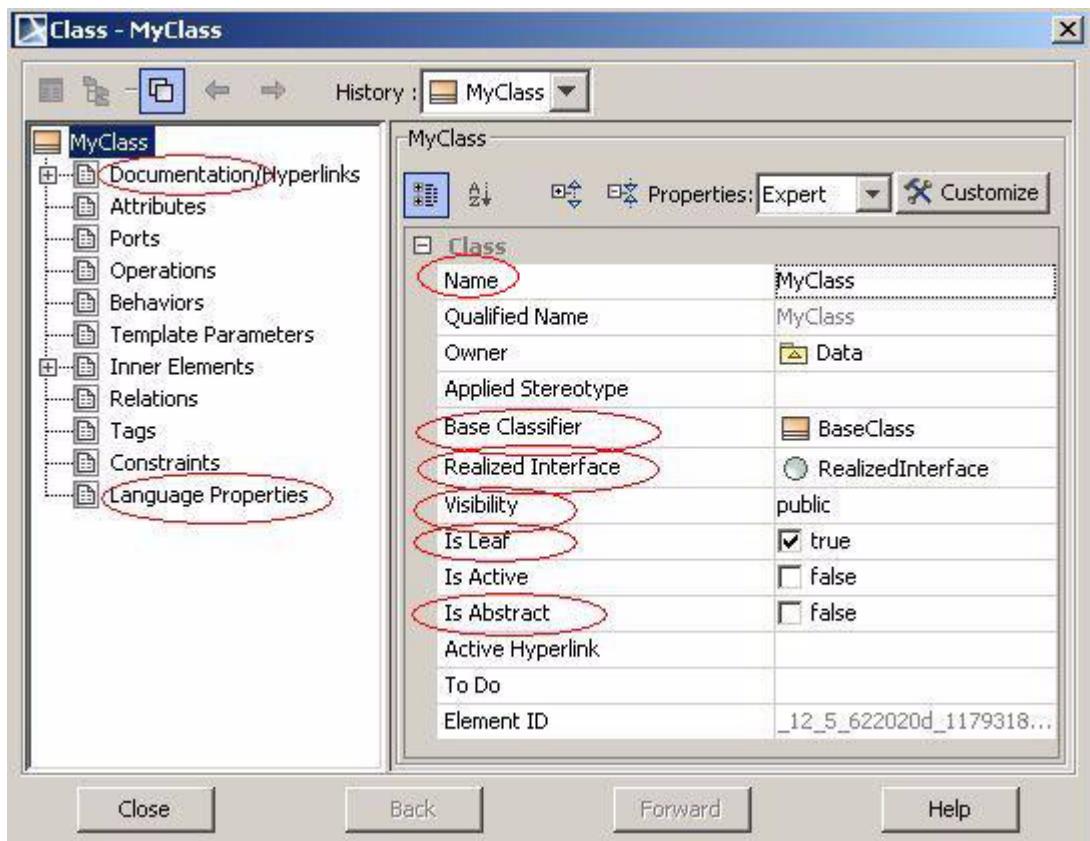


Figure 4 -- UML Class specification dialog

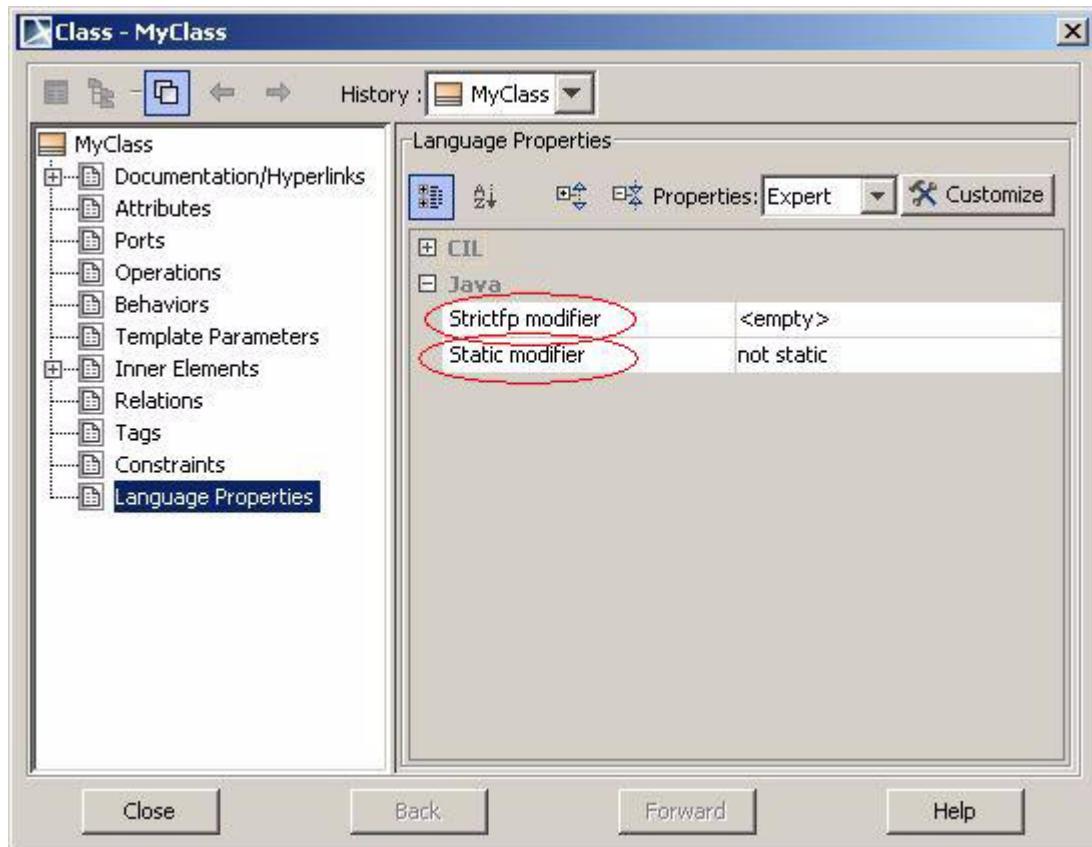


Figure 5 -- UML Class language properties

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

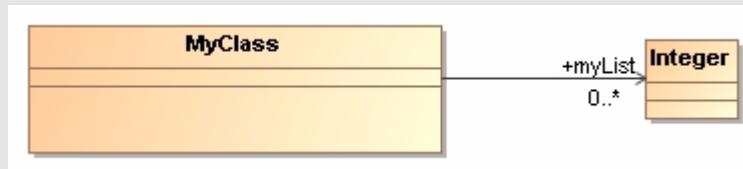


Figure 6 -- Class with property

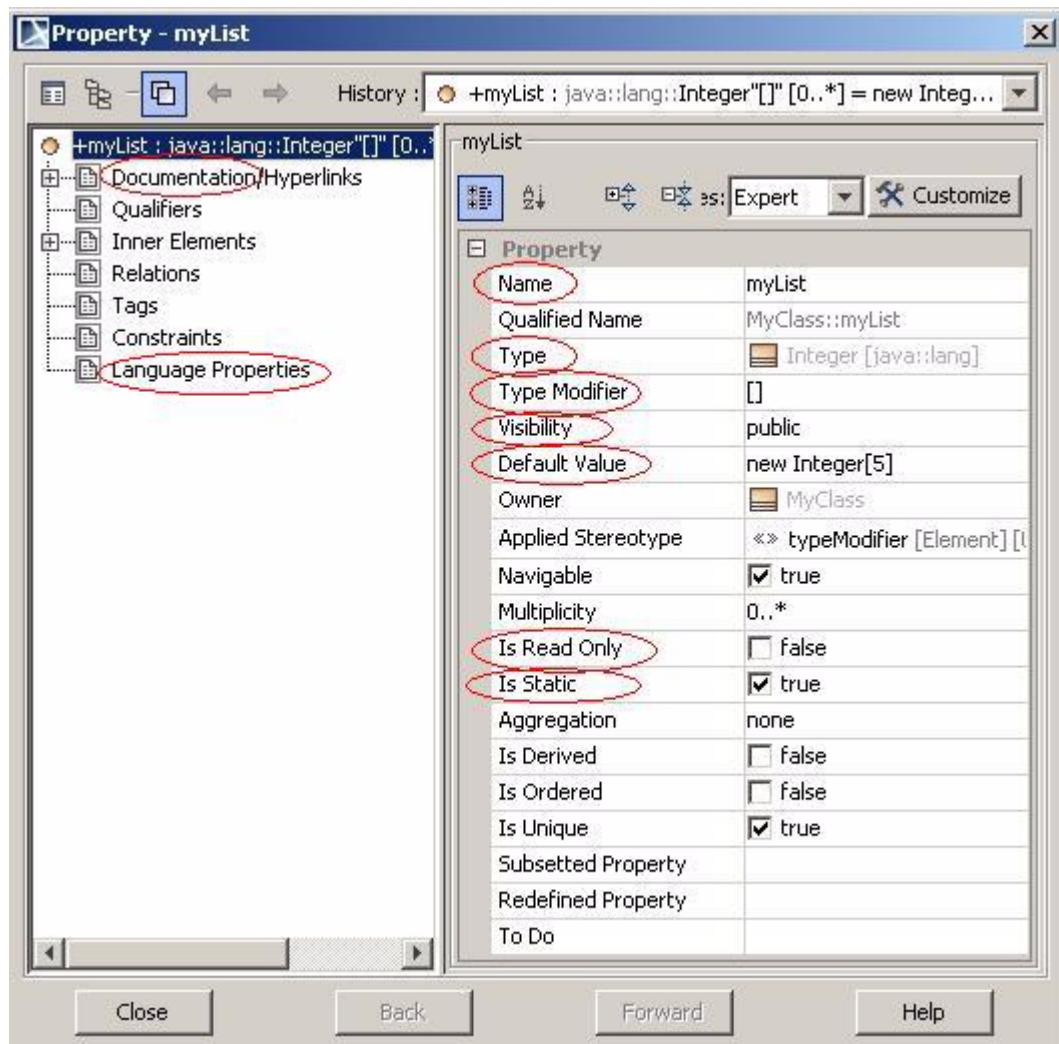


Figure 7 -- UML Property specification dialog

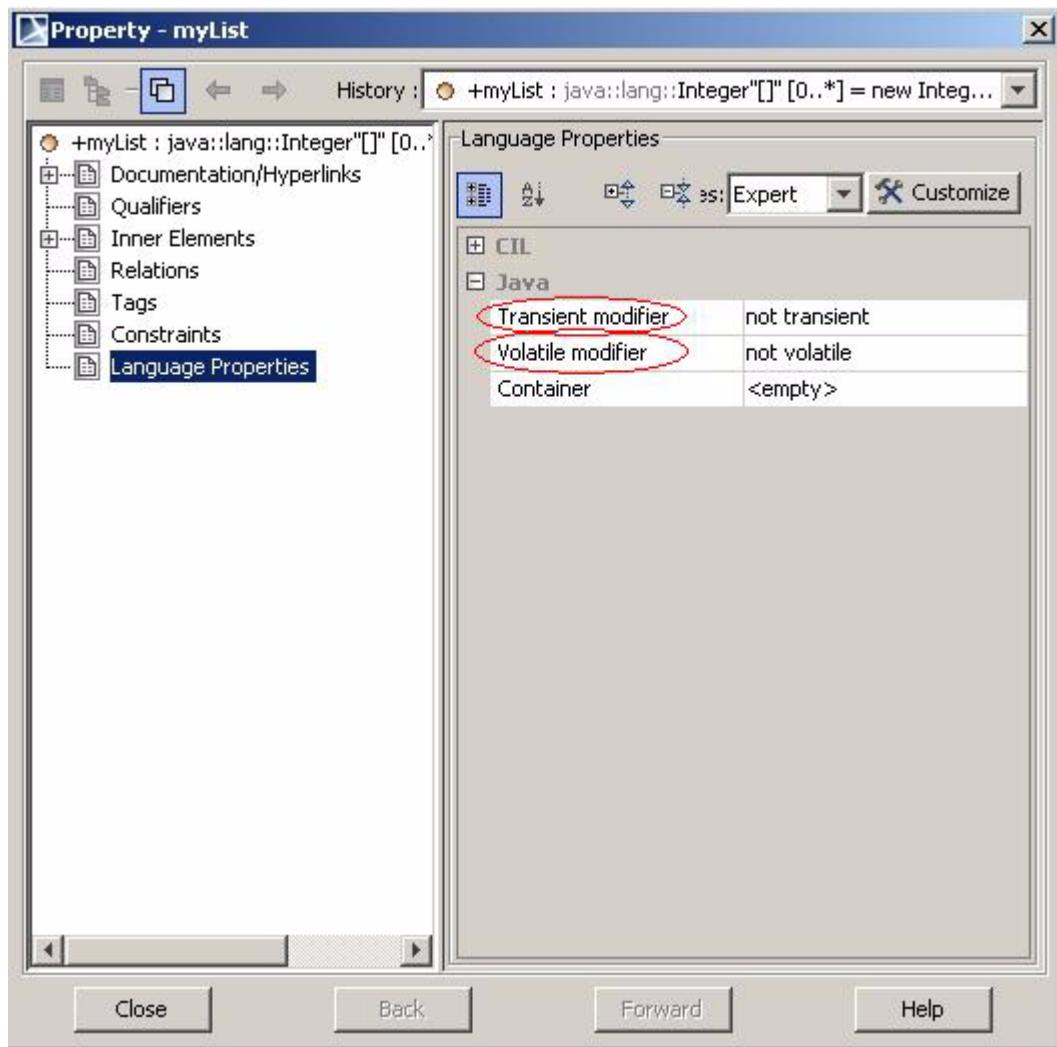


Figure 8 -- UML Property language properties

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 9 -- Class with operation

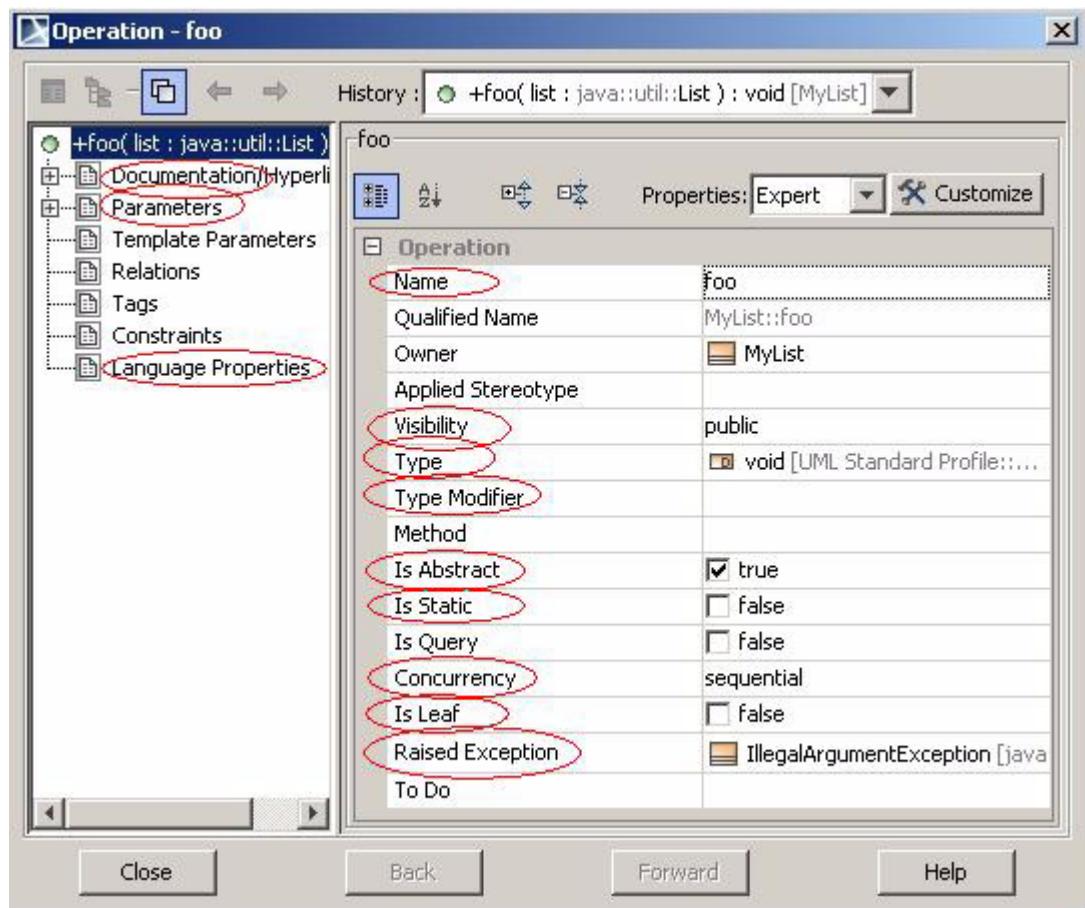


Figure 10 -- UML Operation specification dialog

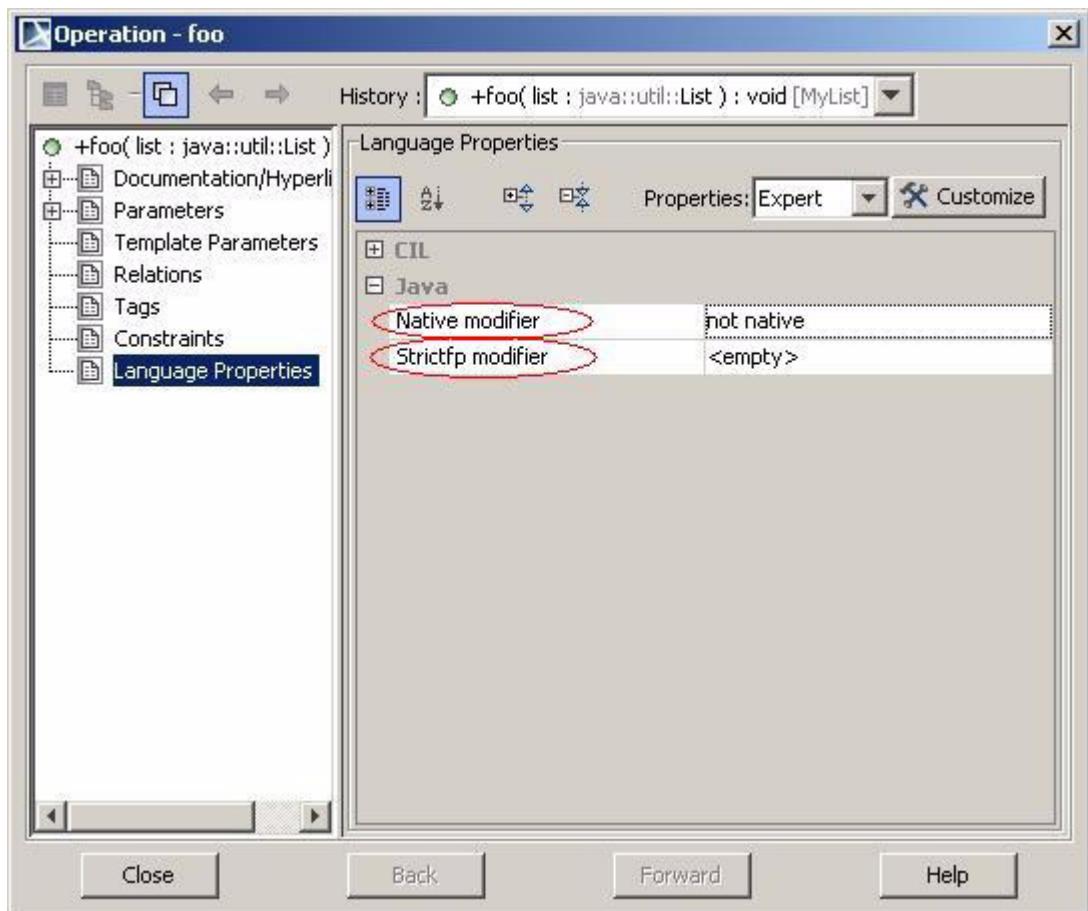


Figure 11 -- UML Operation language properties

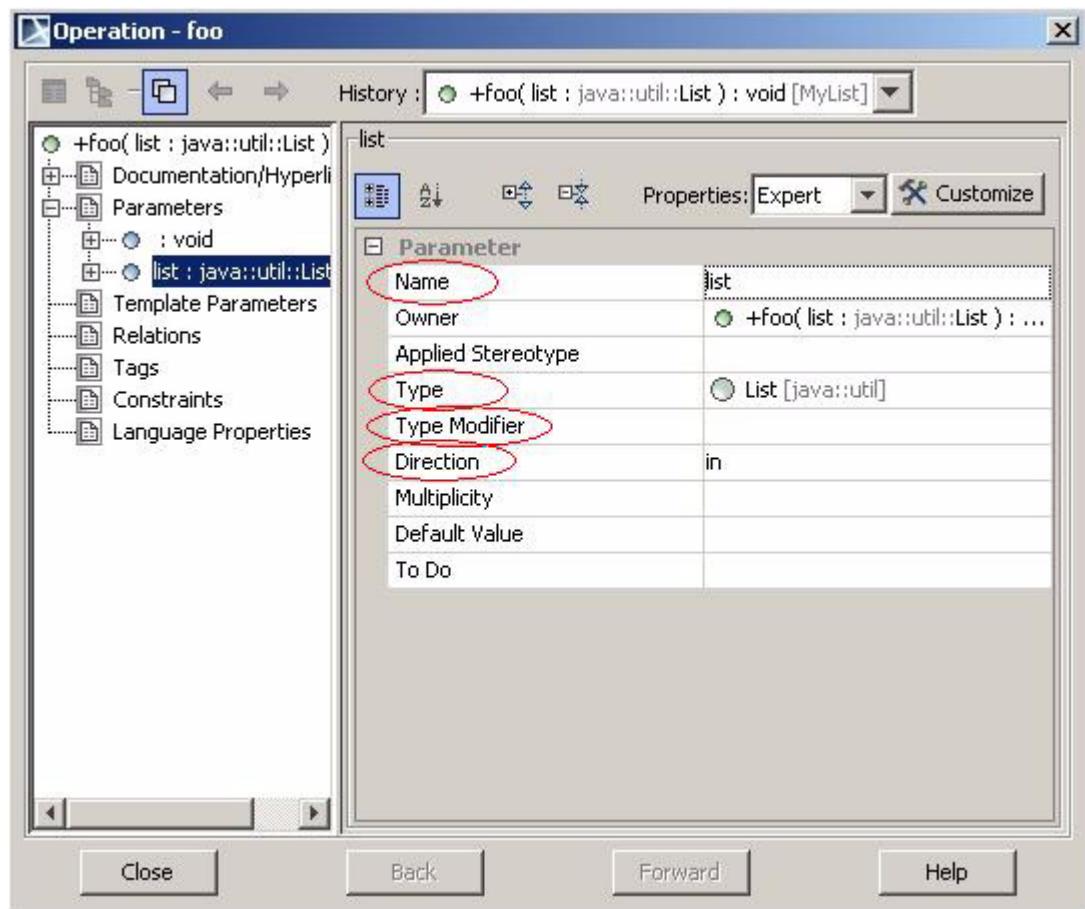


Figure 12 -- UML Parameter specification dialog

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 35.

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 35.

Example

Java Source Code

```
enum MyInterface
{
}
```

MagicDraw UML Model

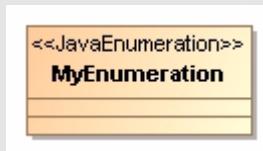


Figure 13 -- UML Enumeration

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 <<JavaEnumerationLiteral>>, owned by UML Class with stereotype <<JavaEnumeration>>
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

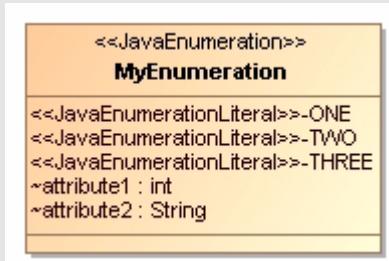


Figure 14 -- UML Class representing Java enumeration with enumeration literals

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 15 -- 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 <>JavaAnnotation-Member>> (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

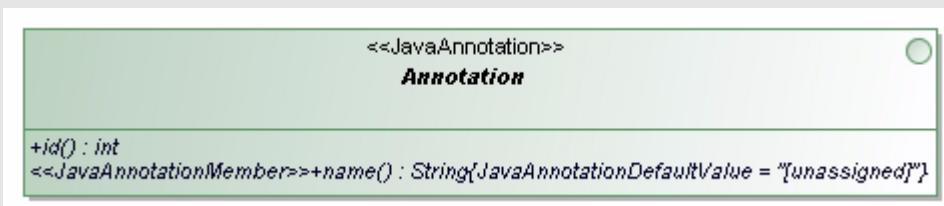


Figure 16 -- UML Interface representing Java annotation type with Java annotation members

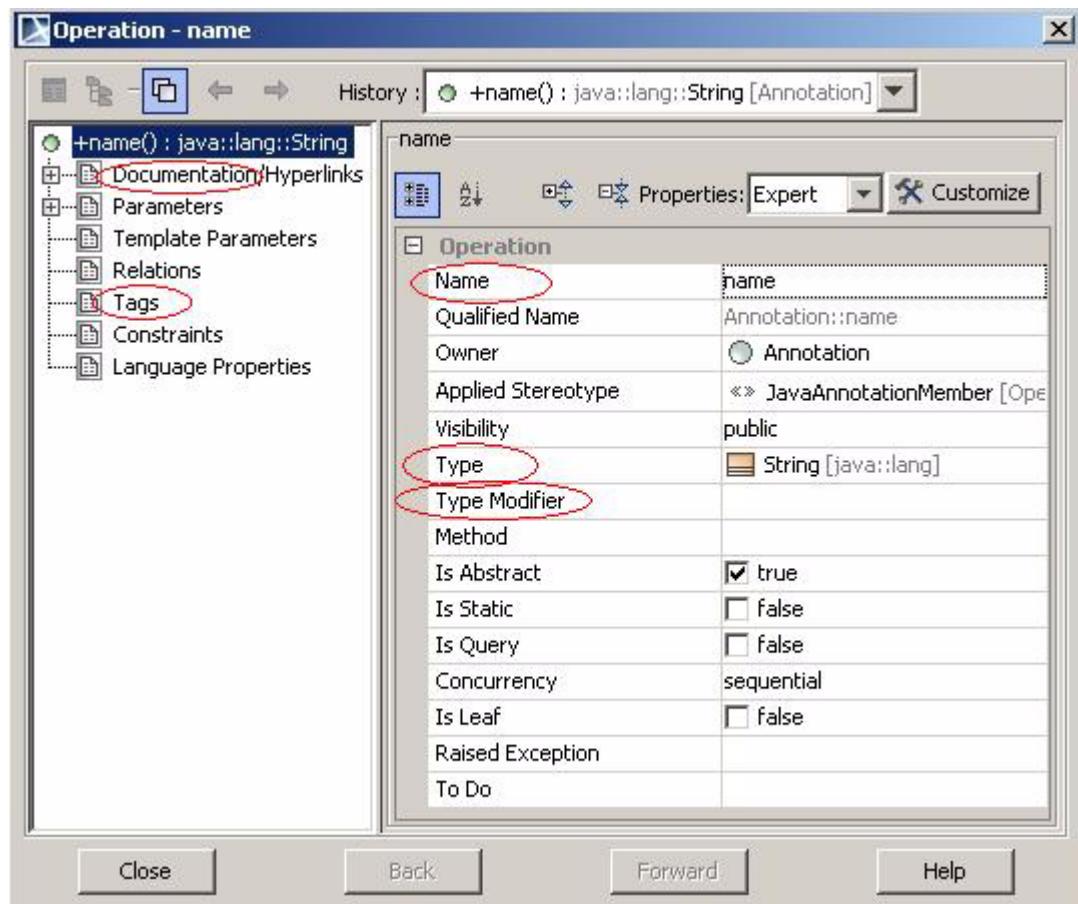


Figure 17 -- UML Operation, representing Java annotation member, specification dialog

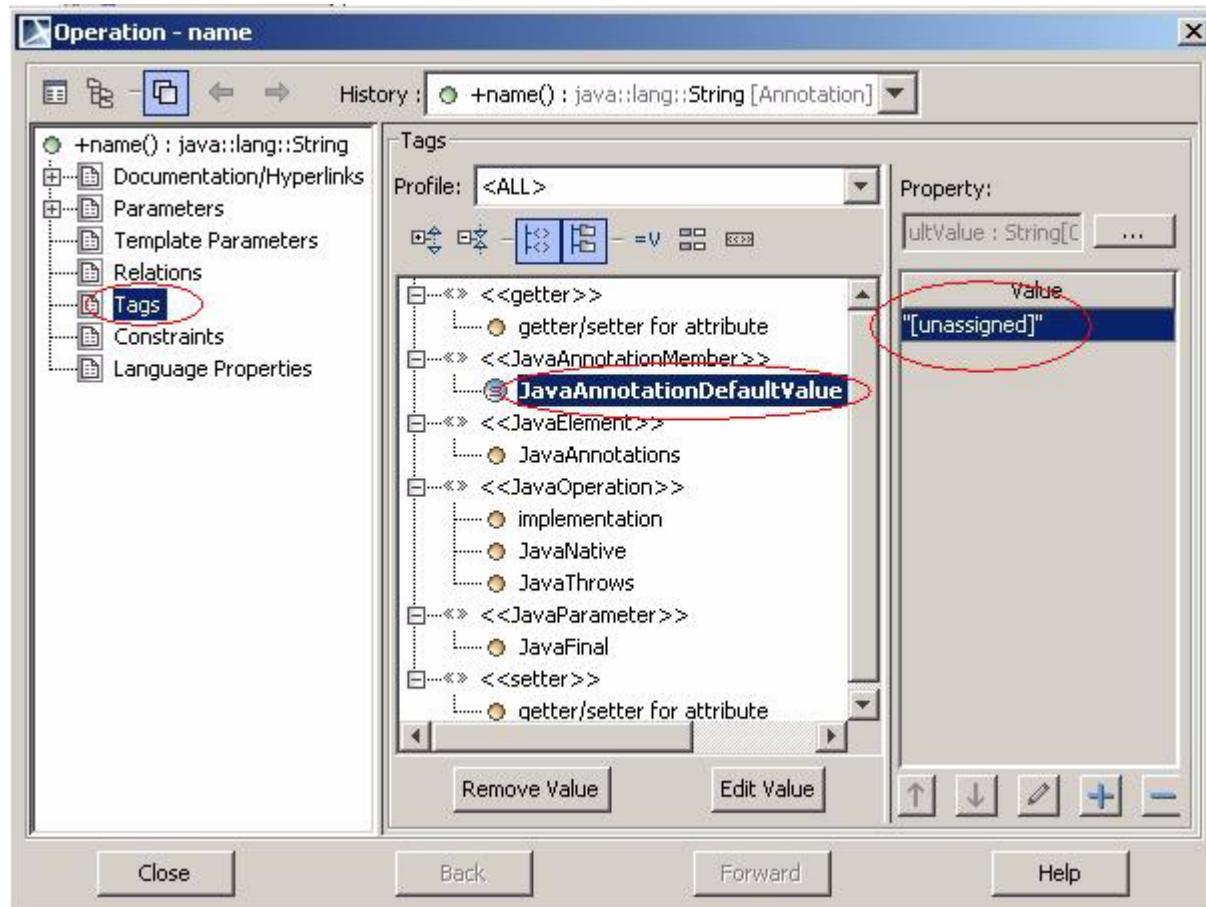


Figure 18 -- Default value set for UML Operation, representing Java annotation member

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

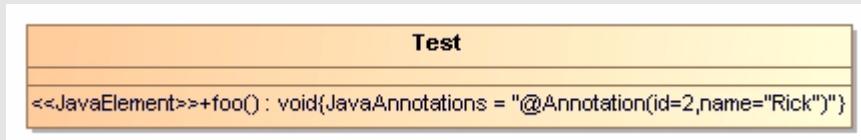


Figure 19 -- Java annotation usage

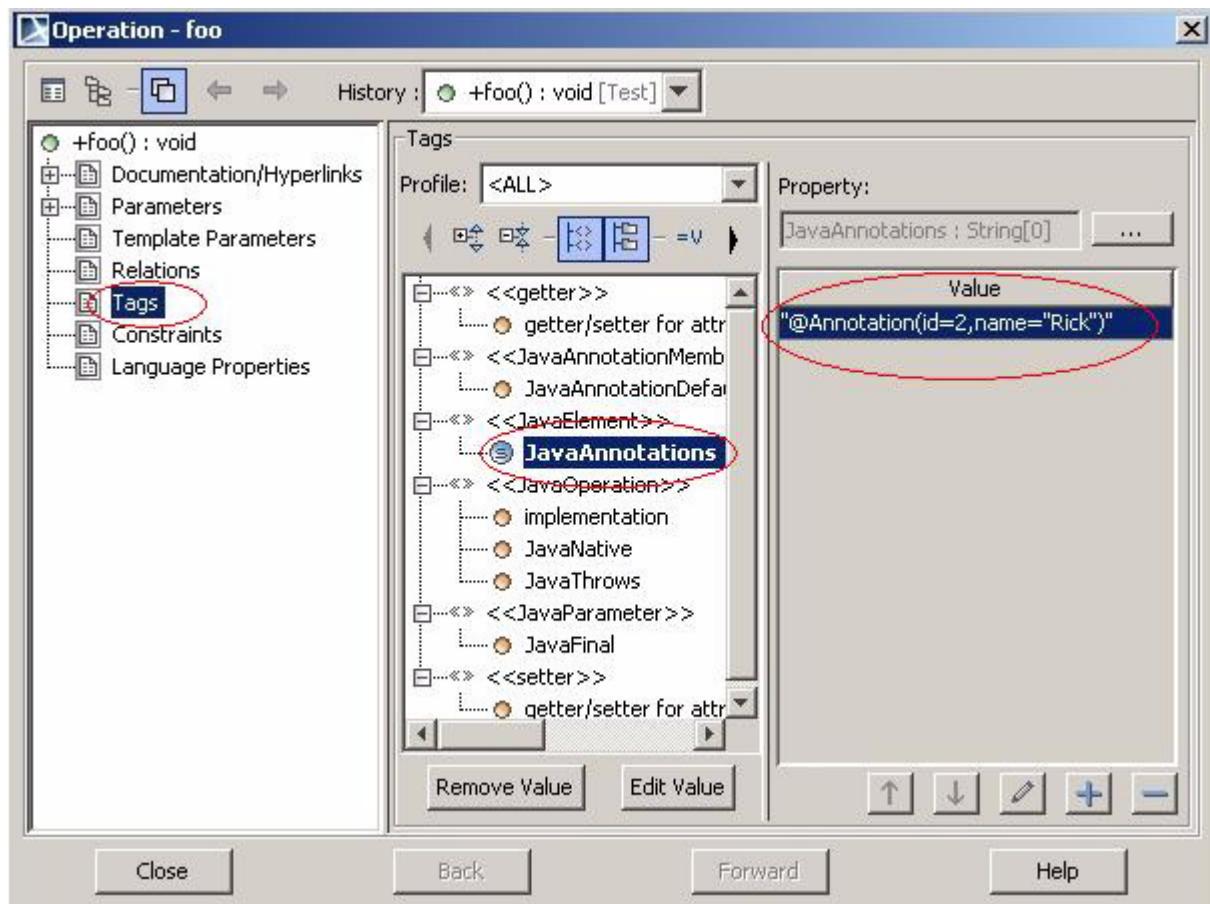


Figure 20 -- Annotation used on operation "foo"

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
--------------	-----------------------

Type variable declaration	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 2.0 Metamodel”
Type variable name	Is mapped to the UML Class name. This UML Class is “Parametered Element“
Type bounds	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

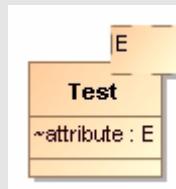


Figure 21 -- UML Class with template parameter *E*, representing Java type variable

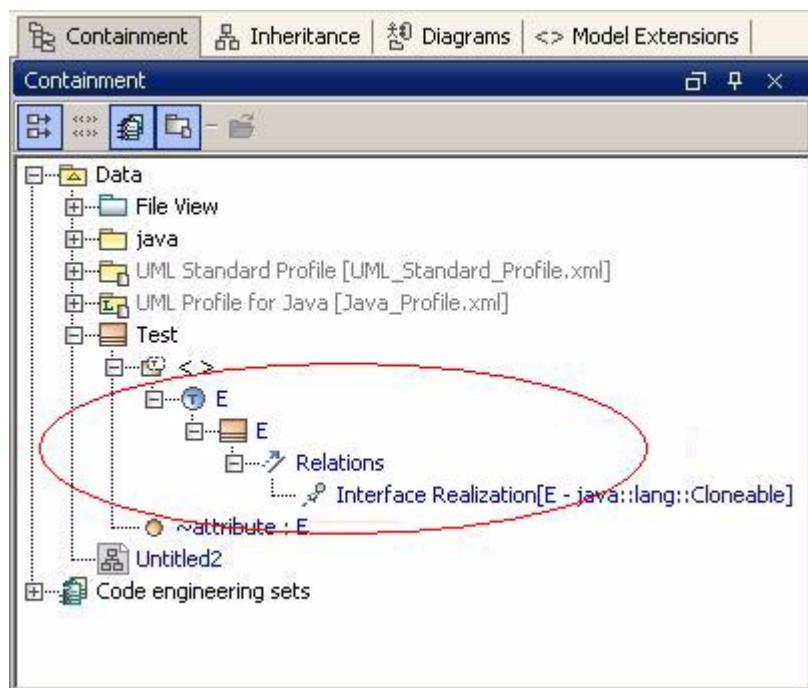


Figure 22 -- Marked UML Template Parameter, with residing UML Class named E (Parameterized Element)

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.

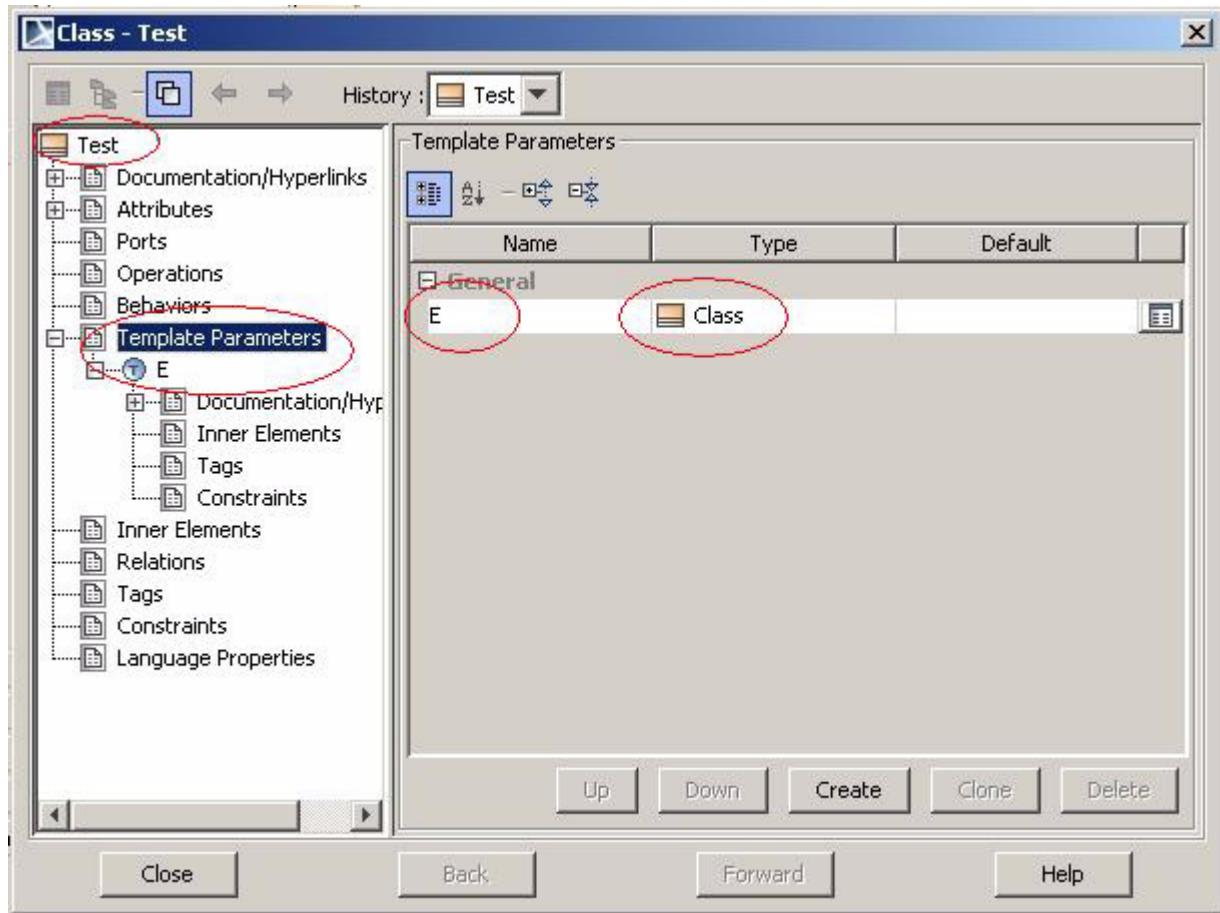


Figure 23 -- Template parameters in UML Class specification dialog

Parameterized Type

Parametrized 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 <<JavaTypeArgument>> 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`”.

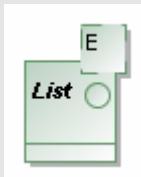


Figure 24 -- UML Interface representing “`java.util.List`” with template parameter

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

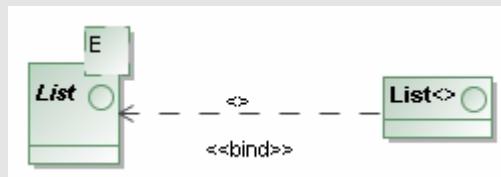


Figure 25 -- UML Template Binding between template class and bounded element

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

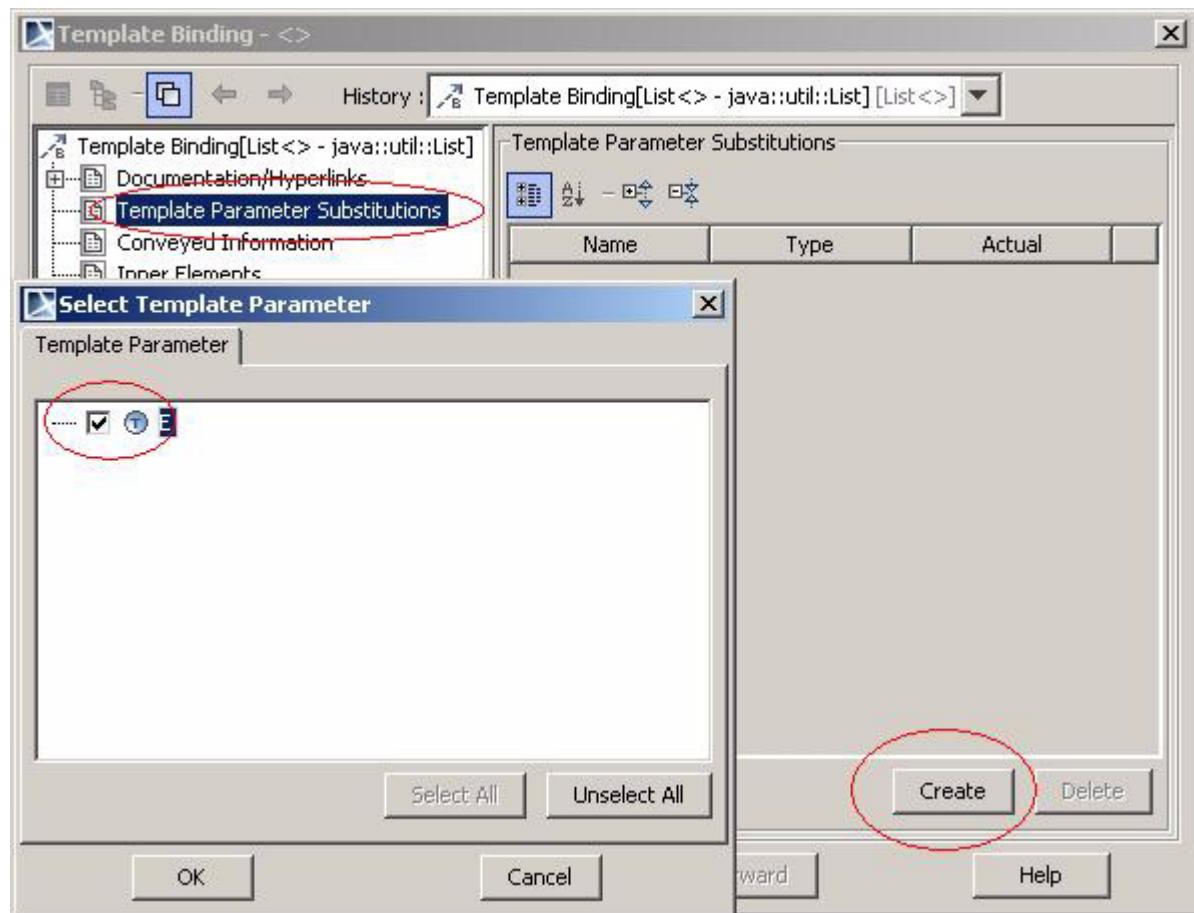


Figure 26 -- Creating UML Template Parameter Substitution

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>`

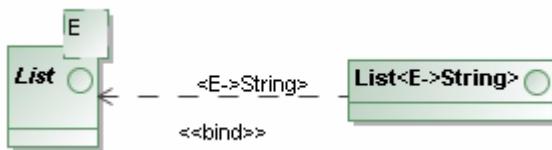


Figure 27 -- Created parameterized type for “java.util.List<String>”

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
{
}
```

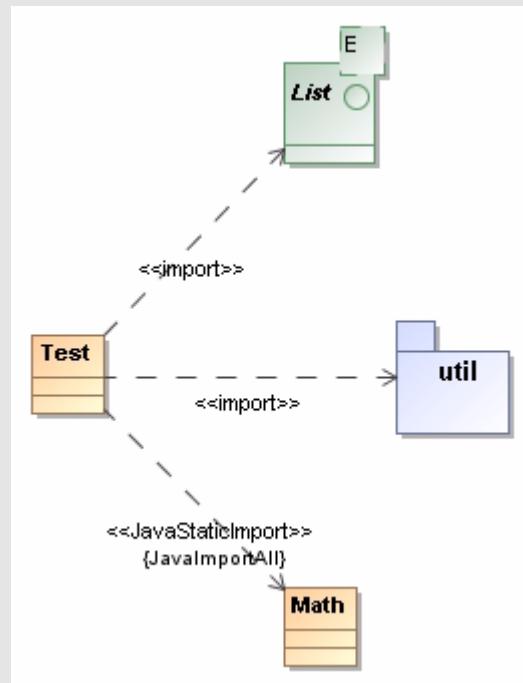
MagicDraw UML Model

Figure 28 -- 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 38 on page 66. Other options are common for all languages and are described in "Reverse", on page 18.

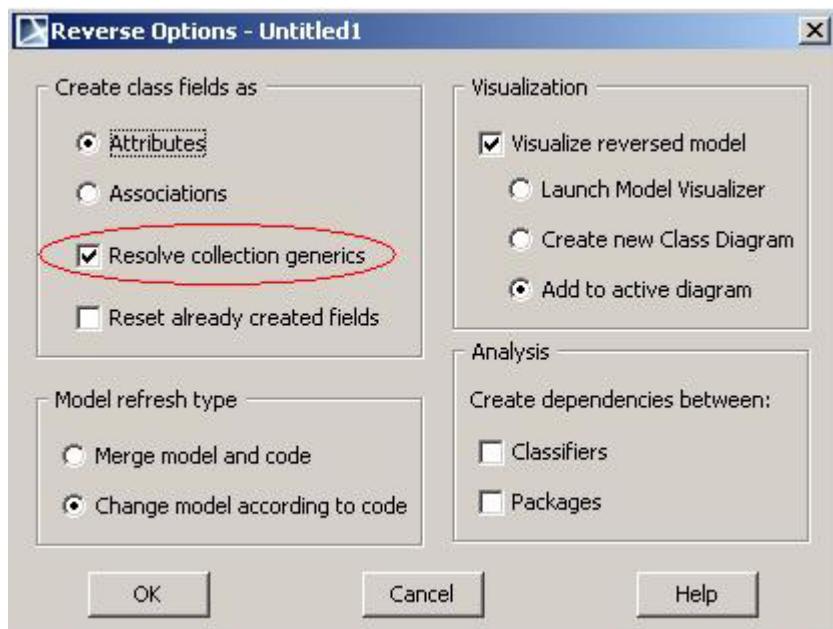


Figure 29 -- Java reverse properties

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

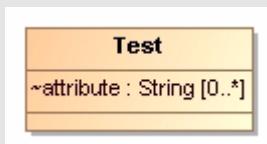


Figure 30 -- Attribute type, retrieved from collection

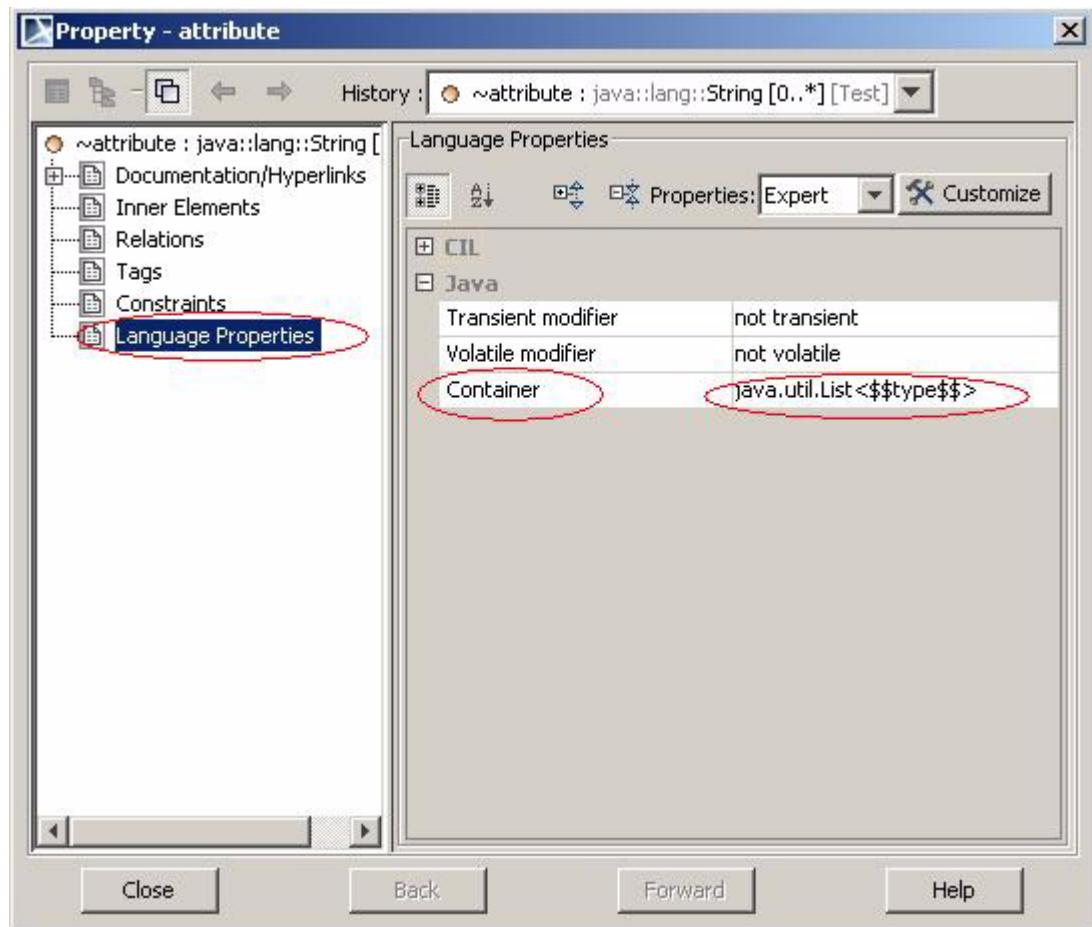


Figure 31 -- Collection type in UML Property specification

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.

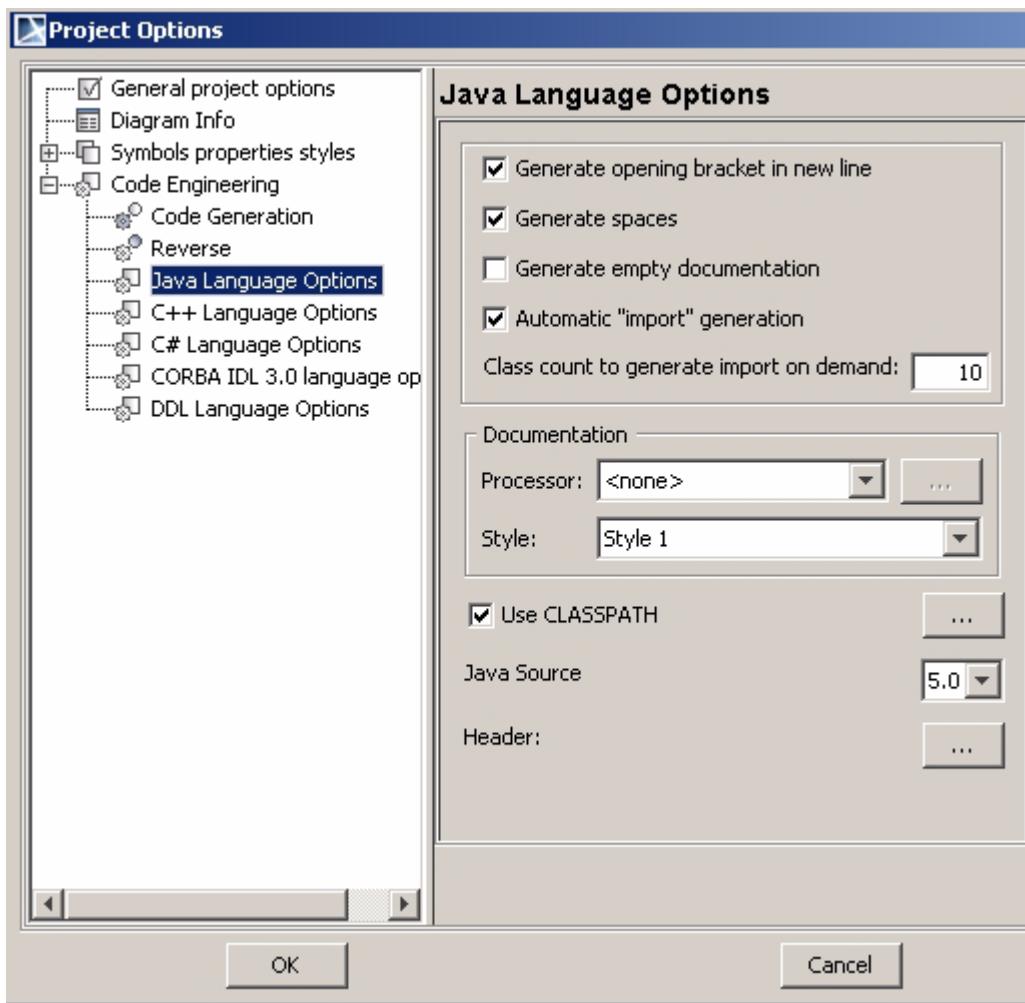


Figure 32 -- Java language options

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 64).

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 64.

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 pre-processed 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 a sequence diagram from Java method. It contains four steps that are described below.

STEP 1 Specify Name and Package.

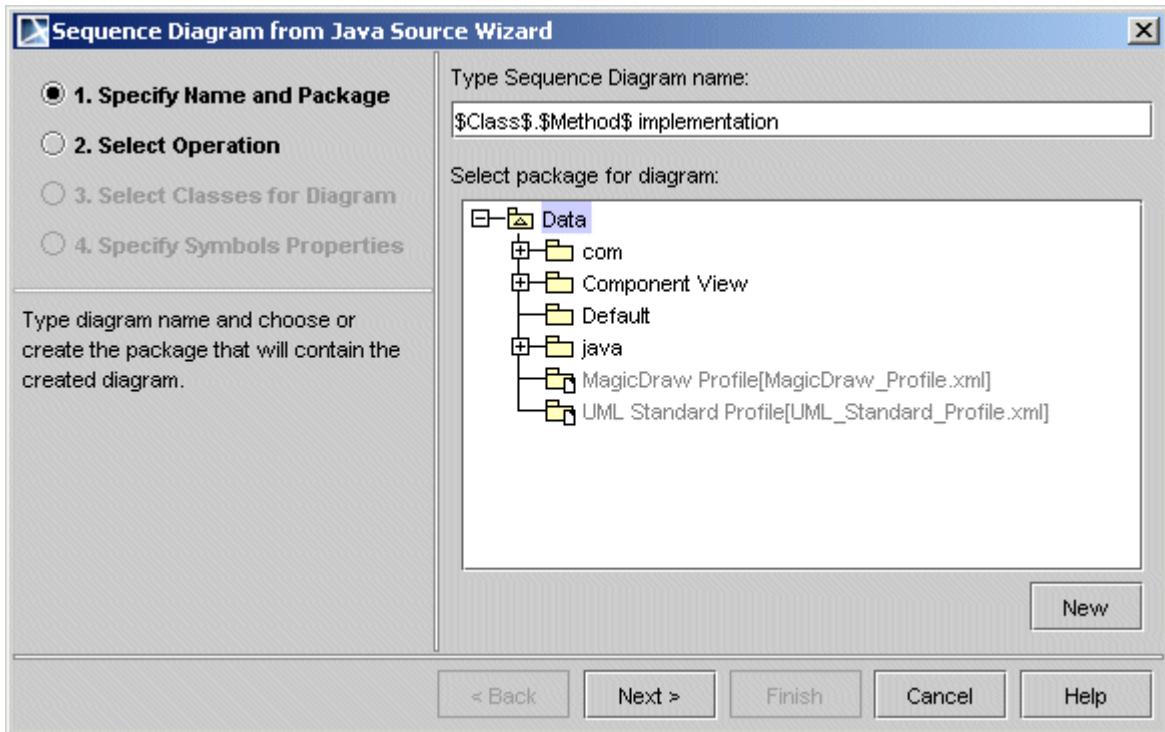


Figure 33 -- Sequence Diagram from Java Source Wizard

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.

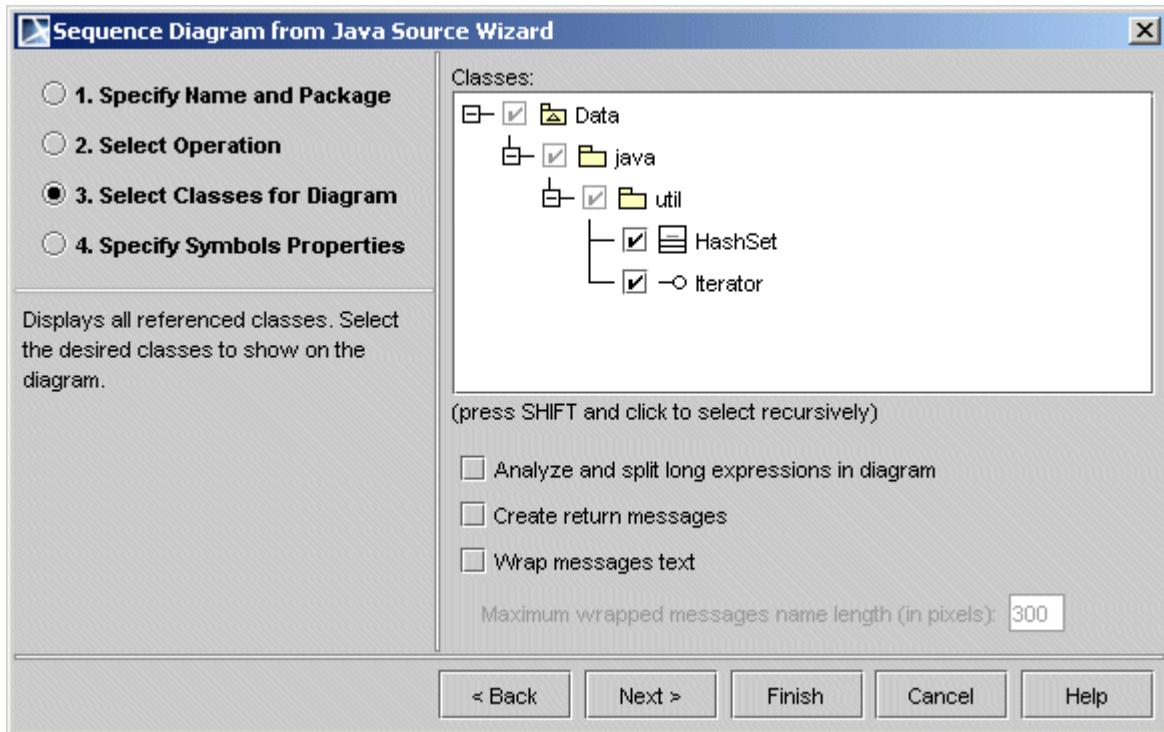
STEP 2 Select Operation



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

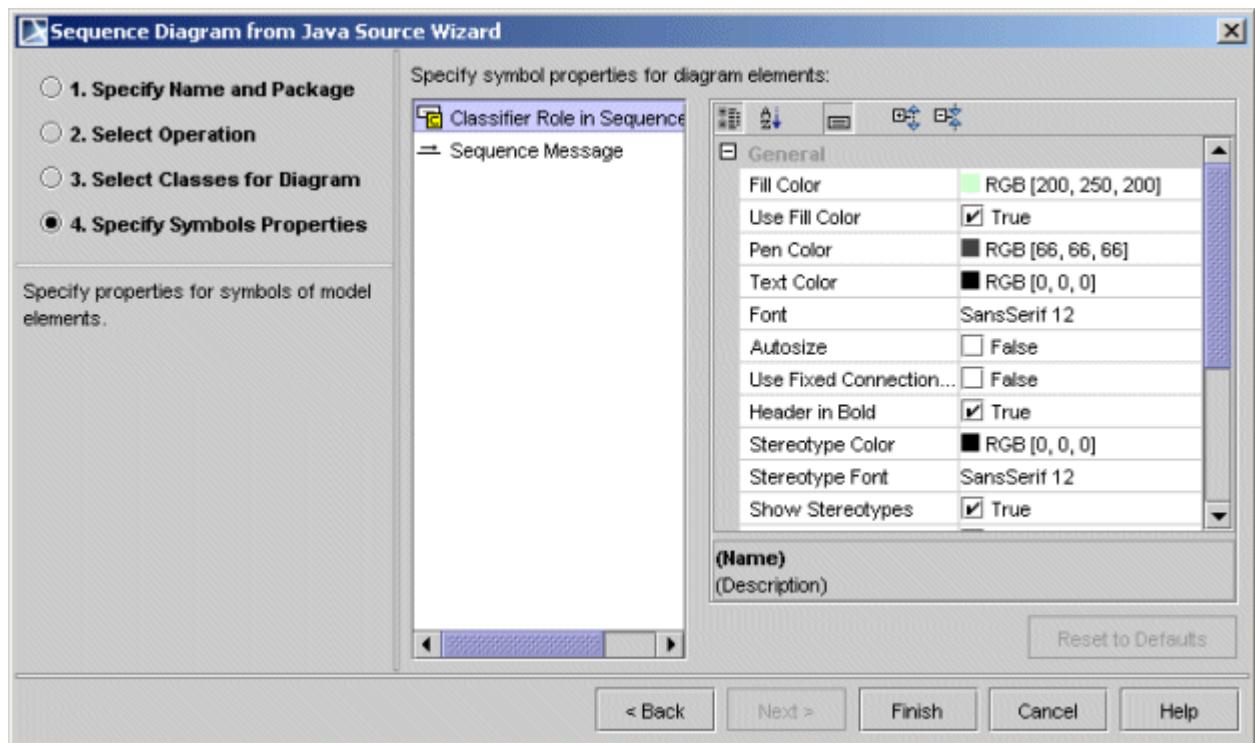


Figure 34 -- Sequence Diagram from Sequence Wizard. Specify Symbols Properties

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 type

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. Ex. short int or int short. Only the version 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 dataType 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 2.0 only Integer, Boolean, String and UnlimitedNatural are defined and they are not defined as datatype, but as primitive. In the current version of MD UnlimitedNatural is not defined in any profile and Integer and boolean (b is lowercase) are datatypes.

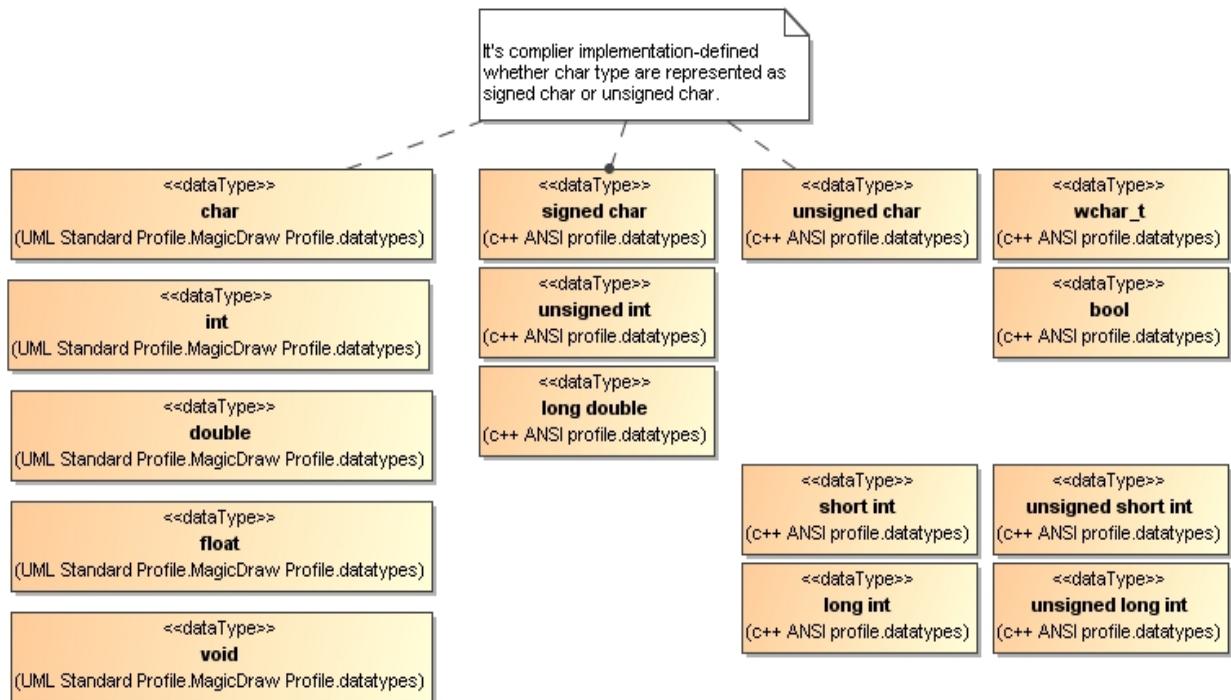


Figure 1 -- C++ datatype

char

char types are represented as signed char or unsigned char according to the compiler implementation. char do not have synonym.

signed char

signed char do not have synonym.

unsigned char

unsigned char do not have 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 do not have synonym.

long double

long double do not have synonym

float

float do not have synonym.

void

void do not have synonym.

bool

bool do not have synonym.

wchar_t

wchar_t do not have synonym.

Stereotype

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>> and <<C++TemplateParameter>> are invisible stereotypes and are used only to store C++ language properties. These stereotypes and their tag definition are used by the DSL framework.

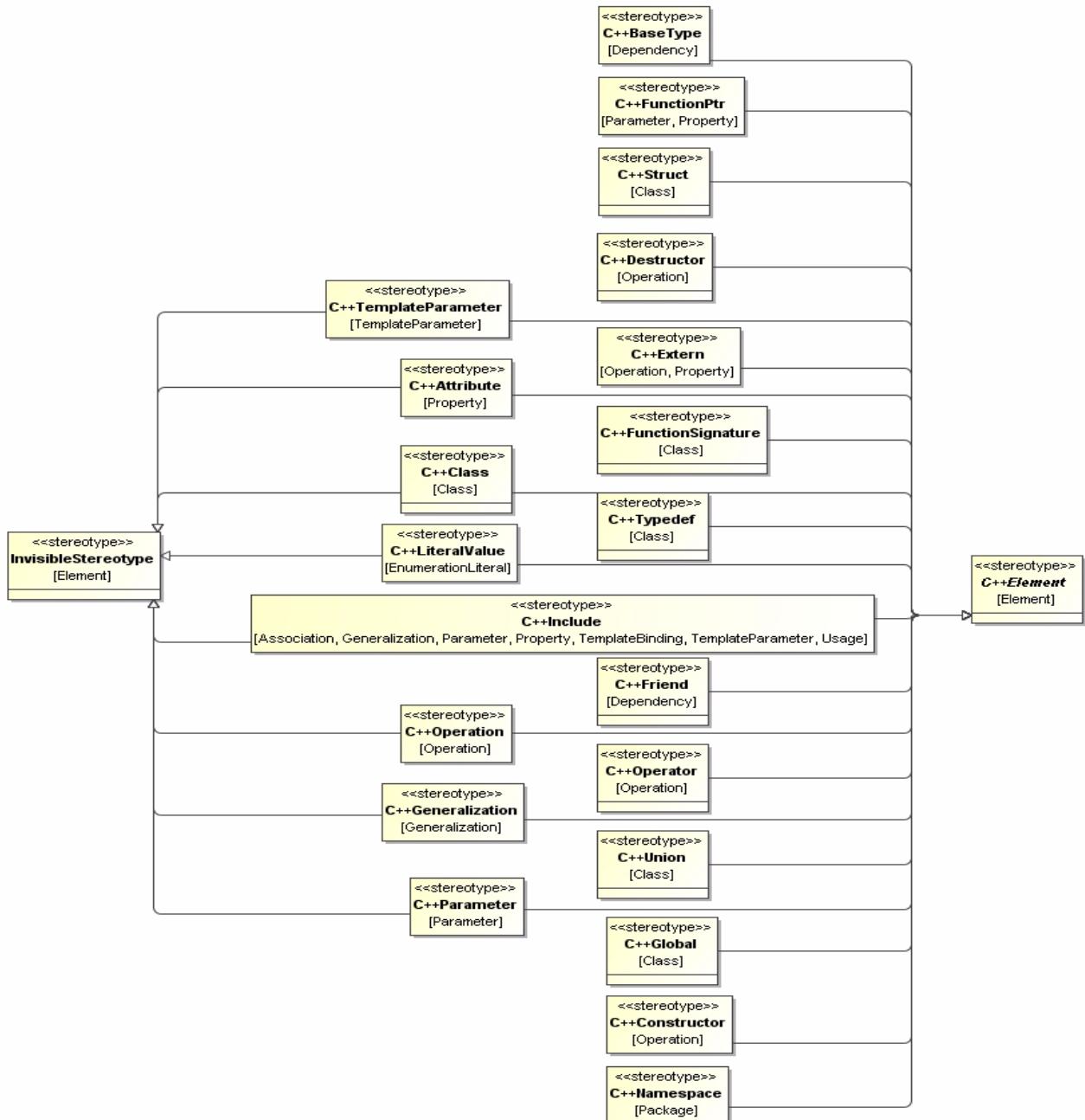


Figure 2 -- C++ Stereotype Hierarchy

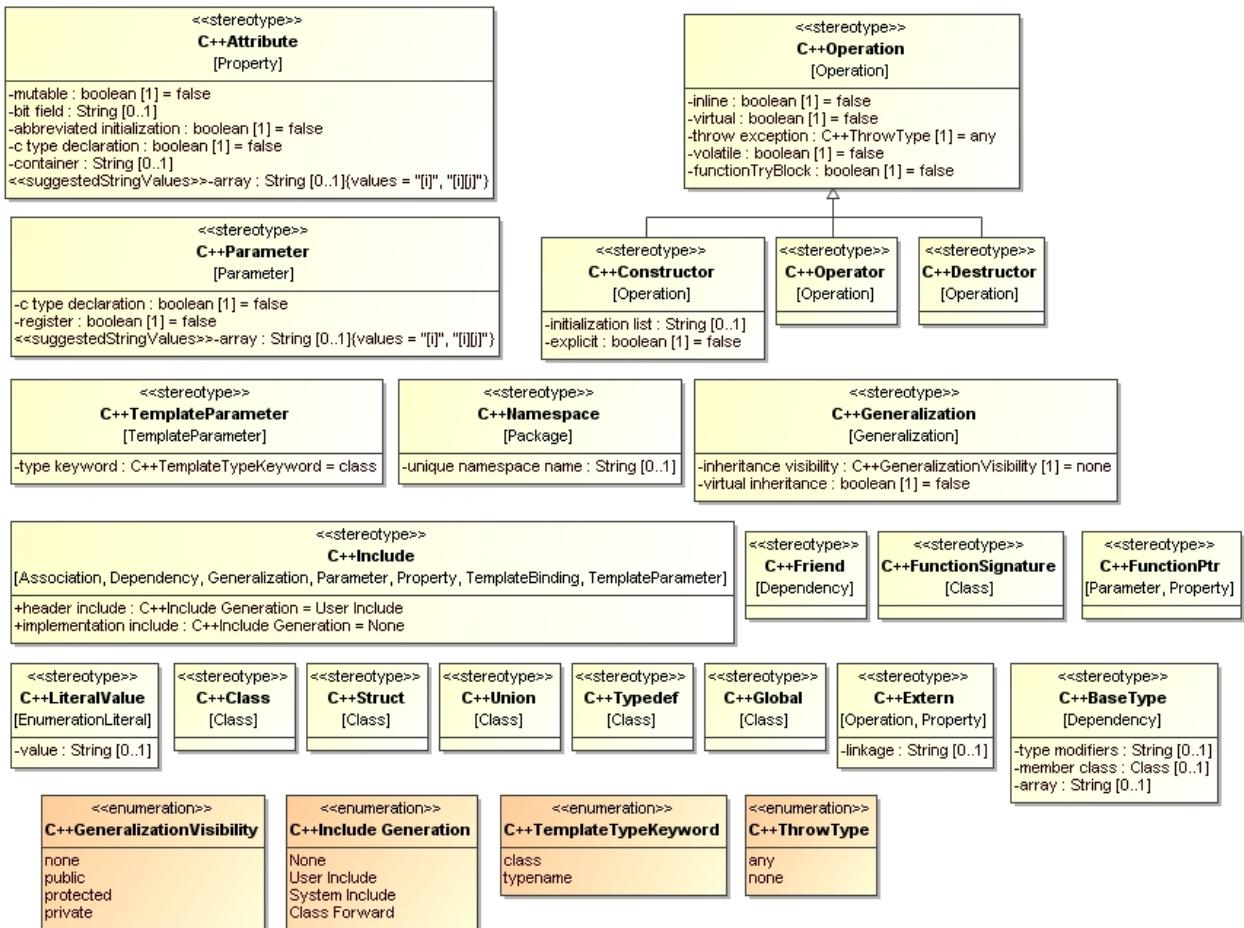


Figure 3 -- C++ Stereotype tag definitions

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

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 info.

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

C++Operator	Operation	operator function T& operator+(T& a); 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. class A { int x = 1; };) if defaultValue.size() > 0 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. <pre>int x(5);</pre> <p>Constraint <code>owner.stereotype->exists(name='C++Global')</code></p>
bit field	String[0..1]	Bit field declaration <pre>int x:2;</pre> <p>Constraint: Only valid for member function <code>stereotype->select(name='C++Global')</code> <code>->isEmpty()</code></p>
c type declaration	boolean[1]=false	Declare attribute's type in C style <p>C style: <code>enum Day x;</code></p> <p>C++ style: <code>Day x;</code></p>
container	String[0..1]	container of the attribute. \$ character is replaced by the attribute type. <pre>vector<\$> x;</pre>
mutable	boolean[1]=false	Attribute mutable modifier. <pre>mutable int x;</pre> <p>Constraint: Only valid for member function <code>stereotype->select(name='C++Global')</code> <code>->isEmpty()</code></p>
array	String[0..1]	C++ array definition <pre>int x[2][2];</pre>

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 (client.oclIsTypeOf(Class) or client.oclIsTypeOf(Operation)) and supplier.oclIsTypeOf(Class)

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 feature->isEmpty() 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 115 for more info.

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

C++Namespace

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

name	Meta class	Constraints
C++Namespace	package	
Tag	Type	Description
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.

Name	Meta class	Constraints
C++Constructor	operation	name = owner.name
Tag	Type	Description
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.

Name	Meta class	Constraints
C++Destructor	operation	name = “~”+owner.name

C++Extern

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

Name	Meta class	Constraints
C++Extern	property	owner.stereotype->exists(name='C++Global')
Tag	Type	Description
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.

Name	Meta class	Constraints
C++FunctionPtr	Parameter, Property	
Tag	Type	Description
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 can't have property. 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.

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 None User Include System Include Class Forward
implementation include	String	The value of tag is one of the following None User Include System Include Class Forward

Mapping

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

Class

C++ class map to a UML class

Code	MD-UML
class A { };	

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>	<pre> classDiagram class BaseClass class OtherBaseClass class SuperClass SuperClass < -- BaseClass SuperClass < -- OtherBaseClass </pre>

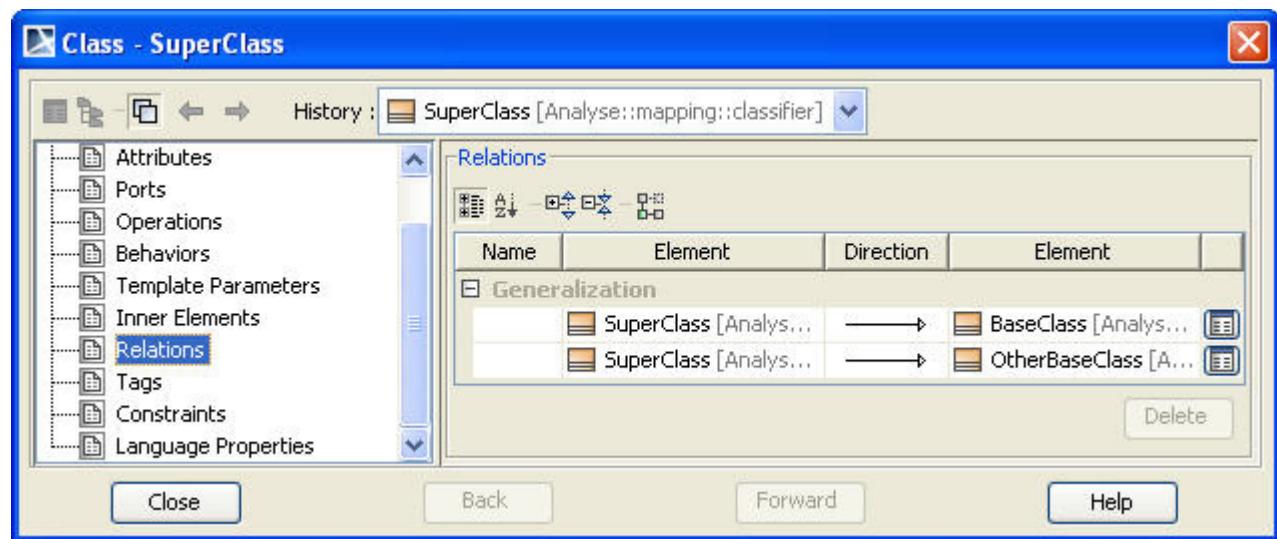


Figure 4 -- SuperClass Generalization relationship

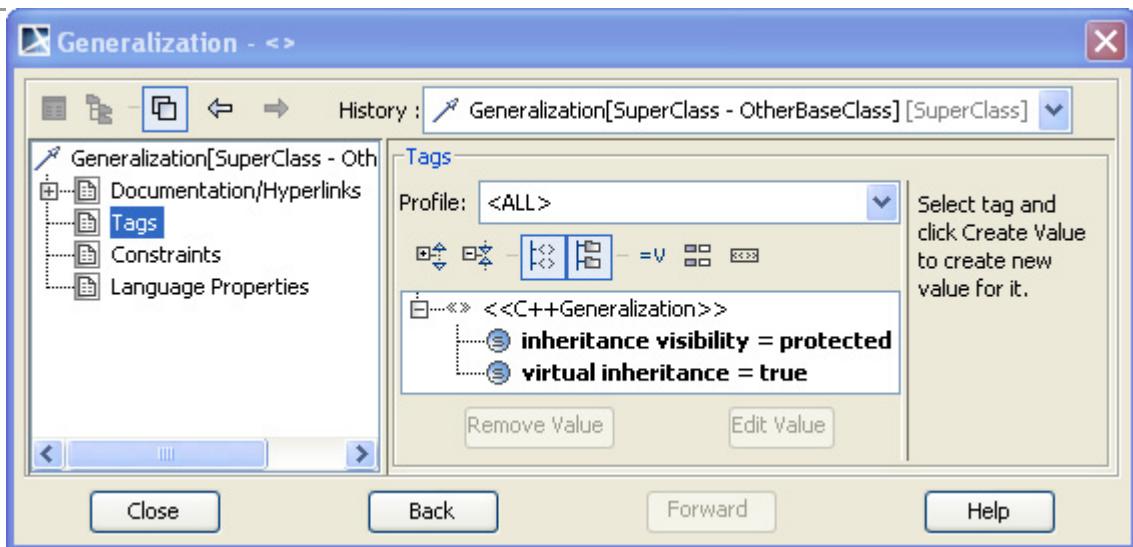


Figure 5 -- Generalization language properties

Class member variable

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

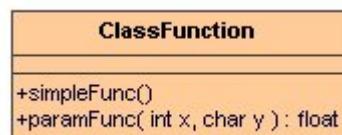
Code	MD-UML
<pre>class ClassVariable { int age; char* name; };</pre>	<pre>ClassVariables --age : int --name : char*"</pre>

Class member function

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

Code	MD-UML

```
class ClassFunction {  
public:  
    void simpleFunc();  
    float paramFunc(int x, char y);  
};
```



Class constructor/destructor

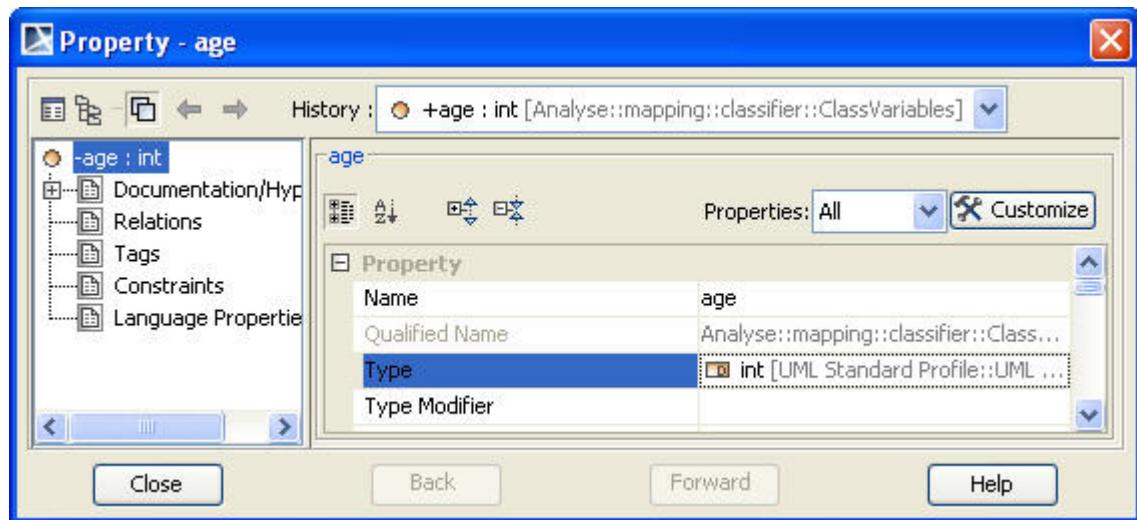
C++ class constructor and destructor are mapped to UML operation with <<C++Constructor>> stereotype and <<C++Destructor>> stereotype. See [C++Constructor](#) and [C++Destructor](#) for more info.

Code	MD-UML
class ConstructClass { public: ConstructClass(); ~ConstructClass(); }	<p>A UML class diagram showing a class named "ConstructClass". It has two operations: "<<C++Constructor>>+ConstructClass()" and "<<C++Destructor>>+~ConstructClass()".</p>

Variable

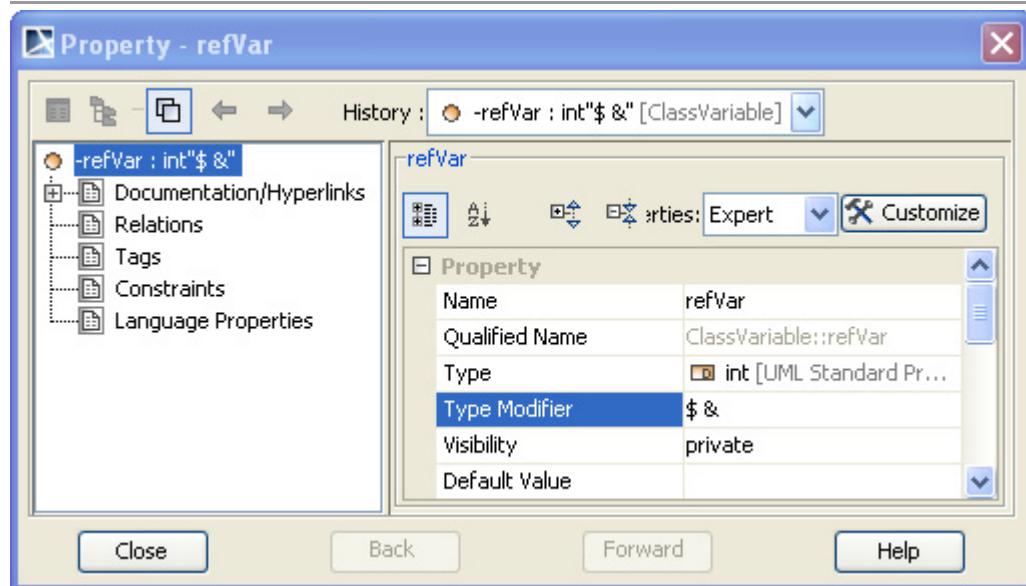
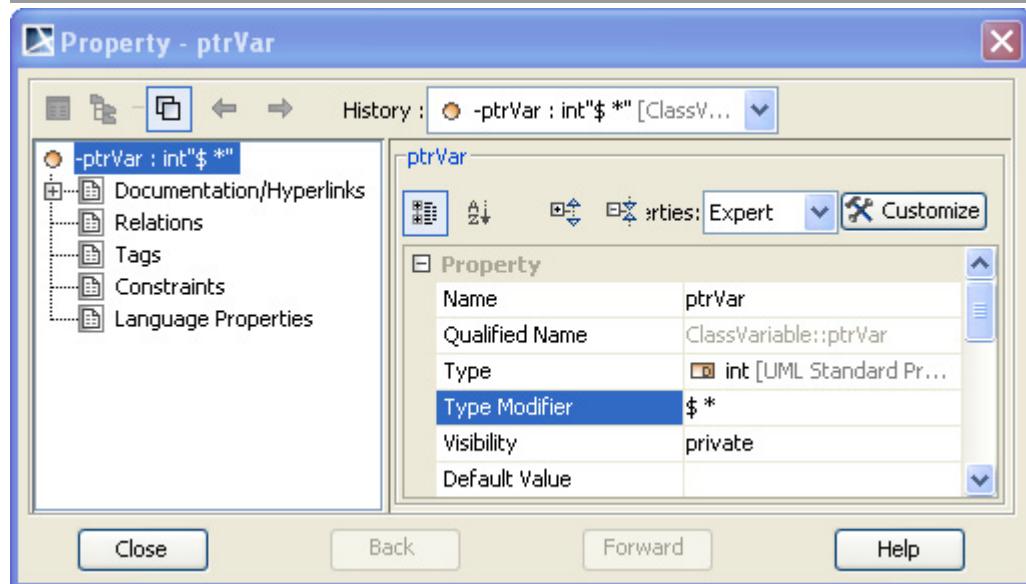
C++ variable is mapped to UML attribute, the variable type is mapped to the attribute type.

Code	MD-UML
int age;	



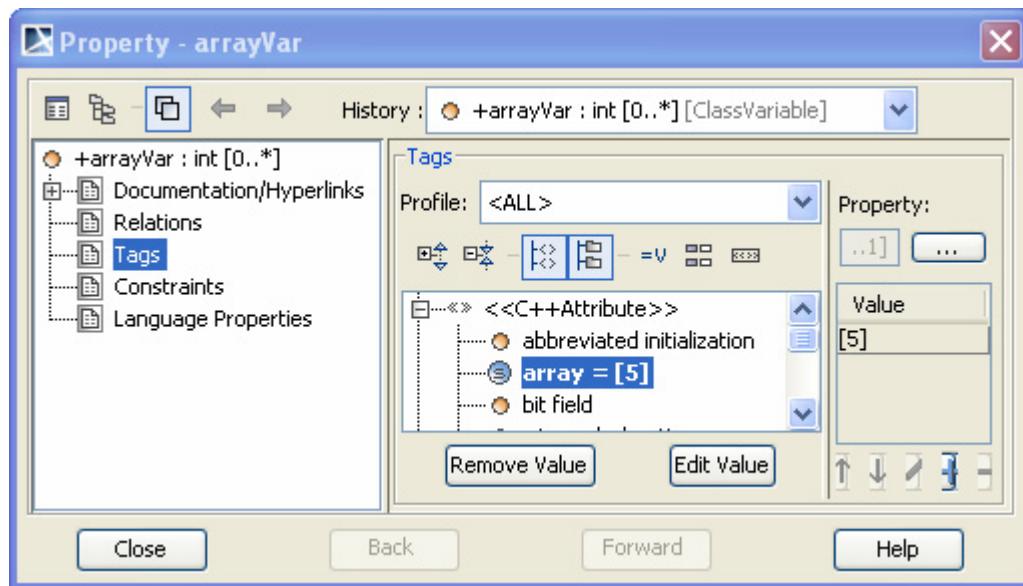
C++ type pointer/reference is mapped to Type Modifier property of the attribute. \$ character is replaced by the type name.

Code	MD-UML
int* ptrVar; int& refVar	



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
int arrayVar[5];	

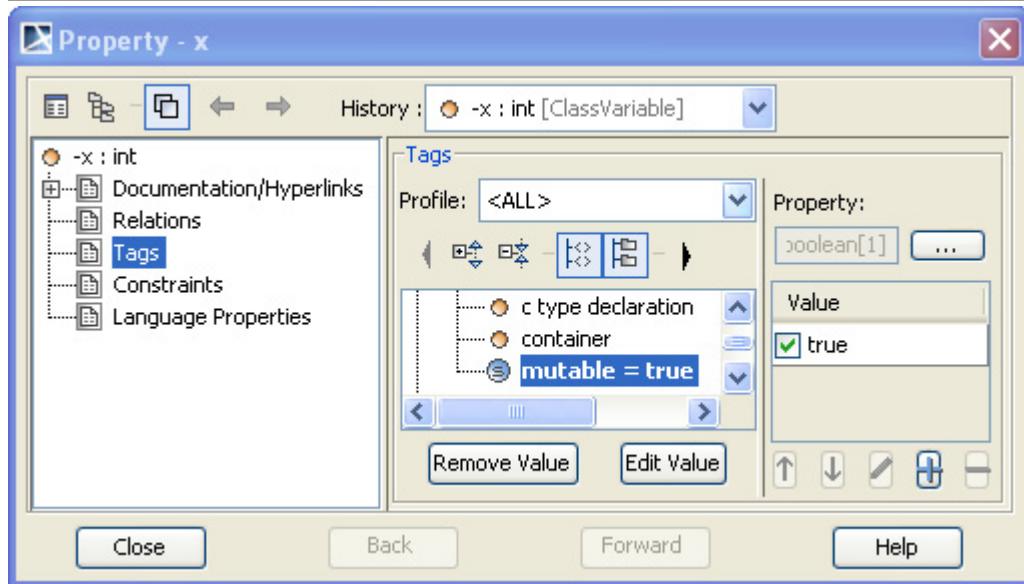


Variable modifiers

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

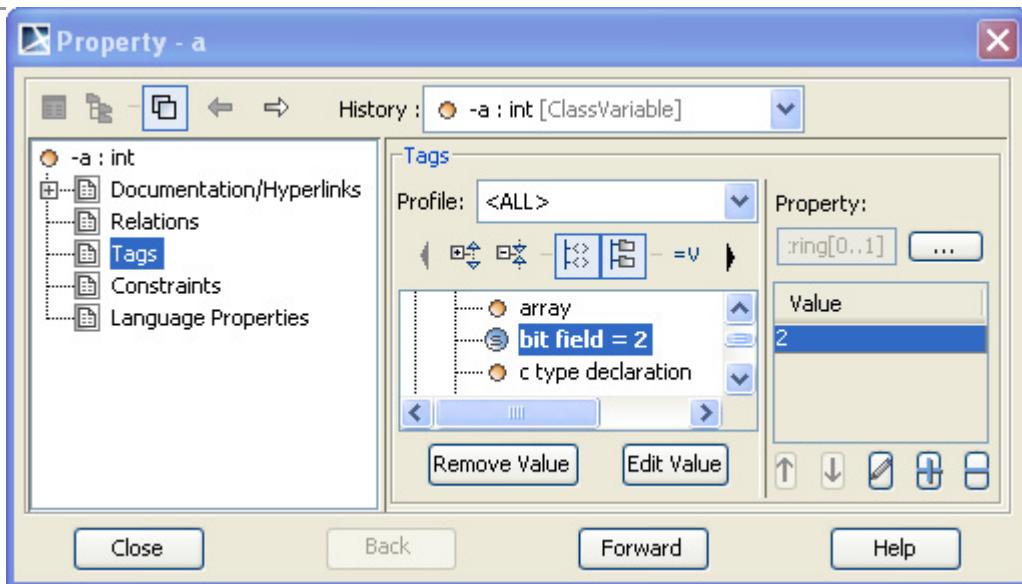
Constraint: only member variable can be mutable (global variable can not).

Code	MD-UML
mutable int x;	



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

Code	MD-UML
<pre>struct BitStruct { int a:2; };</pre>	<p><<C++Struct>></p> <p>BitStruct</p> <p>+a : int</p>



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
extern int externVar;	<pre><<C++Global>> <<C++Extern>>+externVar : int</pre>

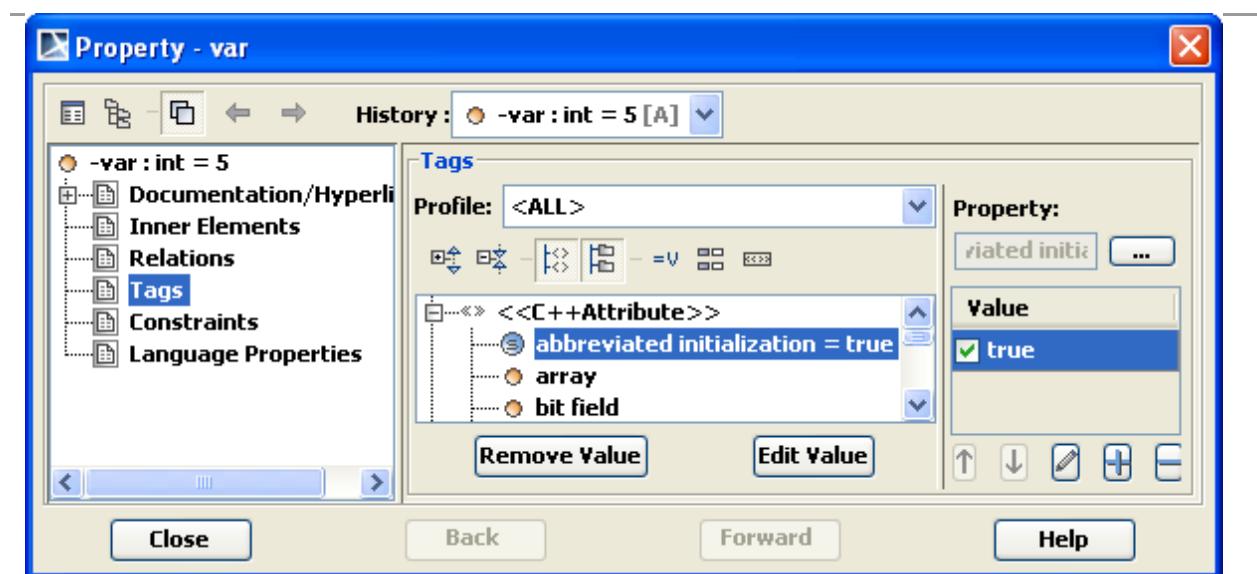
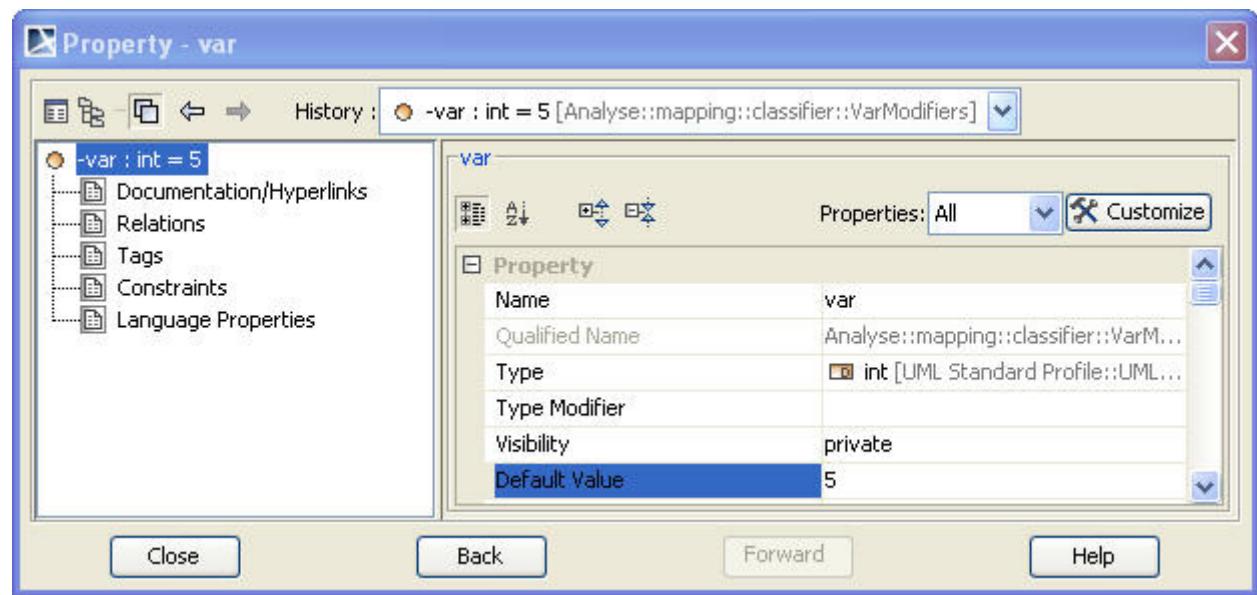
Variable default value

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

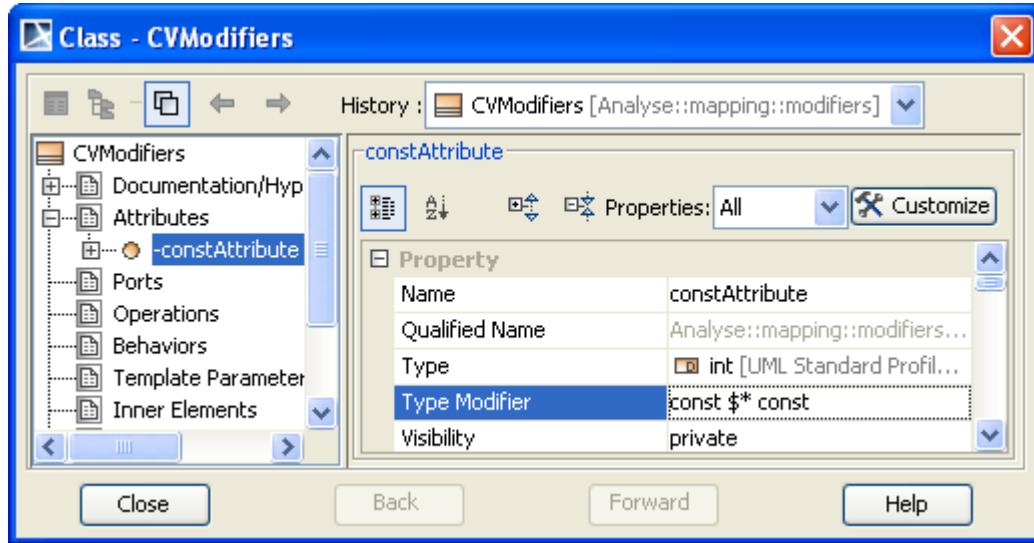
C++ const and volatile modifiers for attribute/function parameter are mapped to *Type Modifiers* properties .

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>	



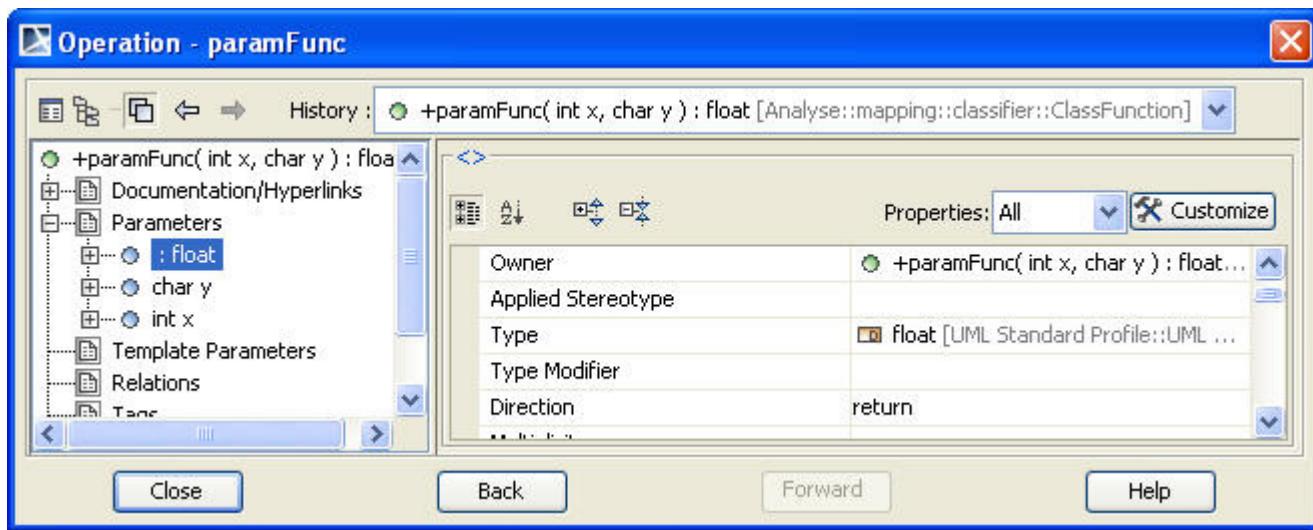
Function

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.

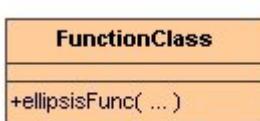
Pointer, reference and array type of parameter are mapped to property *Type Modifier* of parameter. See [Variable modifiers](#) for more info.

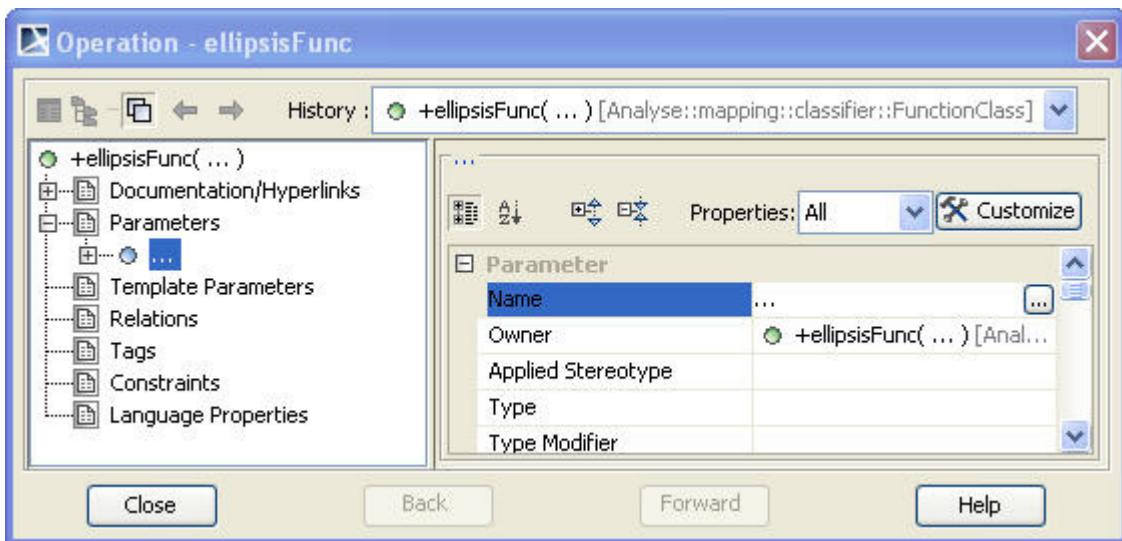
Code	MD-UML
float paramFunc(int x, char x);	



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.

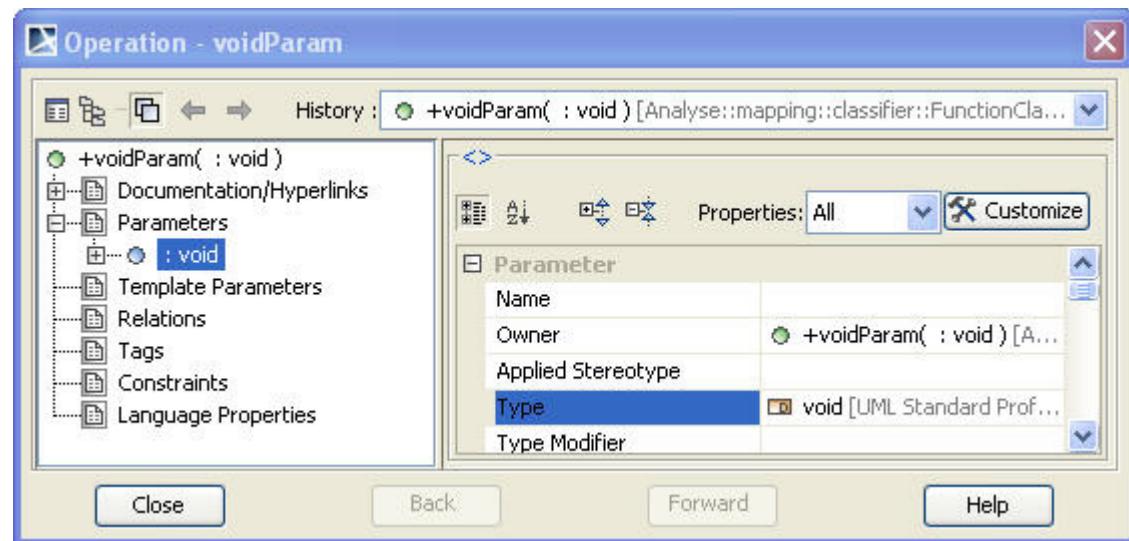
Code	MD-UML
Class FunctionClass { public: EllipsisFunc(...); };	



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>	<p>FunctionClass</p> <pre>+voidParam(: void): void ...</pre>

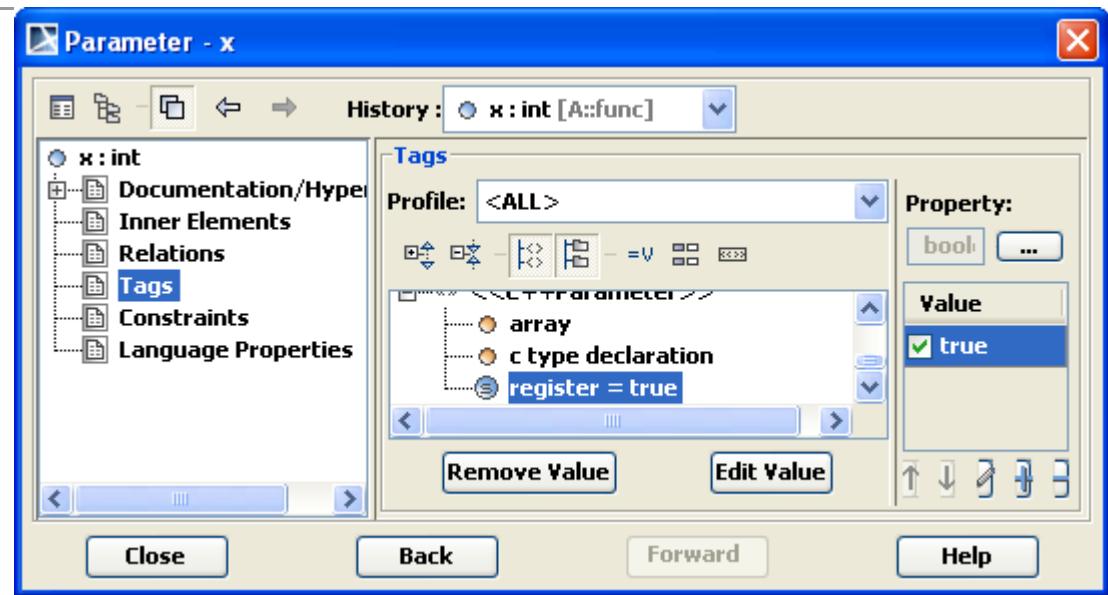


Register parameter

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>	<p>RegisterParamClass</p> <p>+registerParam(int x)</p>



Function modifiers

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

Virtual function is mapped to *Virtual modifier* property.

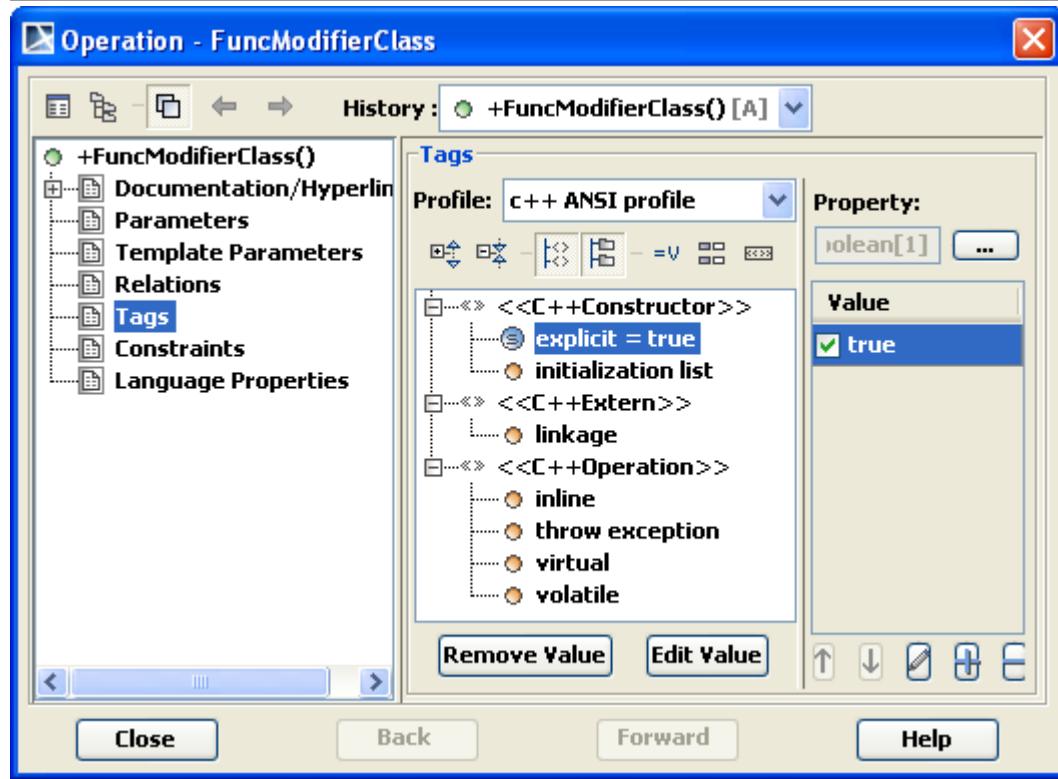
Inline function is mapped to *Inline modifier* property.

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>	



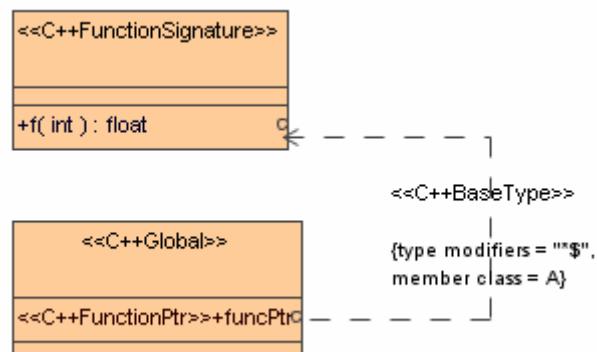
Function pointer

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
------	--------

```
float (A*funcPtr) (int);
```



Function operator

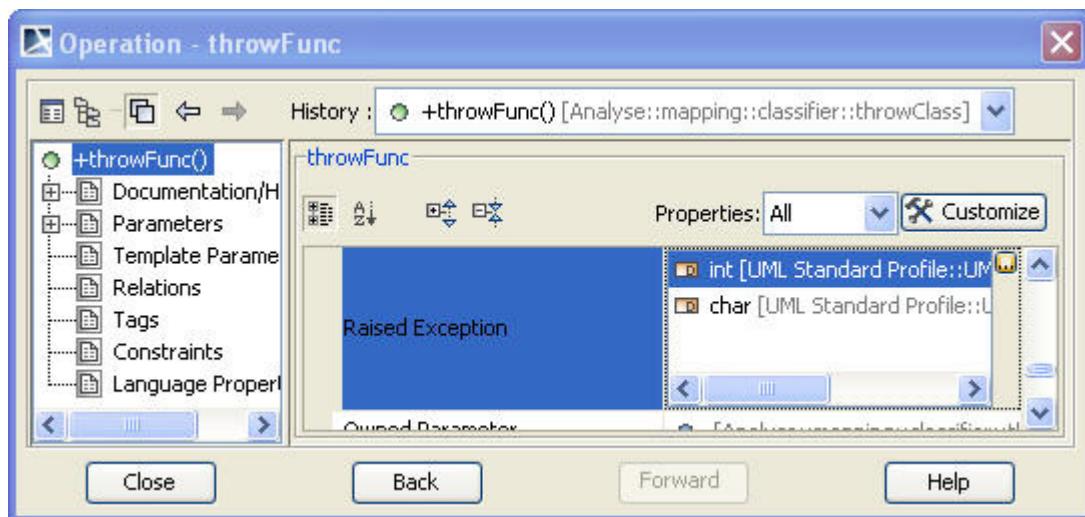
C++ function operator is mapped to normal function with the C++ operator name mapped to UML operation name. See [C++Operator](#) for more info.

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



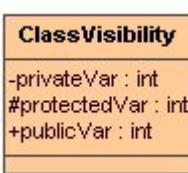
Visibility

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.

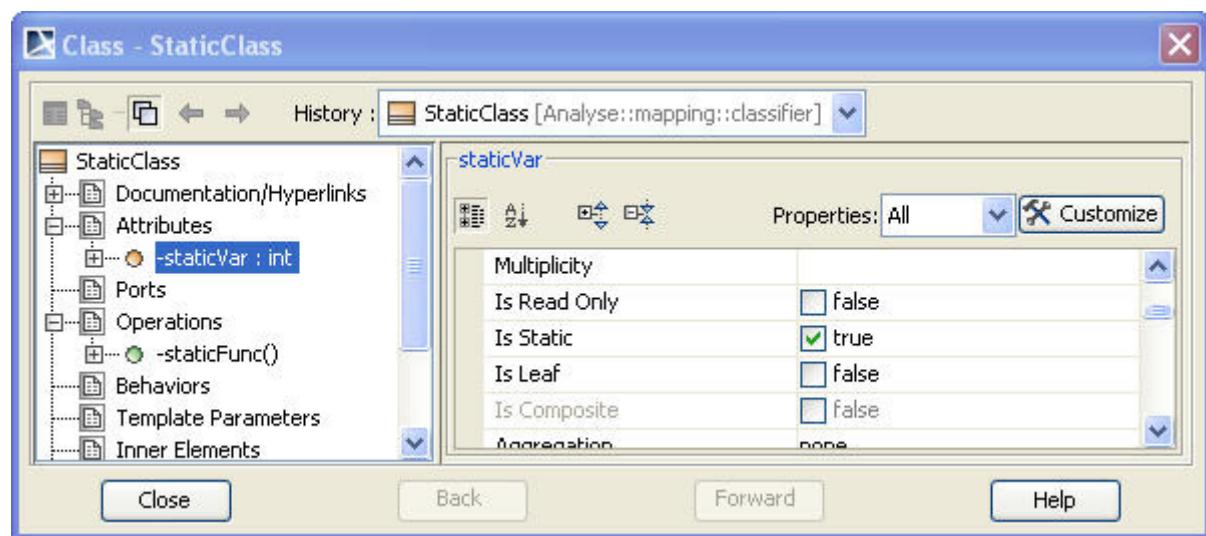
Variables and functions outside a class/struct/union are public.

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



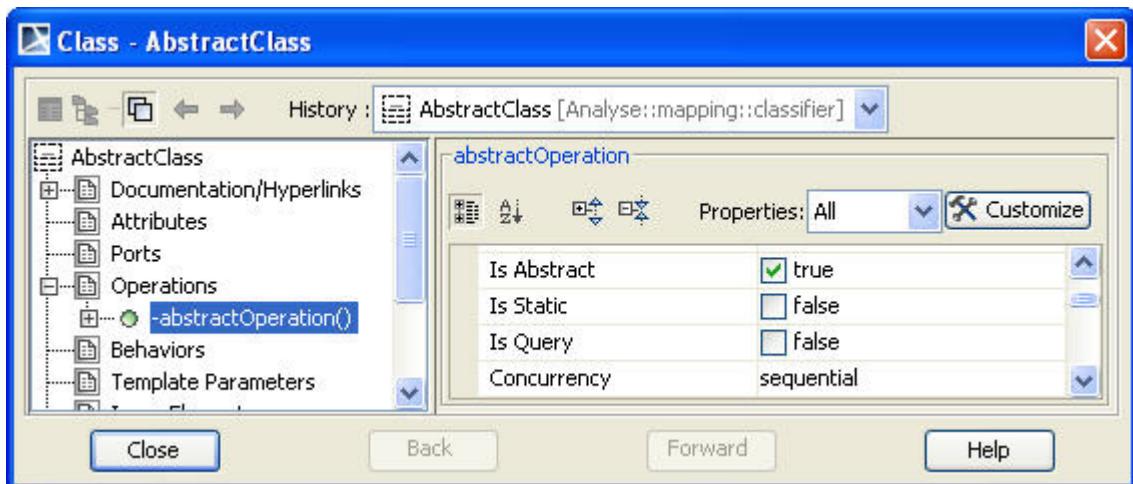
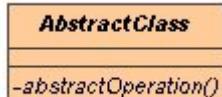
Pure virtual function and abstract class

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

```
class AbstractClass {
    virtual abstractOperation()=0;
};
```



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>	<p>The MD-UML diagram shows a dependency relationship. On the left, there is a global function represented by a box labeled "<<C++Global>>" with an operation "+friendFunc()". On the right, there is a class box labeled "ClassB". A dependency arrow originates from the global function box and points to the class box, with the stereotype "<<C++Friend>>" placed between them.</p>

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>	<pre> classDiagram ClassD "func(c : ClassC) : void" --> <<C++Friend>> ClassC "-a : int" </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>	<pre> classDiagram ClassA --> <<C++Friend>> FriendClass </pre>

Struct

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

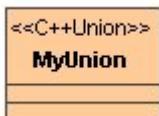
NOTE The current version of MD use class's language property "Class Key"

Code	MD-UML
<pre>struct MyStruc {</pre>	<pre> classDiagram <<C++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"

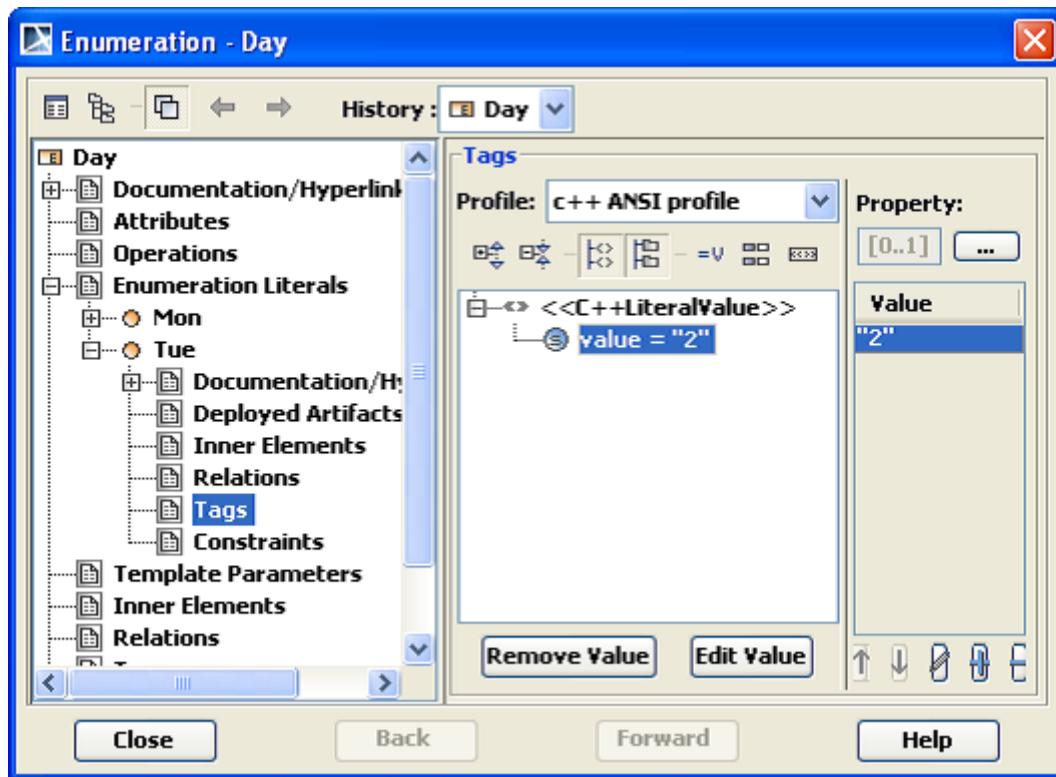
Code	MD-UML
<pre>union MyUnion { };</pre>	 A UML class diagram showing a class named "MyUnion". The class has an orange header labeled "<<C++Union>>" and a blue body labeled "MyUnion". There are no compartments or associations shown.

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>	 A UML enumeration diagram showing an enumeration named "Day". The enumeration has an orange header labeled "<<enumeration>>" and a blue body labeled "Day". It contains two members: "Mon" and "Tue", which are represented by yellow rectangles below the body.



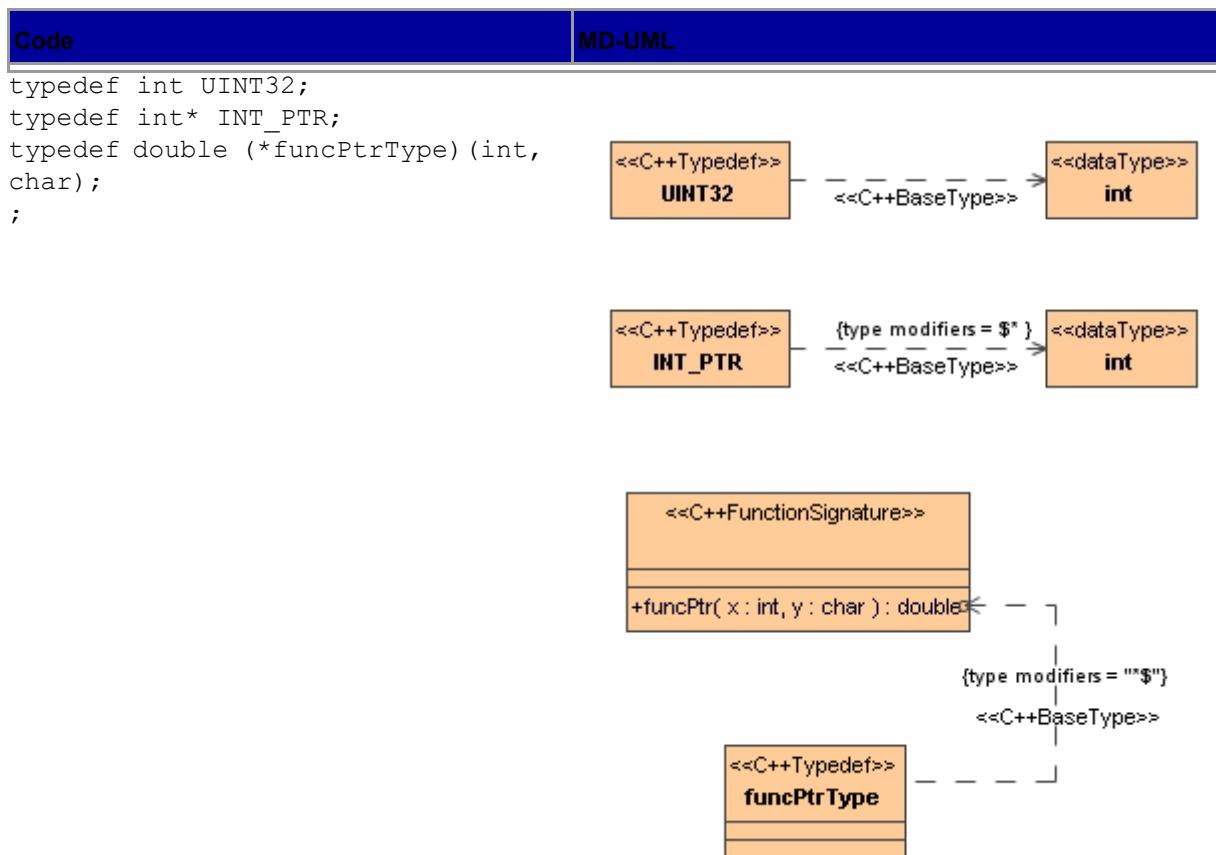
Typedef

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.

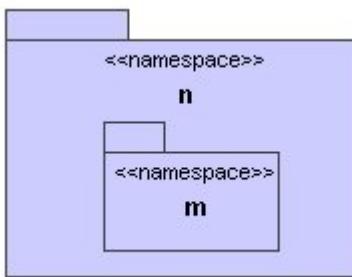
See C++Typedef for more info.



Namespace

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

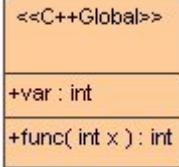
Unnamed namespace is named unnamed+index number of unnamed namespace (start at 1), and *unique namespace name* tag is set to the source file path+:+index number of unnamed namespace (start at 0).

Code	MD-UML
<pre>namespace n { namespace m { } }</pre>	

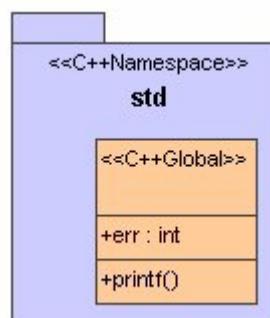
1 Global functions and variables

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
<pre>int var; int func(int x);</pre>	

```
namespace std {
    int err;
    void printf();
}
```



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

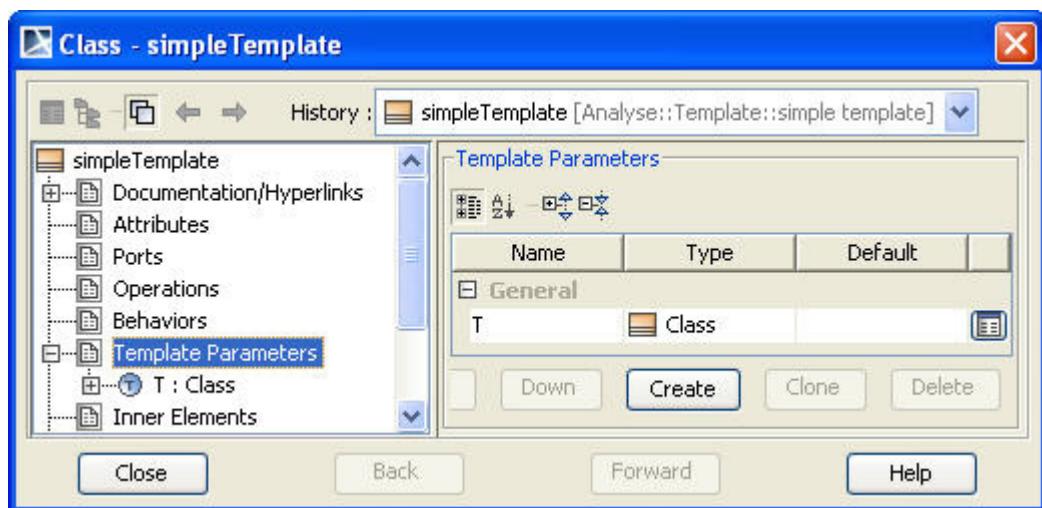
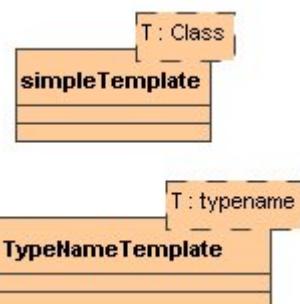
C++ template class is mapped to UML class with template parameters properties added.

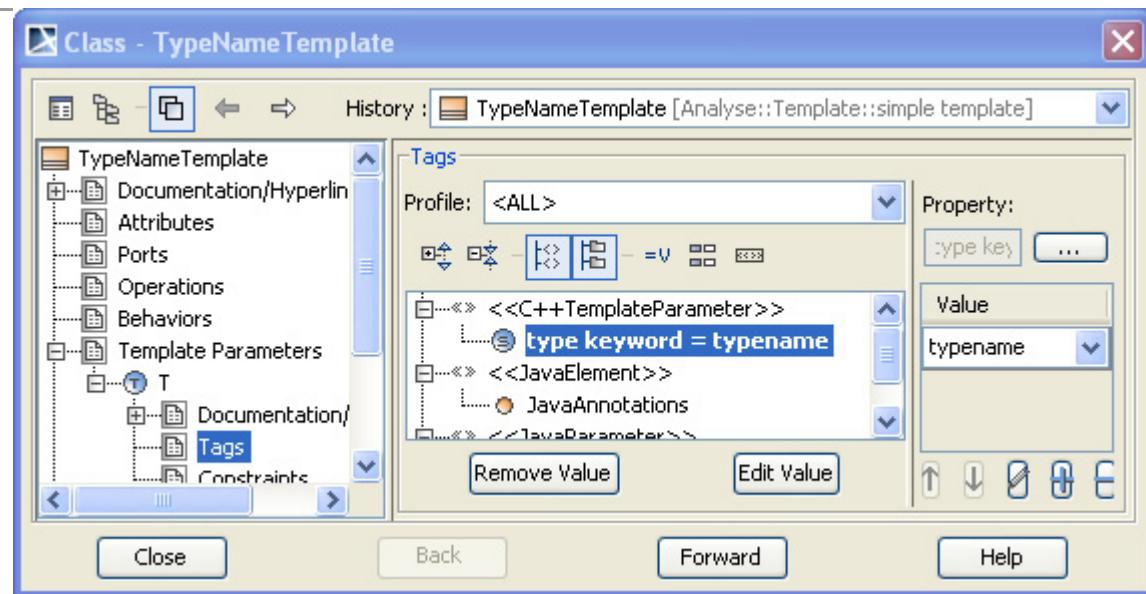
Type of template parameter is always set to UML Class. To generate/reverse typename keyword, type keyword tag is set to *typename*.

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

```
template <class T>
class simpleTemplate {
};

template <typename T>
class TypeNameTemplate {
```





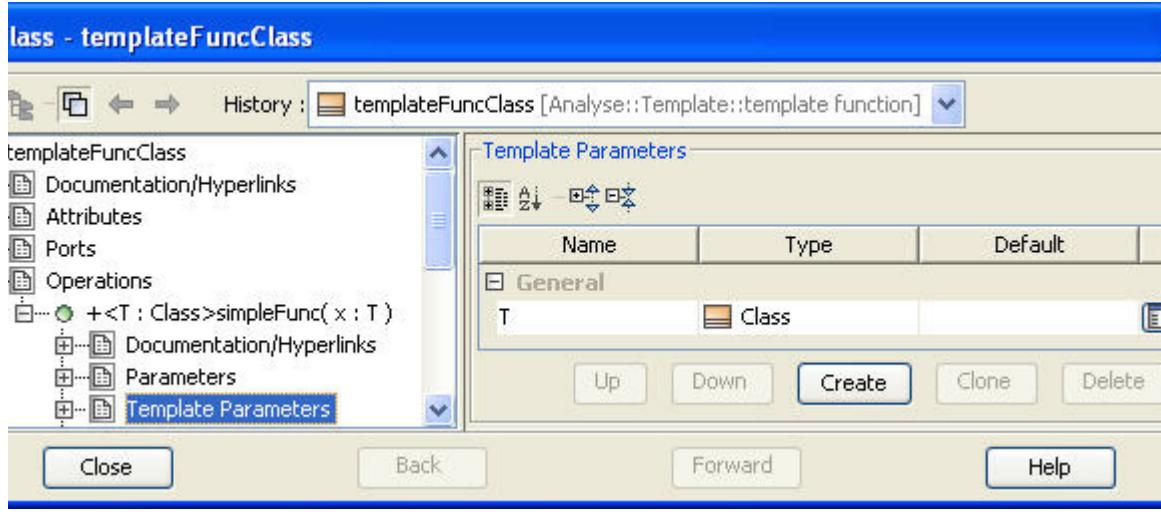
Function Template Definition

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 <class T> void simpleFunc(T x); // overload old style void simpleFunc(int x); // overload new style template<> void simpleFunc<char>(char x);</pre>	<div style="border: 1px solid black; padding: 10px;"> <p style="text-align: center;"><<C++Global>></p> <div style="border: 1px solid black; padding: 5px; background-color: #f0e68c; margin-top: 5px;"> <p>+<T : Class>simpleFunc(x : T)</p> <p>+simpleFunc(x : int)</p> <p>+simpleFunc(x : char)</p> </div> </div>

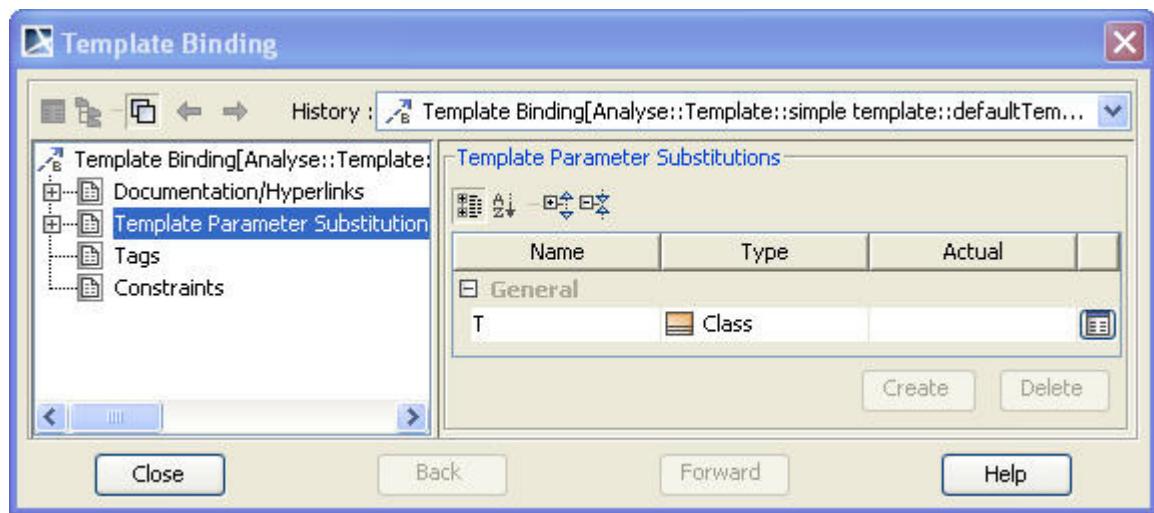
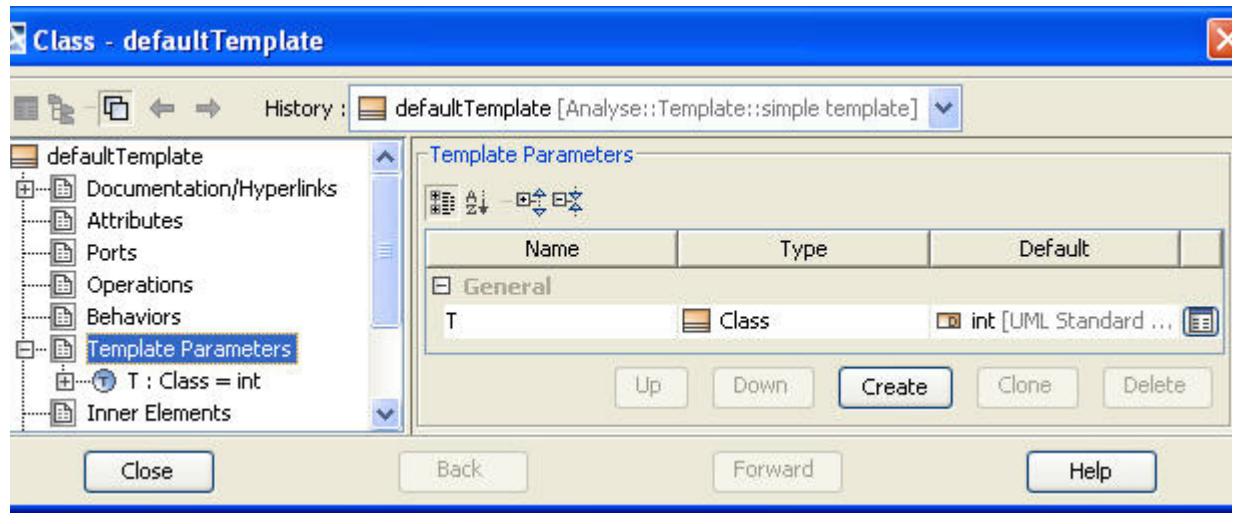


Default template parameter

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*.

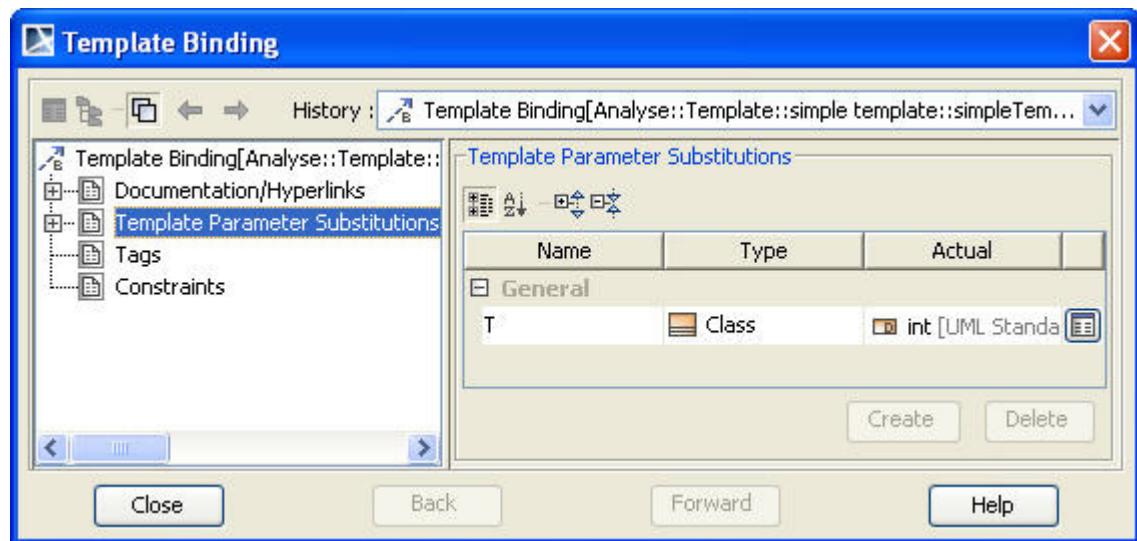
Code	MD-UML
<pre>template <class T=int> class defaultTemplate { };</pre>	<p>The UML diagram illustrates the mapping of a C++ template to UML. It features three components: a template parameter box labeled 'T : Class = int', a 'defaultTemplate' class box below it, and a 'defaultTemplateInstance' class box at the bottom. A vertical line connects the template parameter to the 'defaultTemplate' class. Below this, another vertical line connects the 'defaultTemplate' class to the 'defaultTemplateInstance' class. The text '<T->>' is positioned between the two vertical lines, and '<<bind>>' is placed to the right of the bottom line.</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 <class T> class simpleTemplate { }; simpleTemplate<int> simpleTemplateInstance;</pre>	<p>The UML diagram illustrates the mapping of a C++ template to its instantiation. At the top, a box labeled "T : Class" is connected via a line to a box labeled "simpleTemplate". Below this, a box labeled "simpleTemplateInstance" contains a double-headed arrow labeled "<T->int> <<bind>>".</p>



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

Code	MD-UML
<pre>template <class T> class T1Class { }; template <class T> class T2Class { }; T1Class<T2Class<int>> ...</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 <class T> class b { template <class T> class c { }; }; b<int>::c<char> ...</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.

```

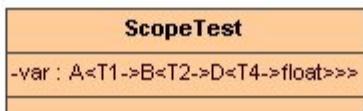
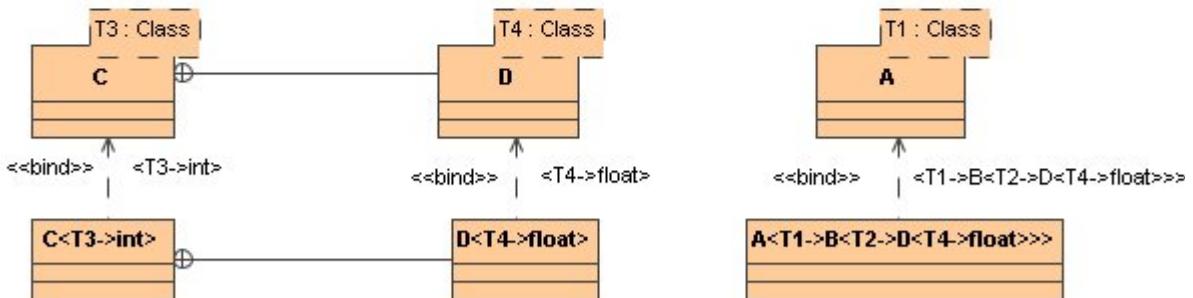
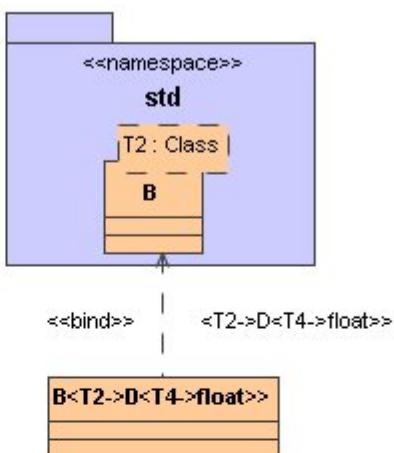
template <class T1>
class A {};

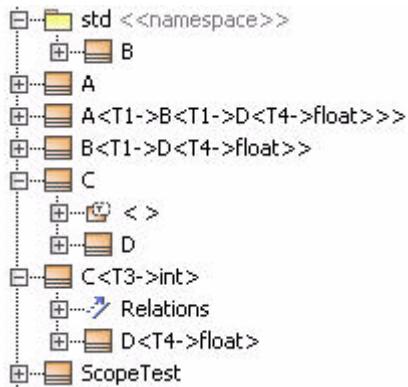
namespace std {
    template <class T2>
    class B {};
}

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

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

```

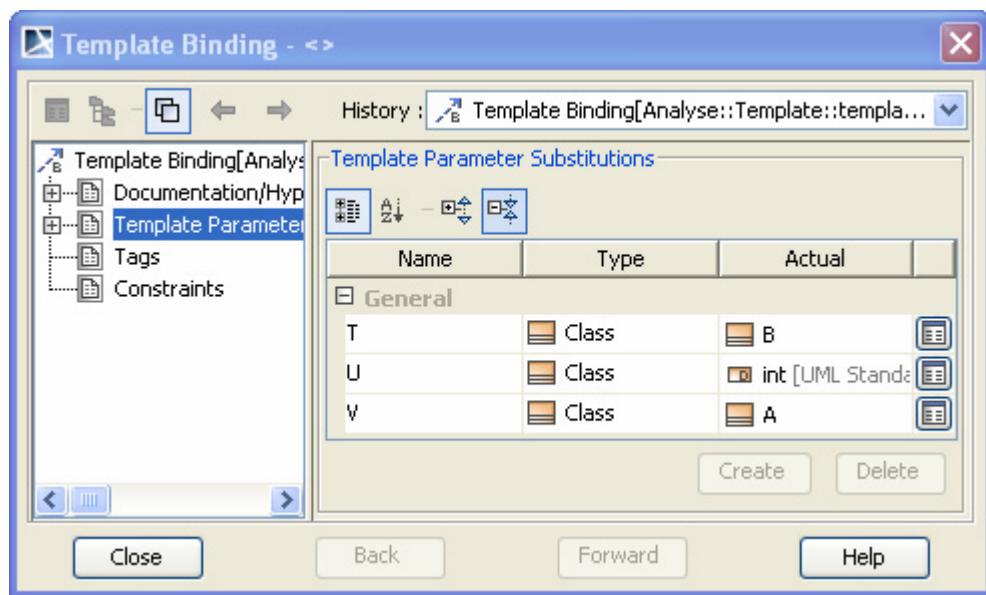




Partial template instantiation

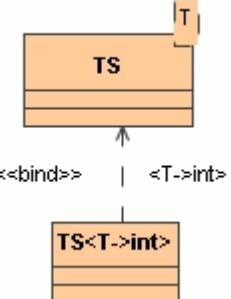
C++ partial template instantiation use the same mapping as Template Instantiation and the unbinded parameter is binded to the specific template parameter class.

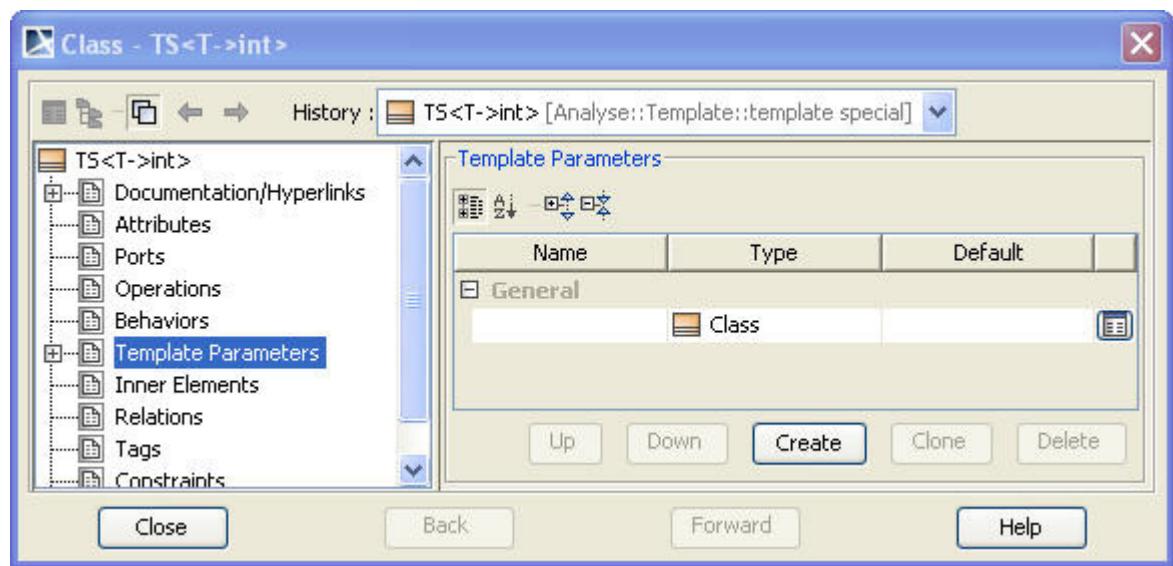
Code	MD-UML
<pre> template <class T,class U,class V> class PT {}; template <class A,class B> class PT<B, int, A> {}; </pre>	<p>MD-UML diagram illustrating the mapping of the provided C++ code:</p> <pre> PT --- T --- U --- V --- PT<U>int, T>B, V>A> --- U --- V --- int --- A --- B </pre>



Template specialization

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

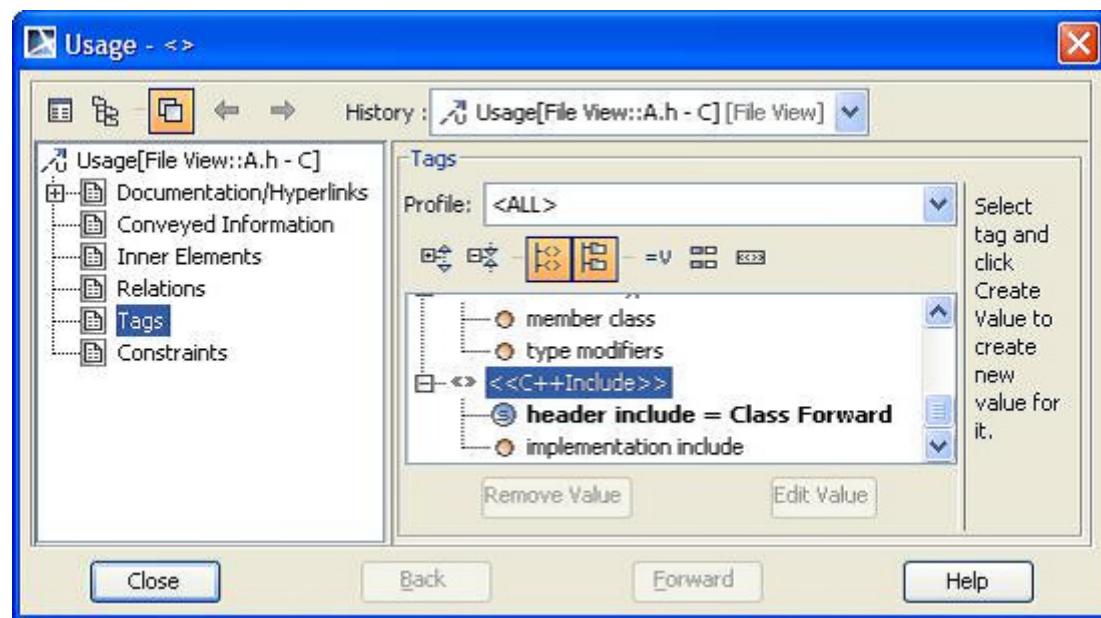
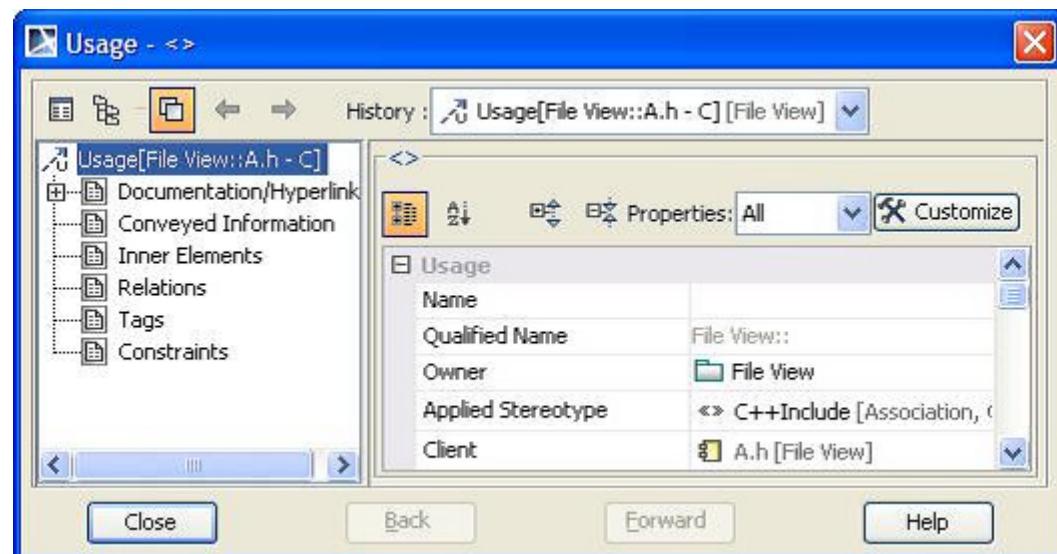
Code	MD-UML
<pre>template <class T> class TS {}; template <> class TS<int> {};</pre>	



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>	<p>The MD-UML diagram illustrates the relationship between classes A and C. At the top, a yellow rounded rectangle labeled "A.h" is labeled with "<<component>>". Below it, a dashed line labeled "<<use>>" connects to two other components: a blue rectangle labeled "A" and an orange rectangle labeled "C". The blue rectangle "A" contains the text "-c : C#\$ *".</use></component></p>

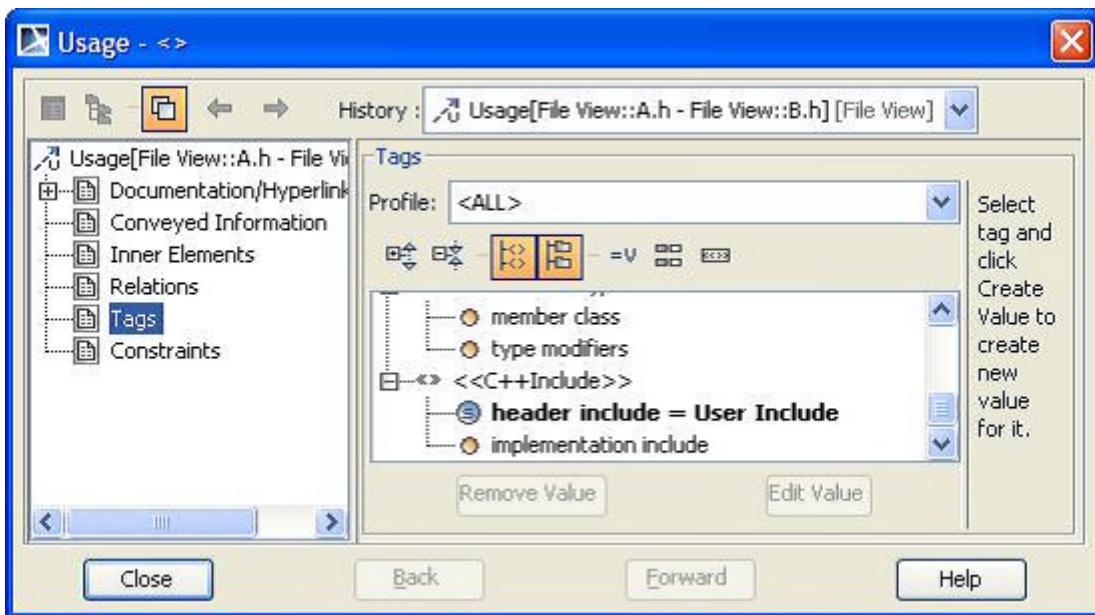


Include declaration

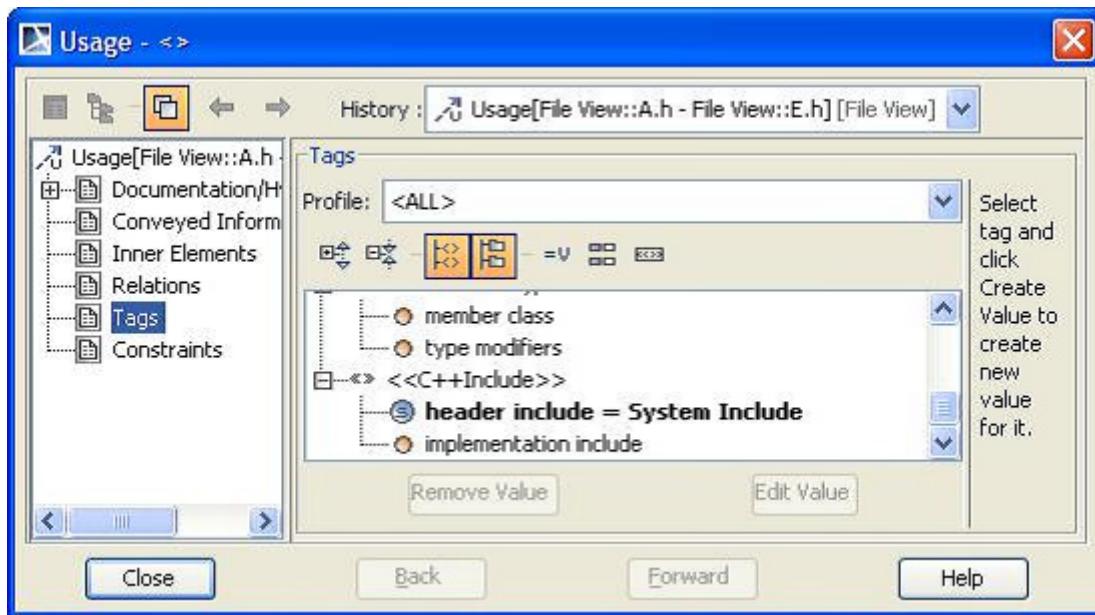
The example code is declared in A.h file. The <<use>> association is also applied to C++Include stereotype shown in section 3.36 Forward class declaration.

Code	MD-UML
<pre>#include "B.h" #include <E.h> class A { private: B* b; E* e; };</pre>	

Specification for #include "B.h"



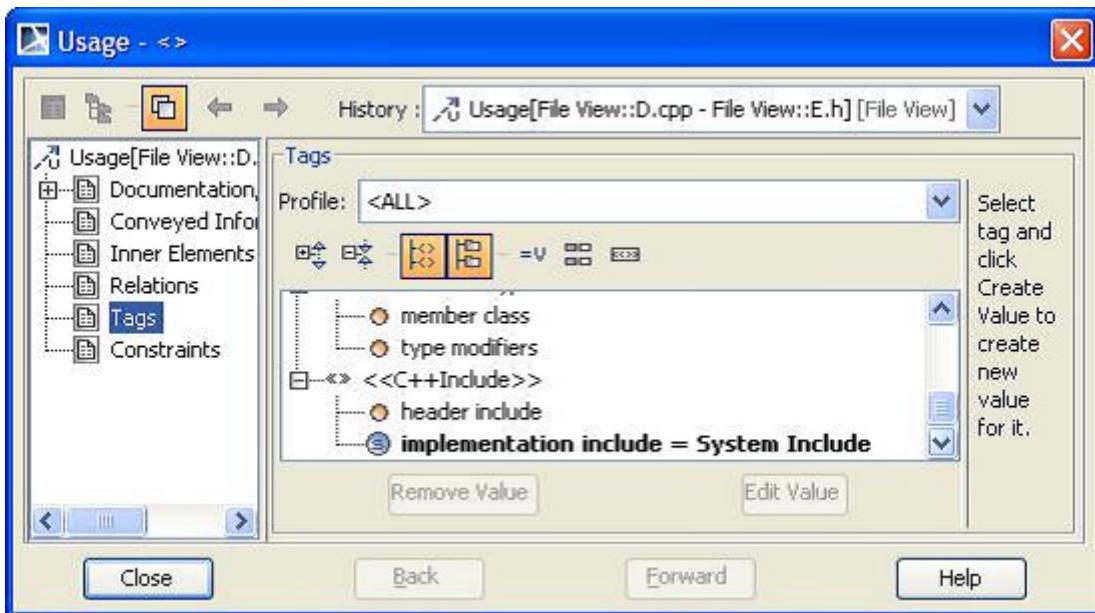
Specification for #include <E.h>



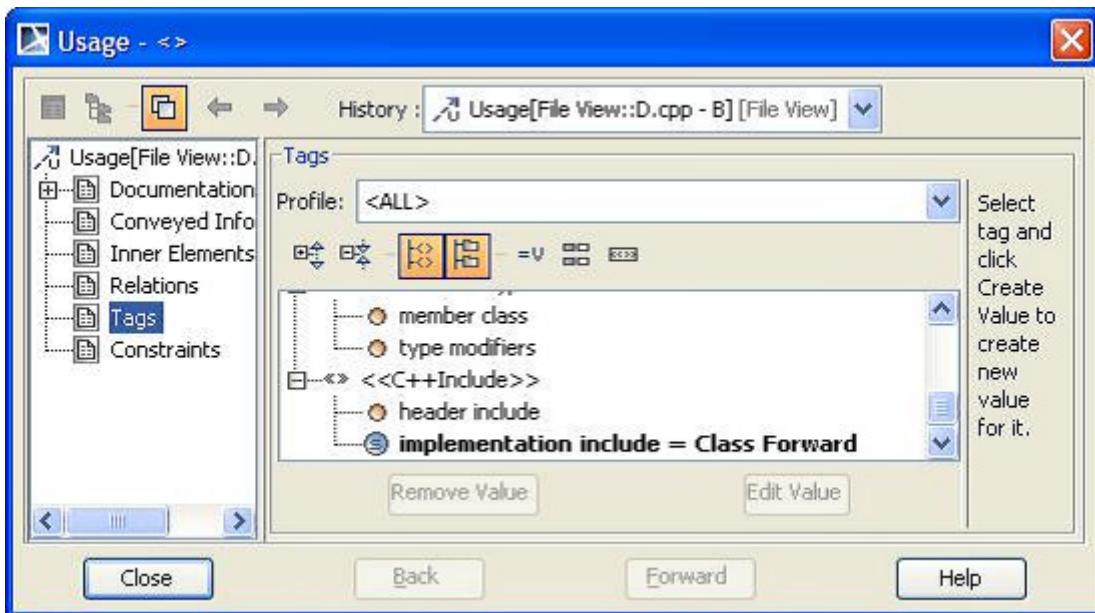
The example code is declared in D.cpp file.

Code	MD-UML
<pre>#include <E.h> class B; class D { private:B* b; E* e; }</pre>	<pre> graph TD Dcpp["<<component>>\nD.cpp"] -- "<<use>>" --> Eh["<<component>>\nE.h"] Dcpp -- "<<use>>" --> B["B"] Dcpp -- "<<use>>" --> E["E"] classD["D\n-b : B*\$*\n-e : E*\$*"] </pre>

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 <= 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

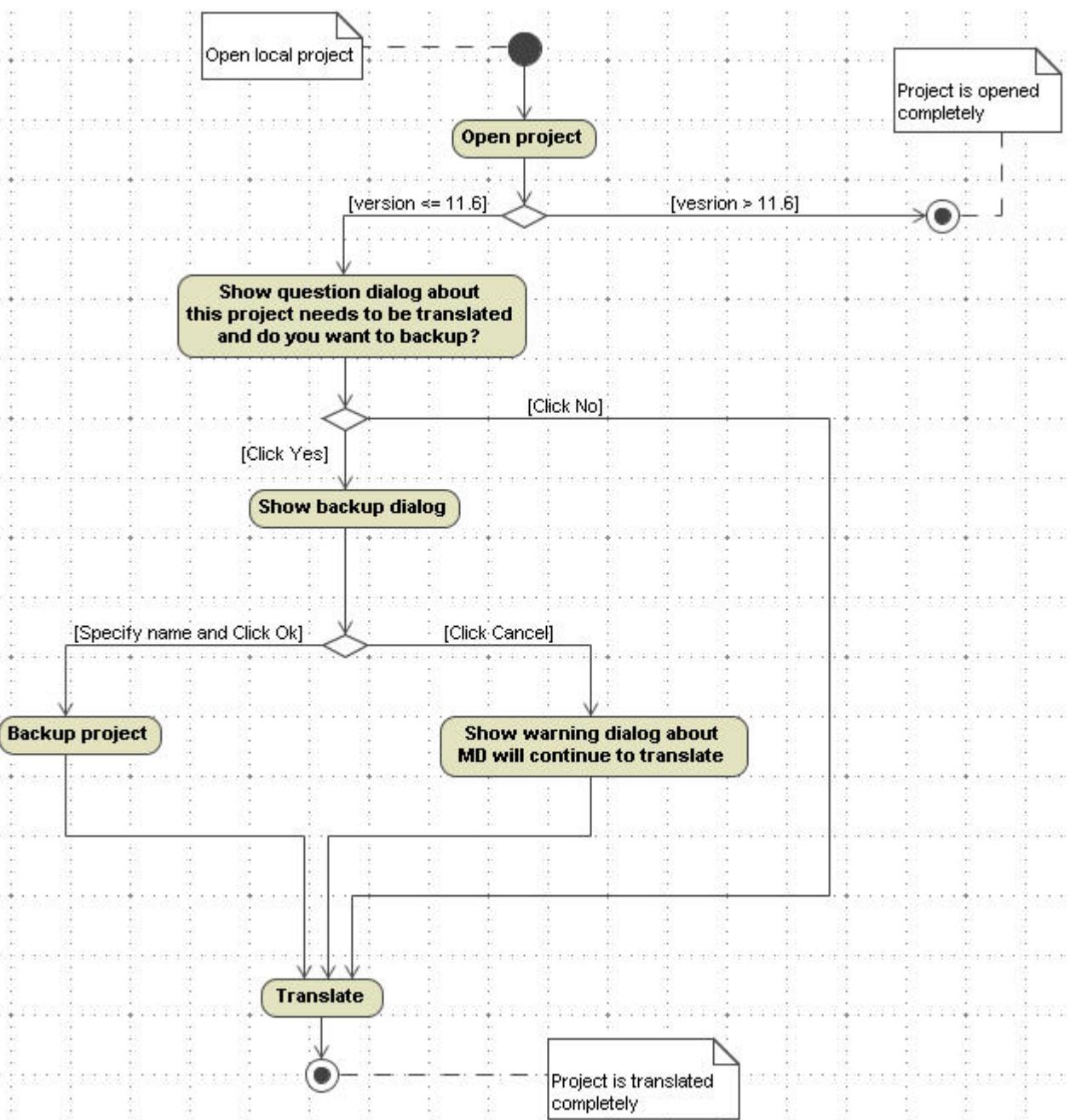


Figure 6 -- Open local project Activity Diagram

Open teamwork project

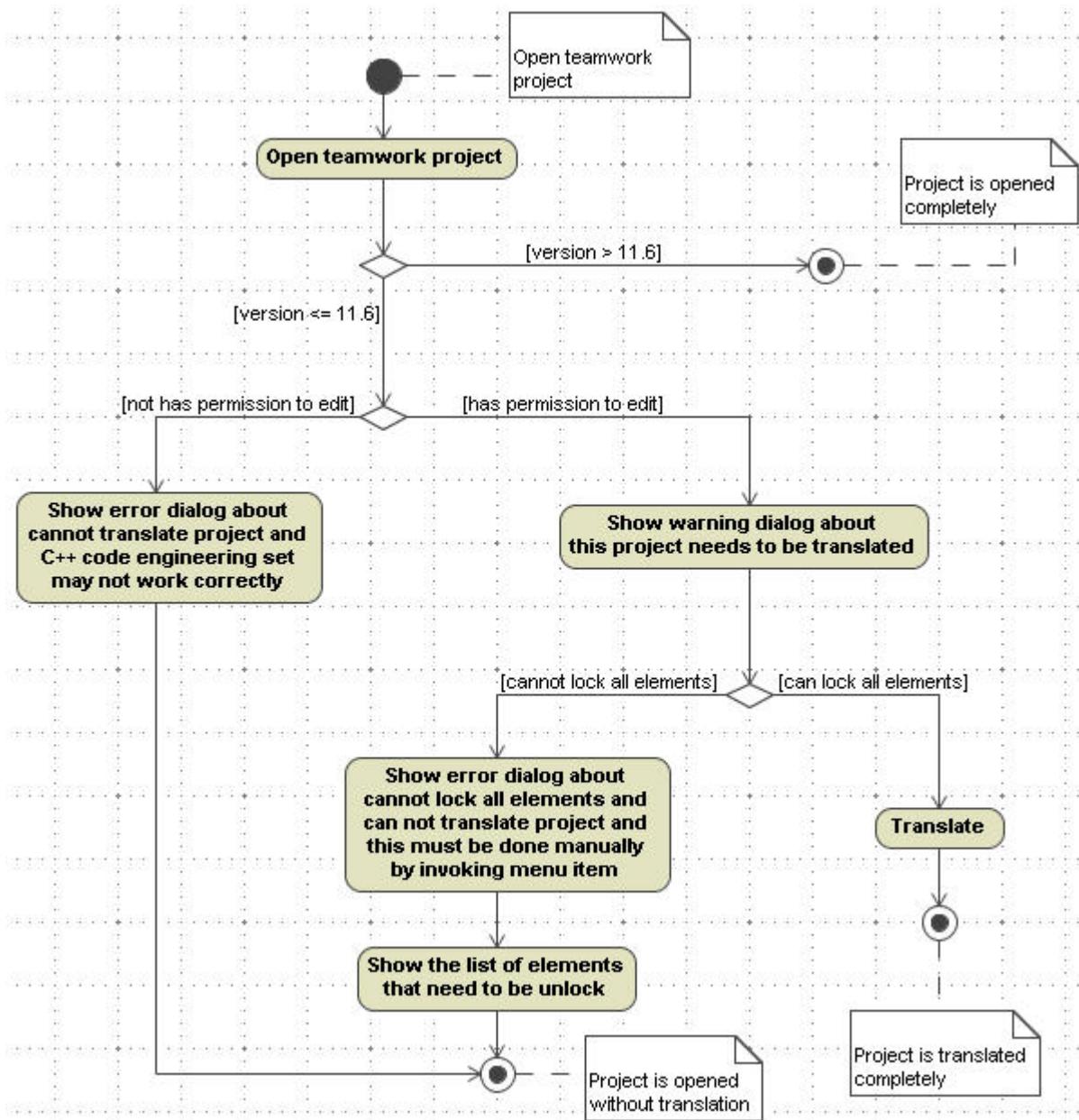


Figure 7 -- Open teamwork project Activity Diagram

Import MagiDraw project

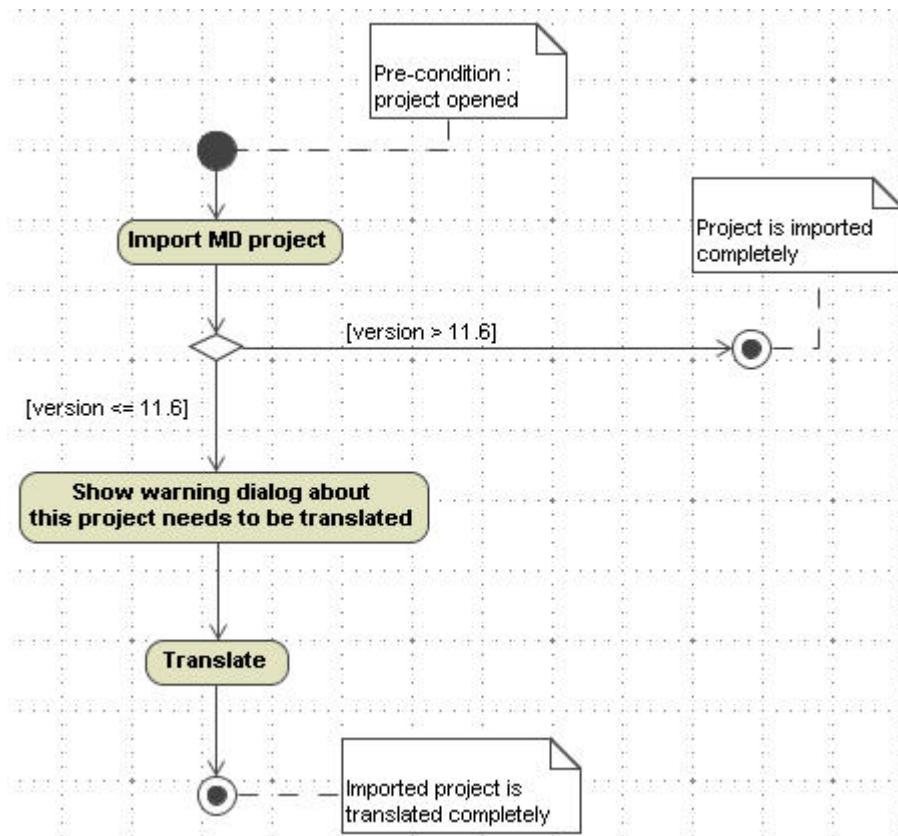


Figure 8 -- Import MagicDraw project Activity Diagram

Use module

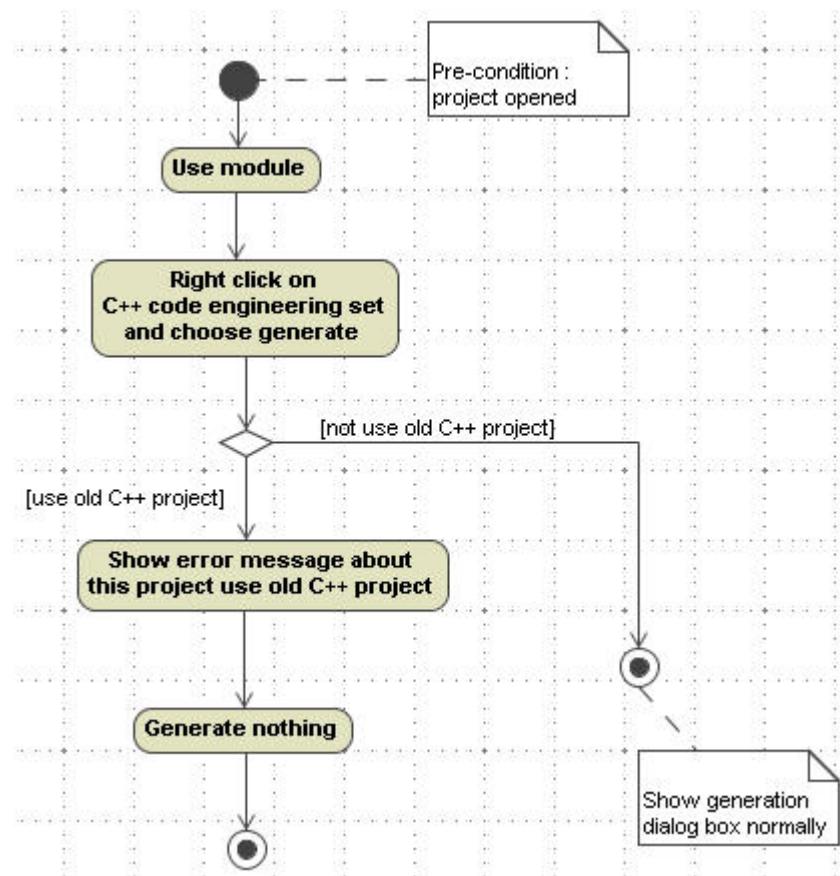


Figure 9 -- Use module Activity Diagram

Update C++ Language Properties and Profiles

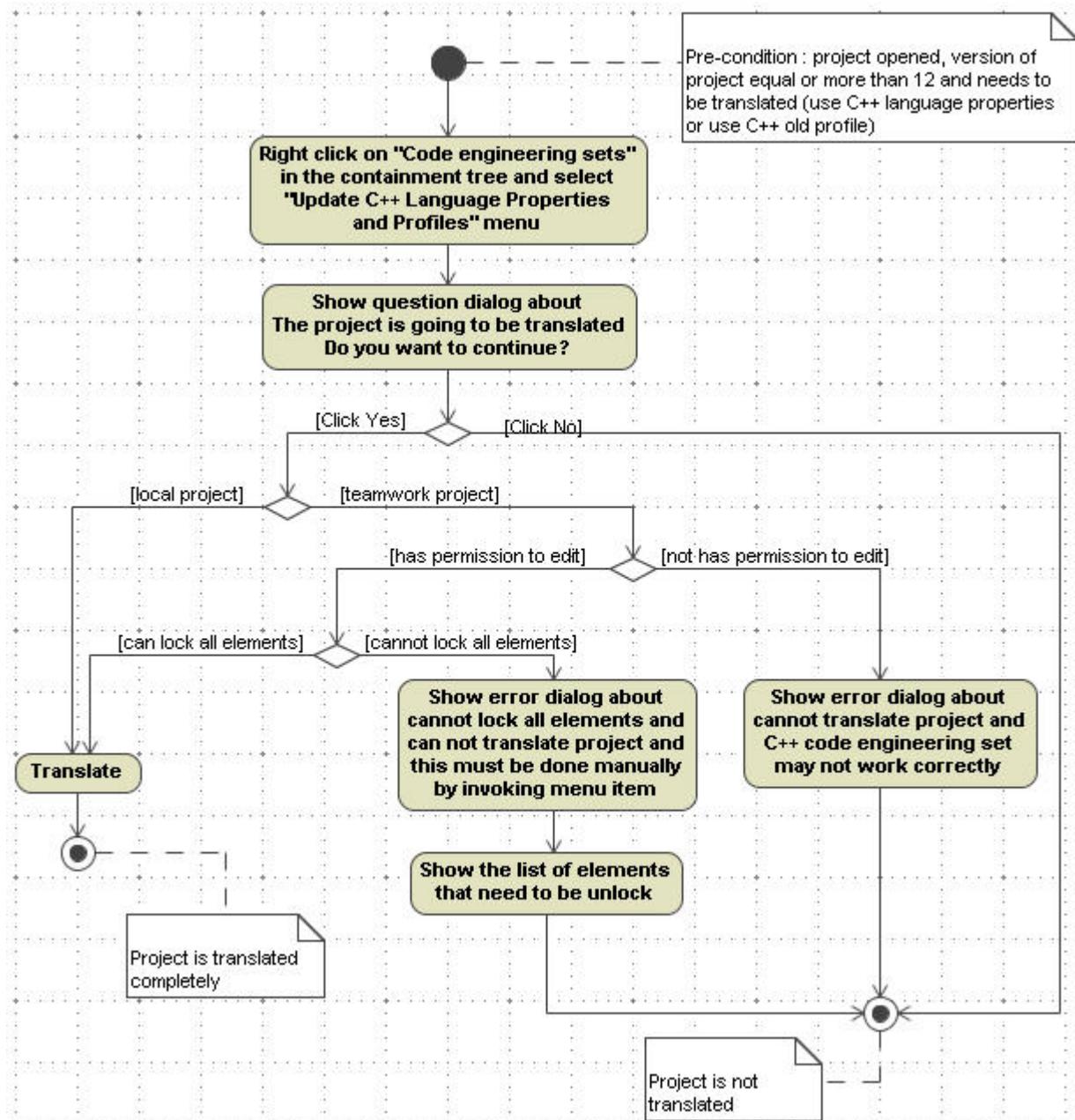


Figure 10 -- Update C++ Language Properties and Profiles Activity Diagram

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.

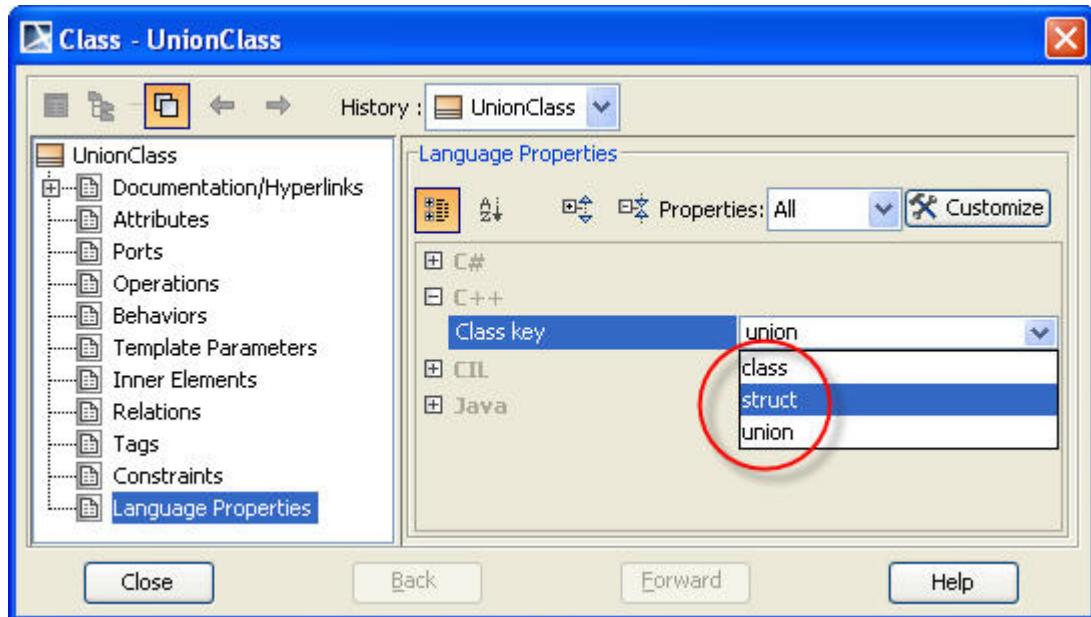
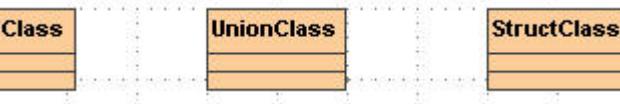
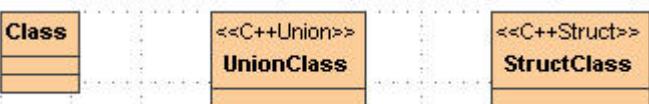
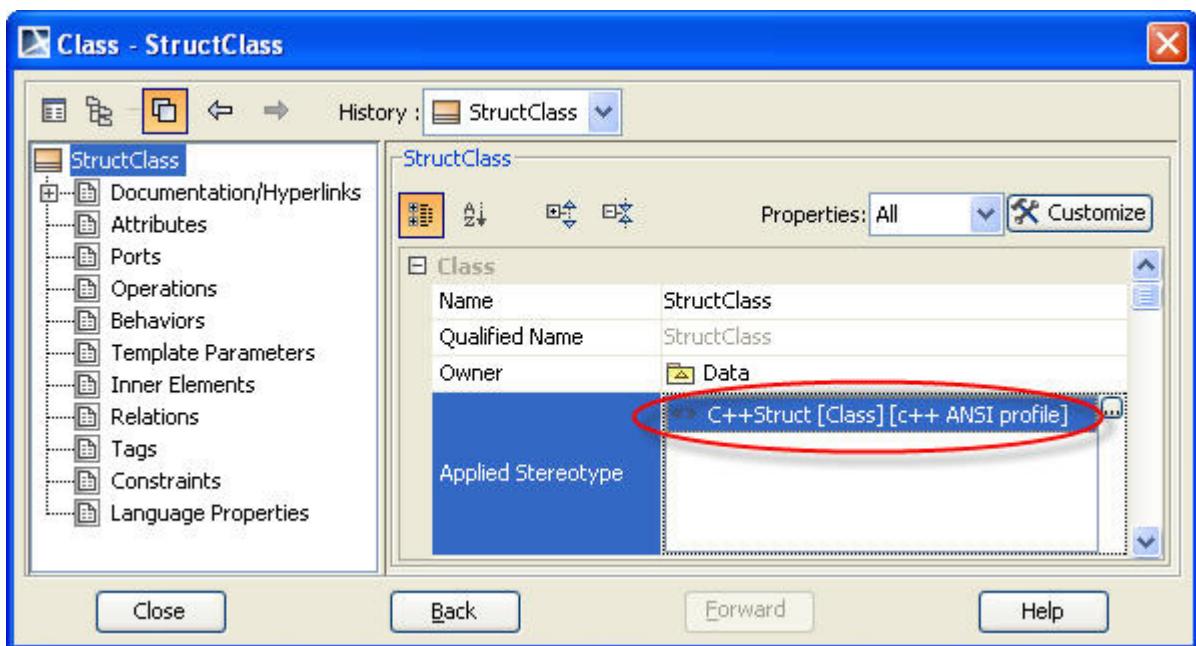


Figure 11 -- Class Language Properties

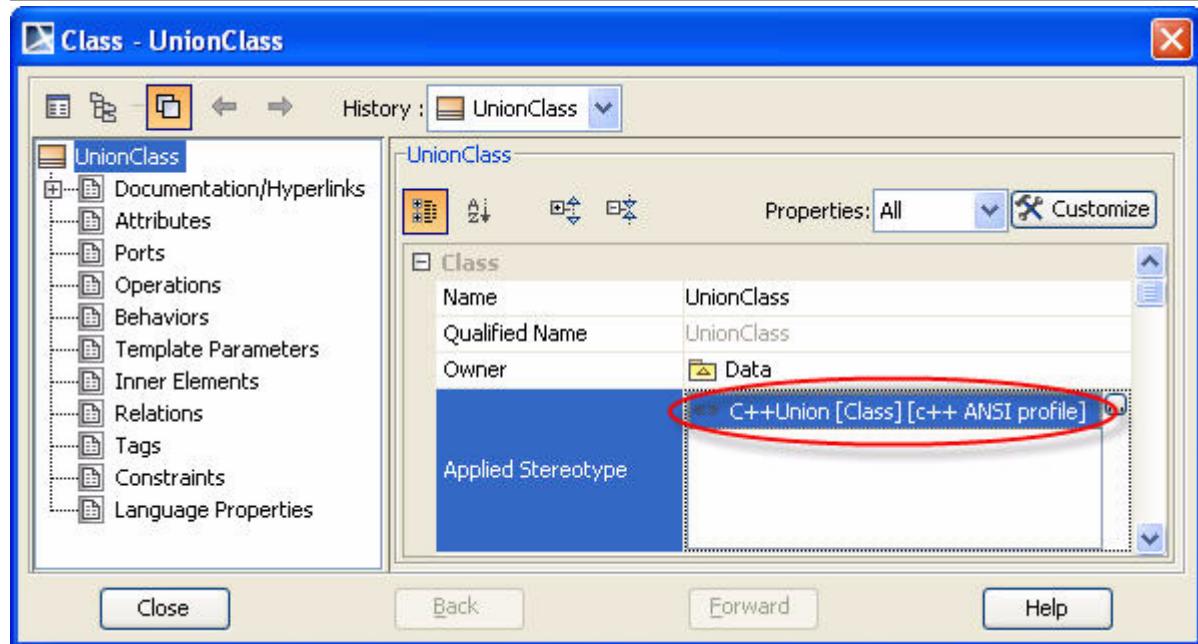
Class - Class key

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**.

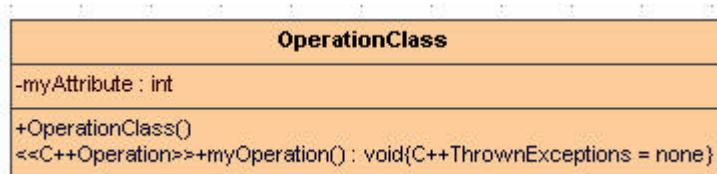


Figure 12 -- Operation Example in Class Diagram

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

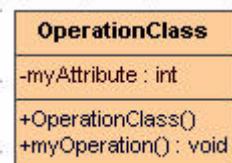


Figure 13 -- Translated Operation in Class Diagram

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

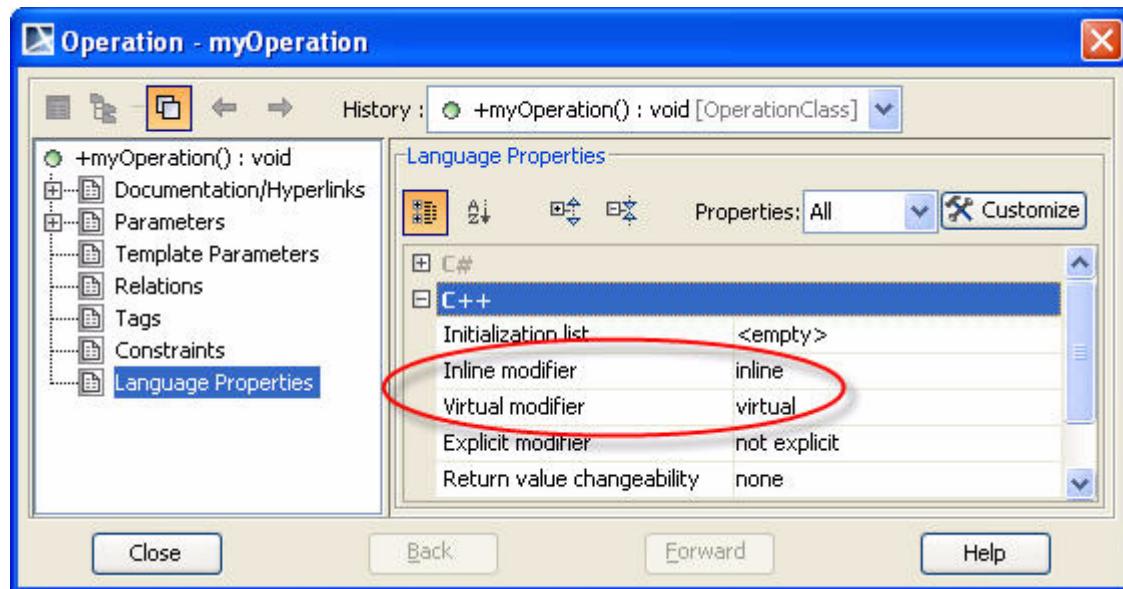


Figure 14 -- Operation Language Properties (Operation)

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

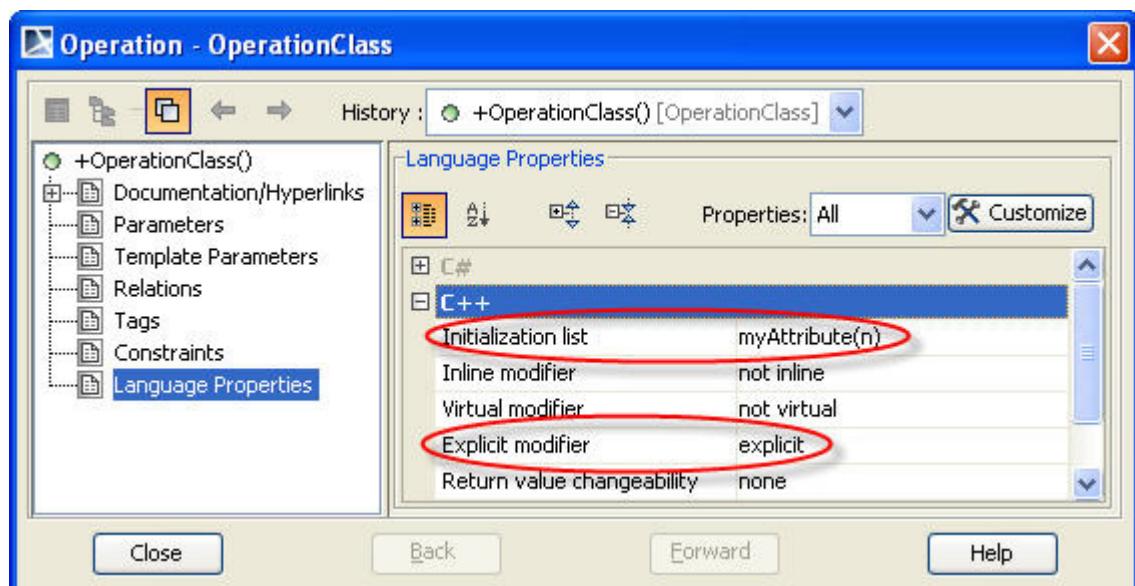


Figure 15 -- Operation Language Properties (Constructor)

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

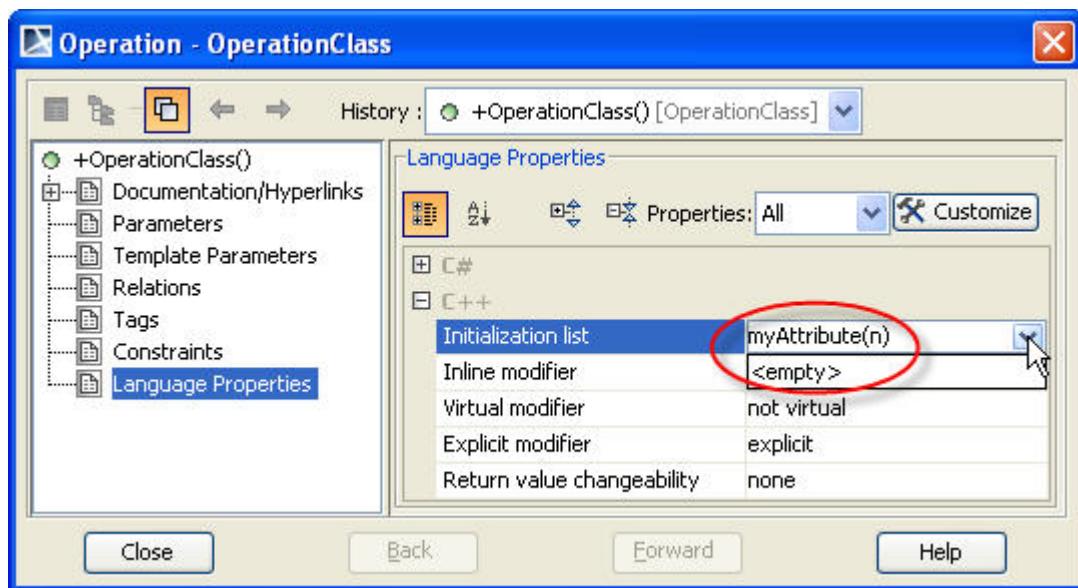
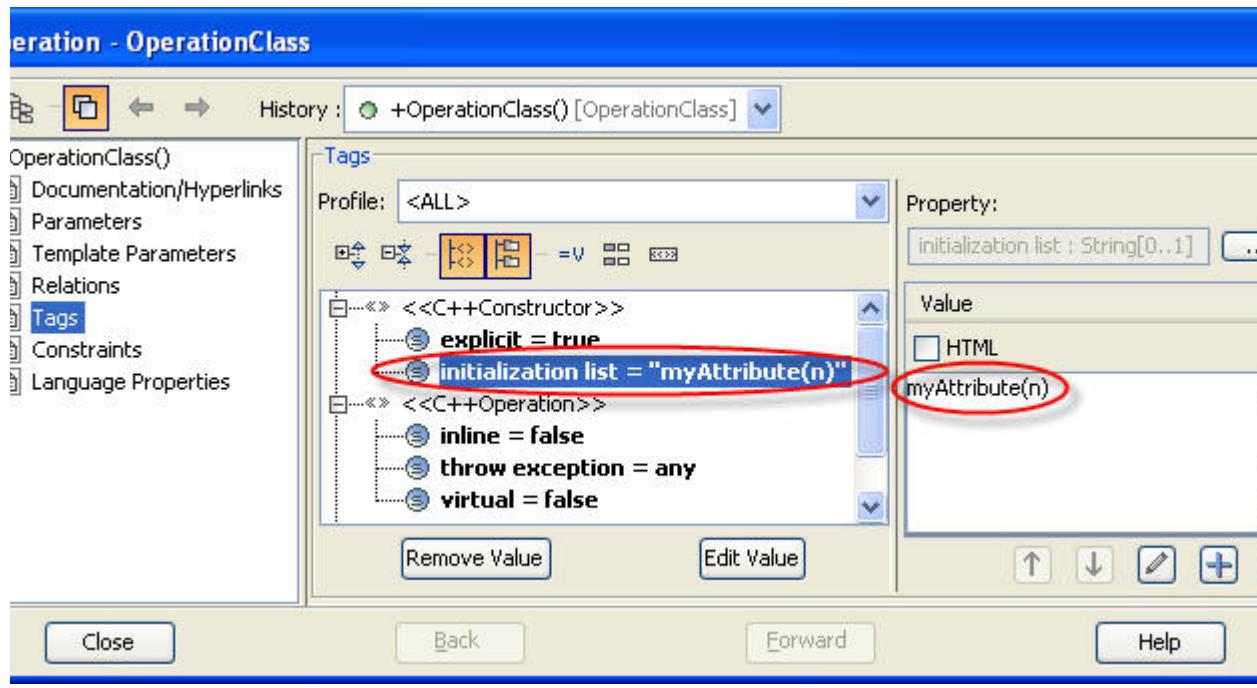


Figure 16 -- Operation Initialization list

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



Operation - Inline Modifier

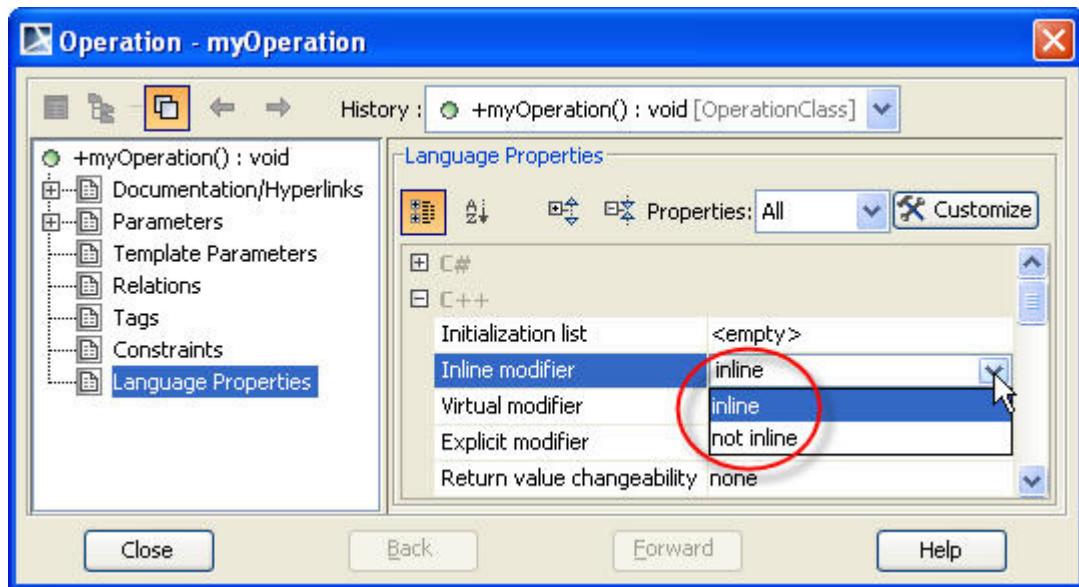
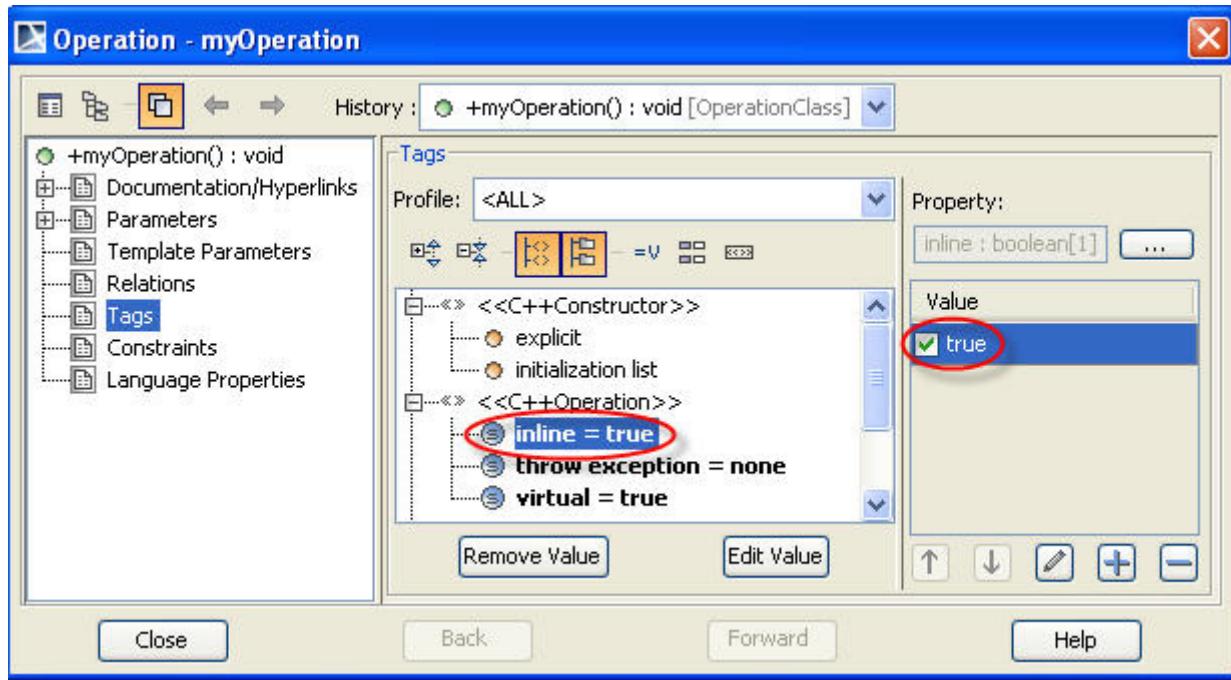


Figure 17 -- Operation Inline modifier

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



Operation - Virtual Modifier

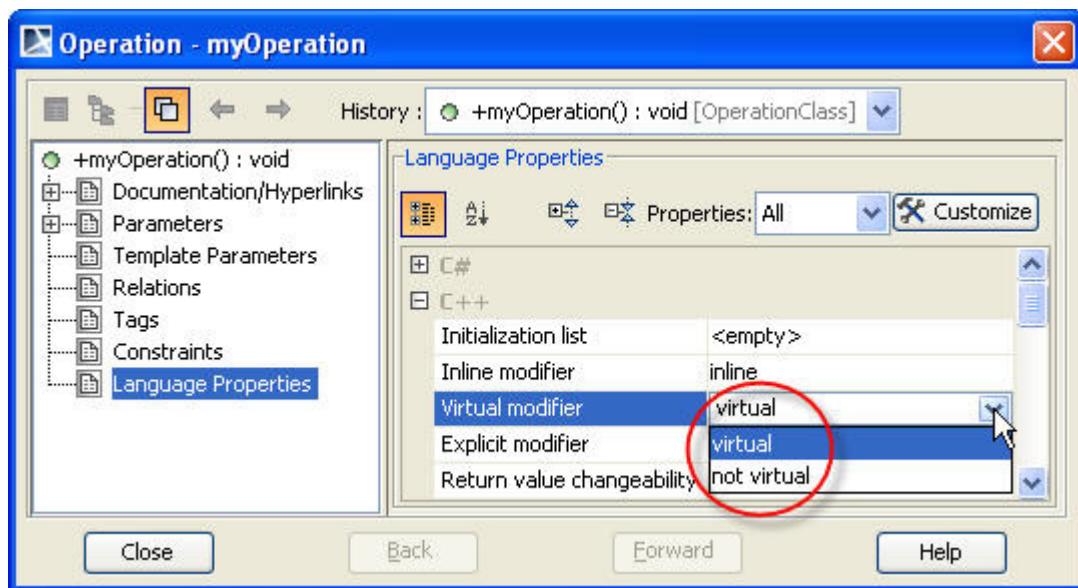
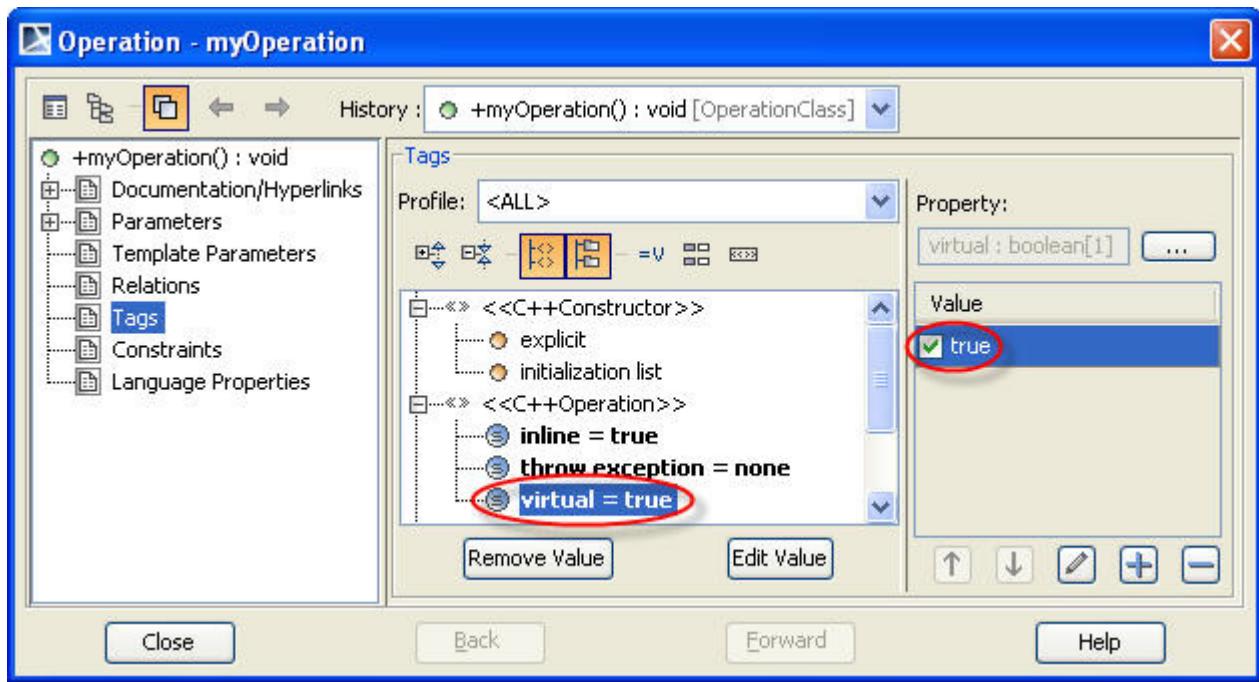


Figure 18 -- Operation Virtual modifier

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



Operation - Explicit Modifier

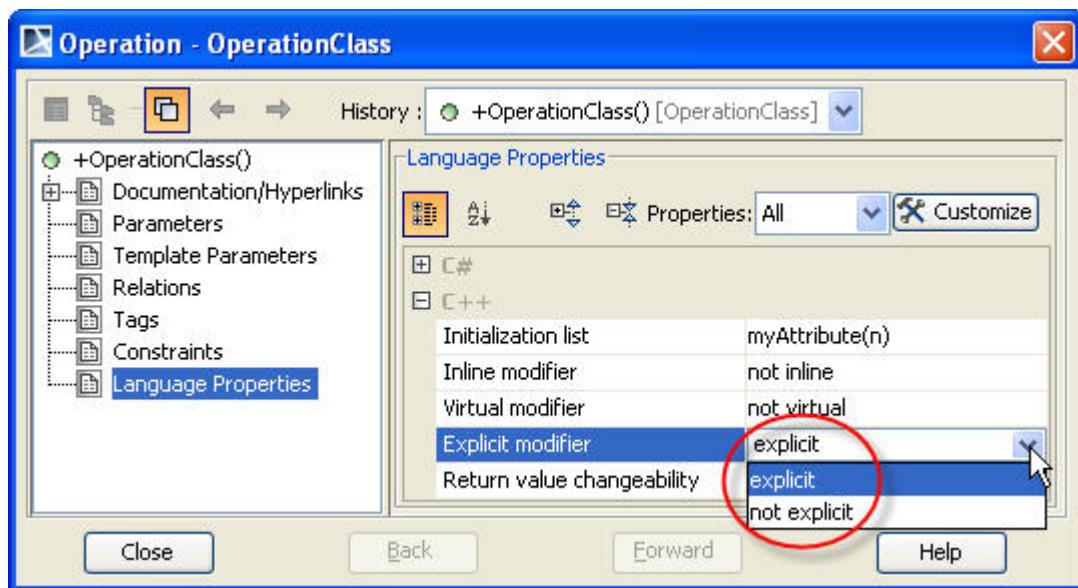
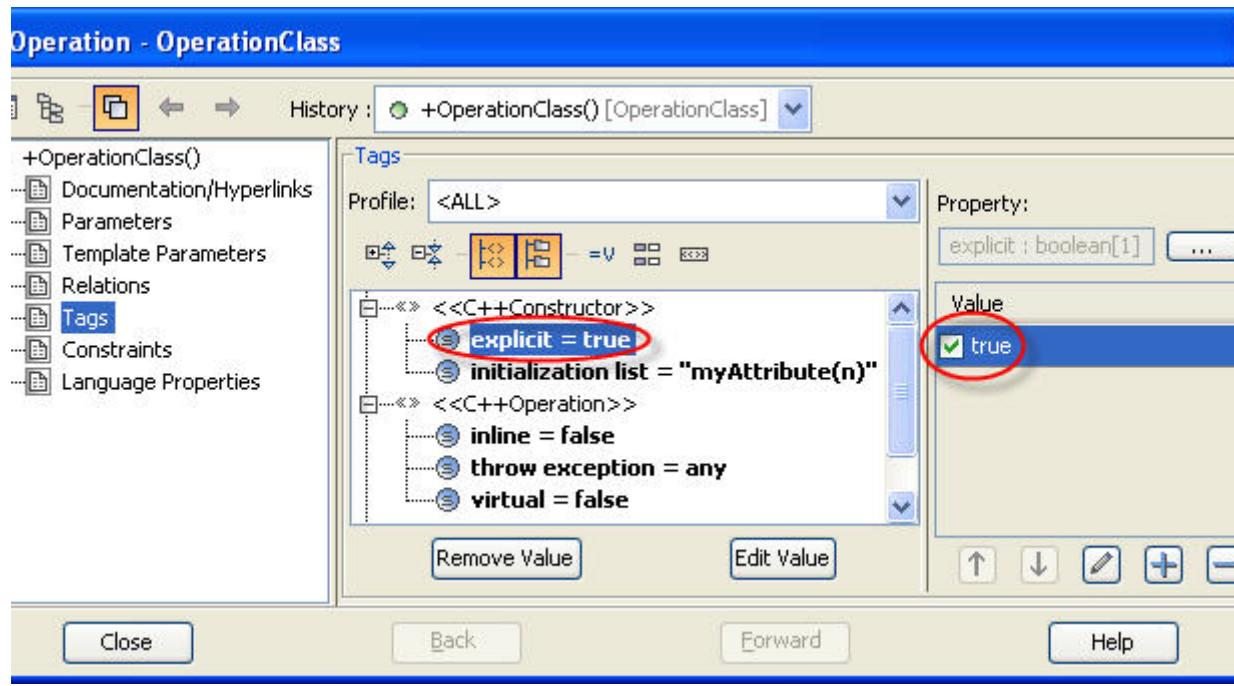


Figure 19 -- Operation Explicit modifier

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



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.

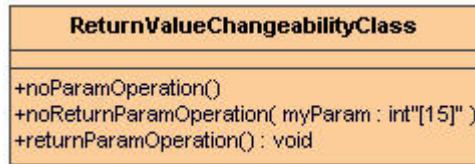


Figure 20 -- Return value changeability Example in Class Diagram

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

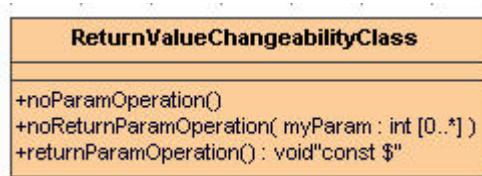


Figure 21 -- Translated Return value changeability in Class Diagram

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

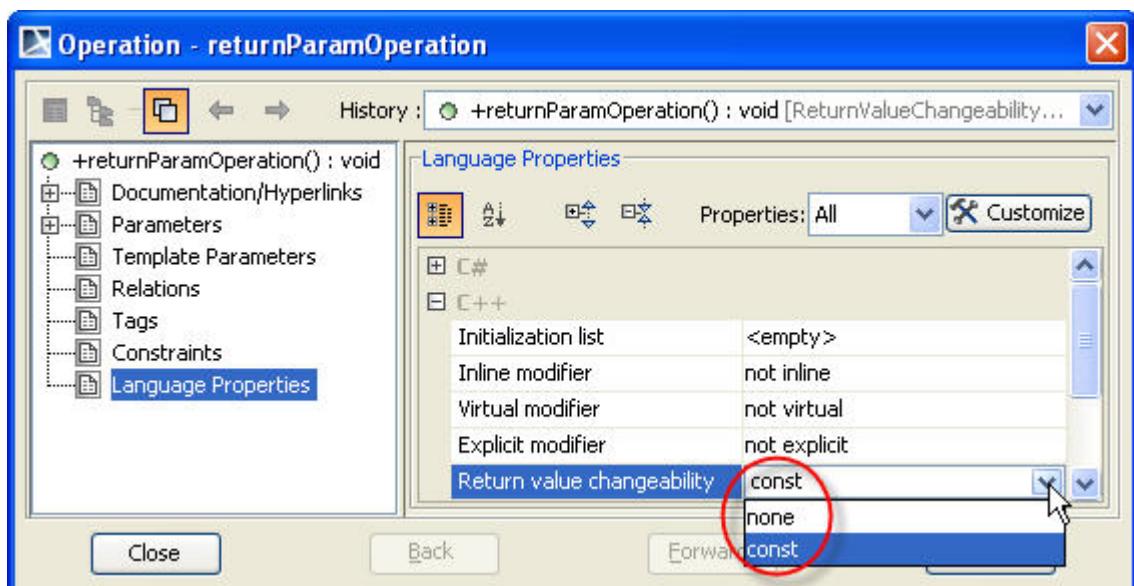
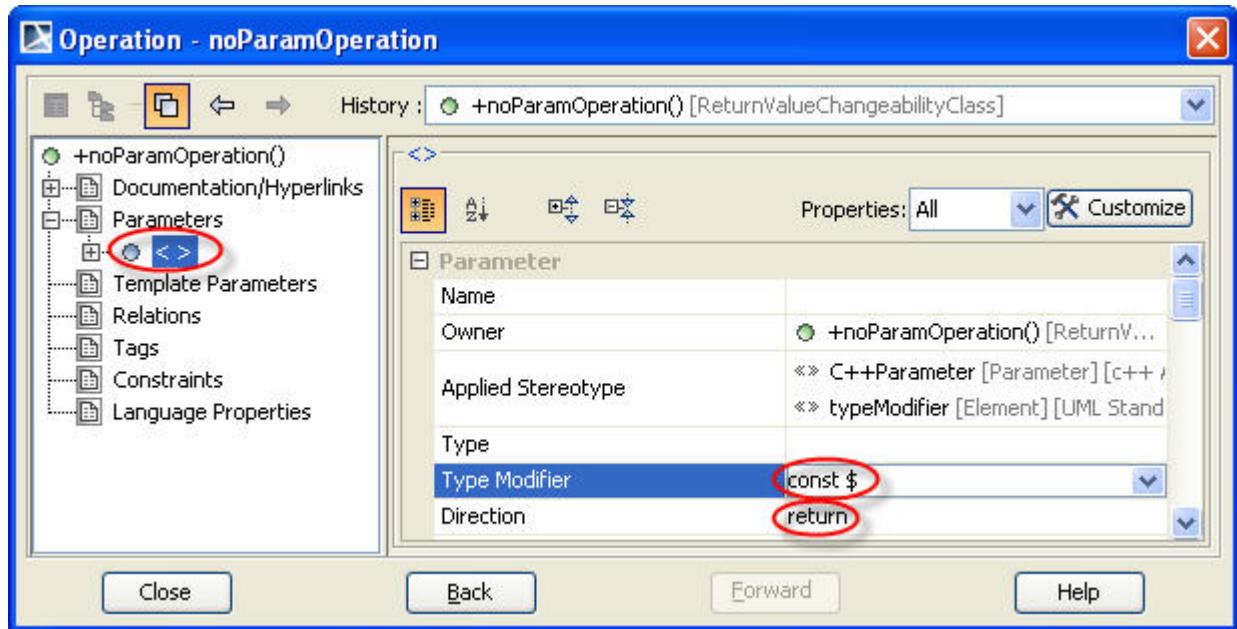


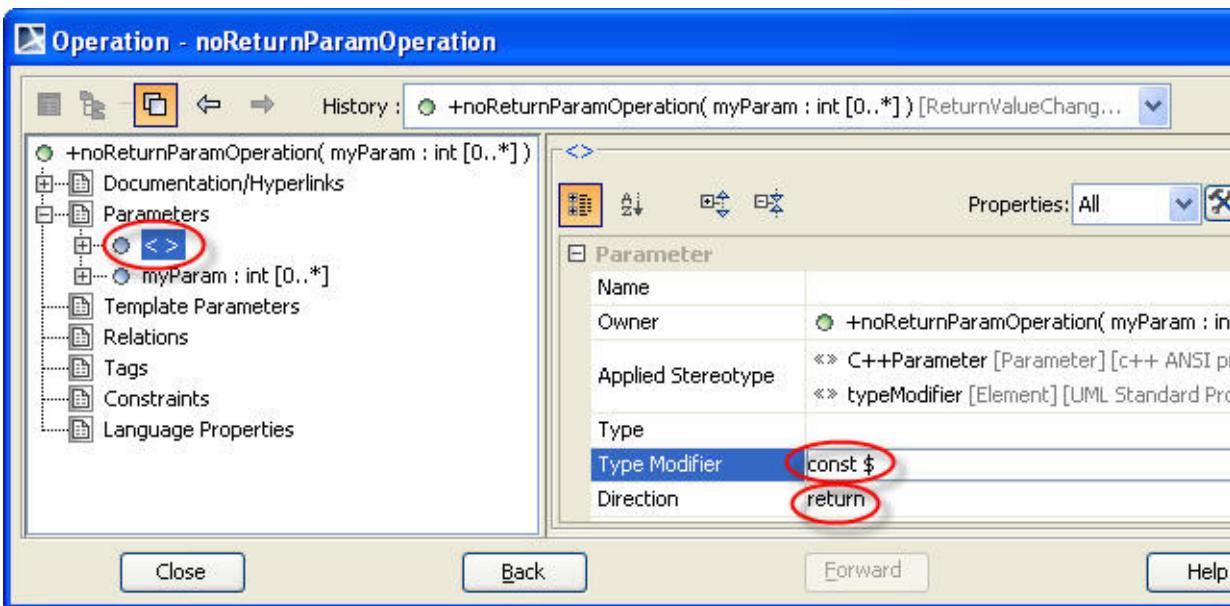
Figure 22 -- Operation Return value changeability

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



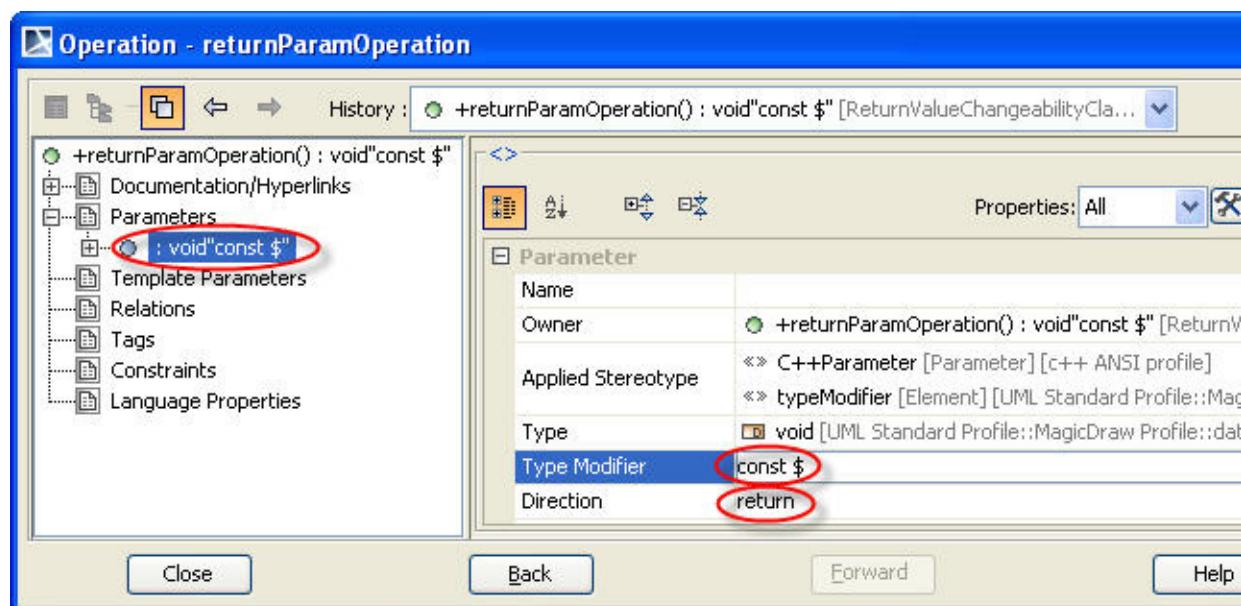
const, operation has parameter but does not have return type parameter.

Create one return type parameter and set **Type Modifier** to **const \$**.



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]**.

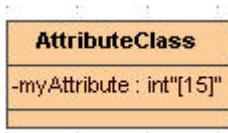


Figure 23 -- Attribute Example in Class Diagram

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

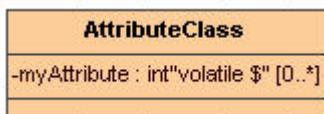


Figure 24 -- Translated Attribute in Class Diagram

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**.

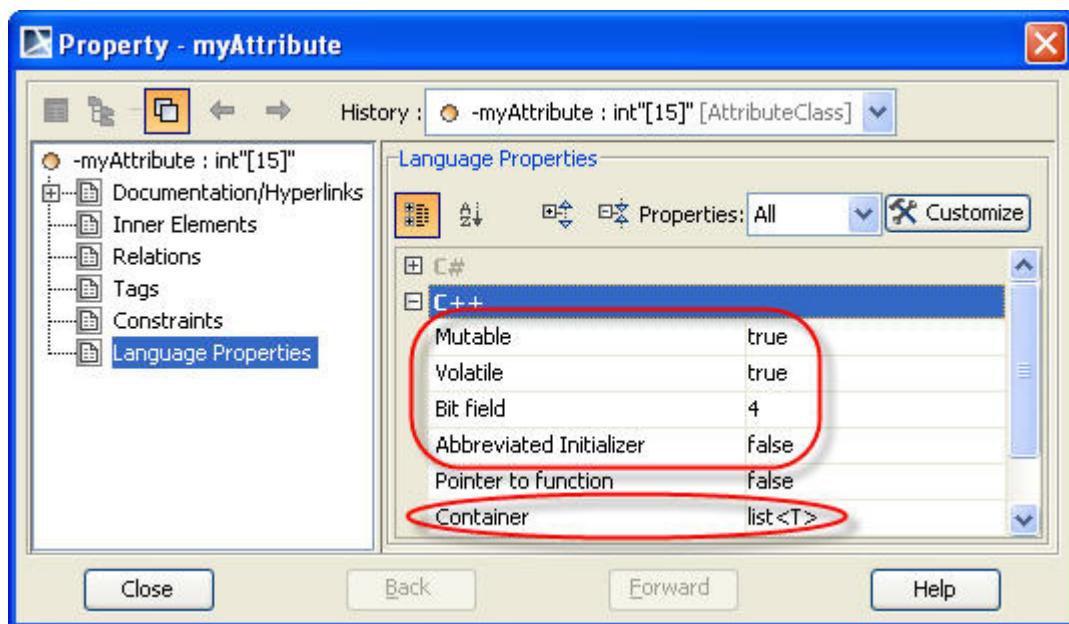


Figure 25 -- Attribute Language Properties

Attribute - Mutable

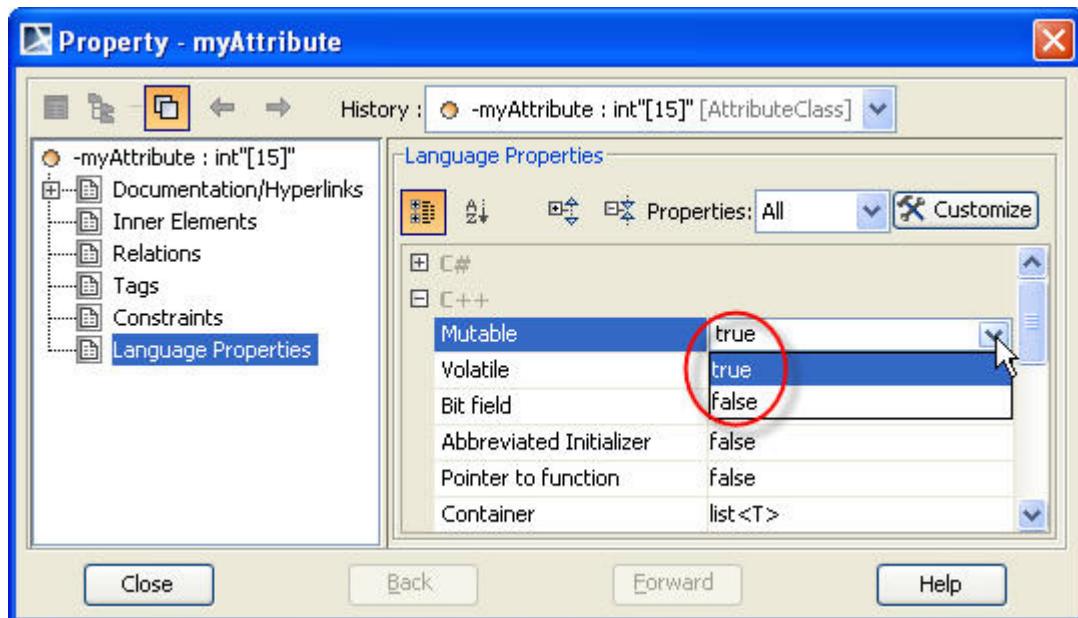
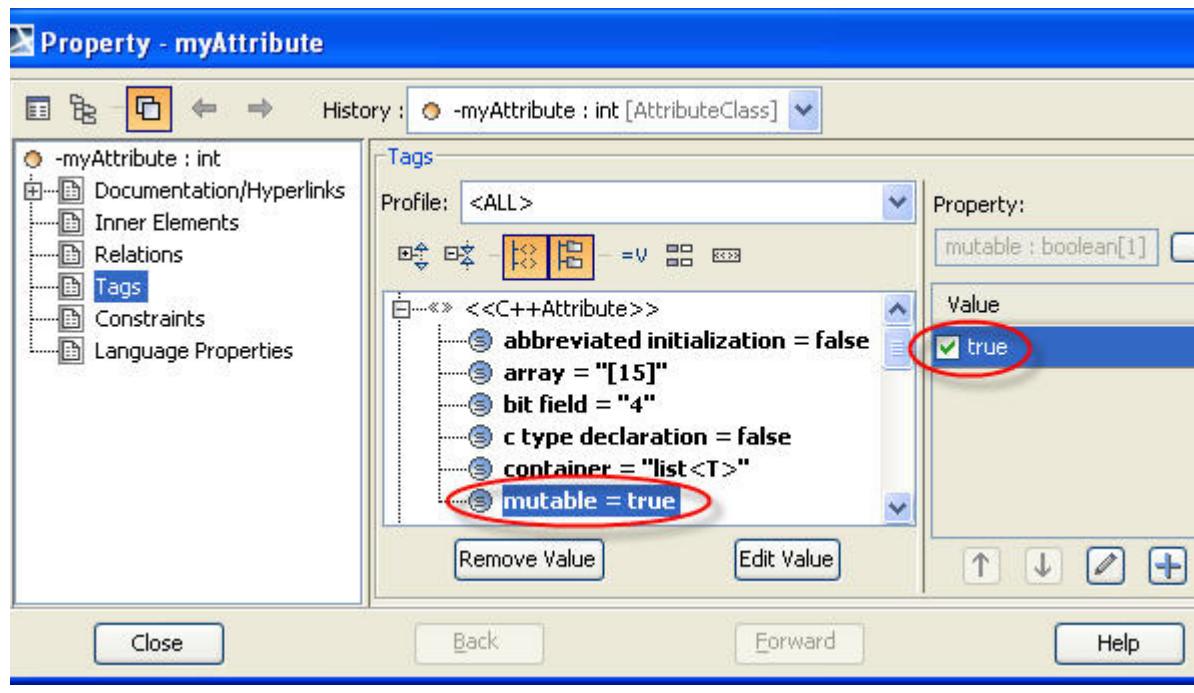


Figure 26 -- Attribute Mutable

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



Attribute - Volatile

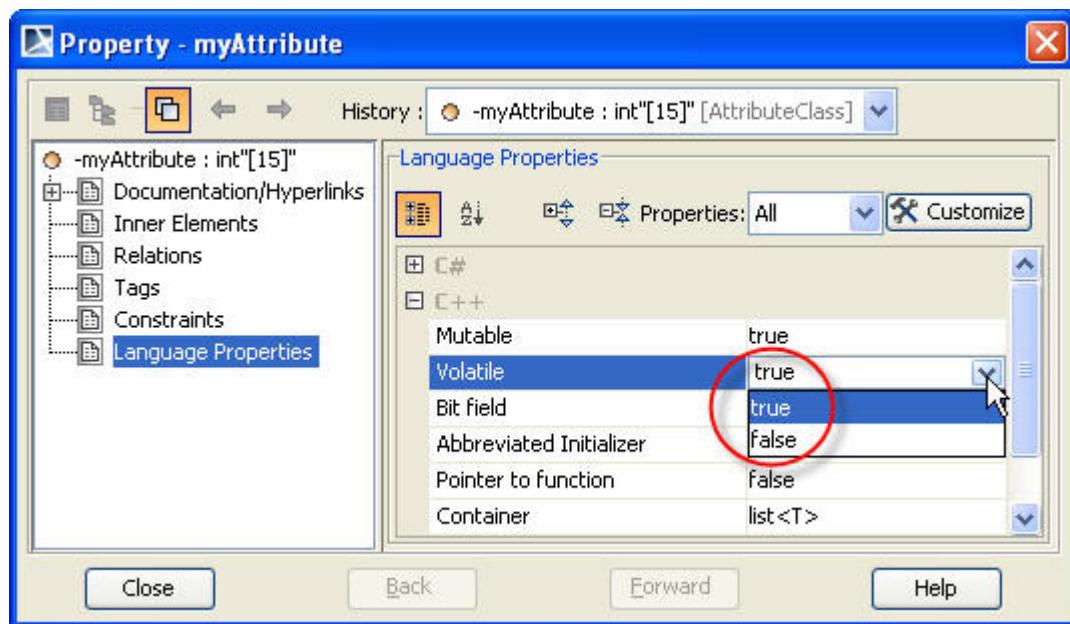
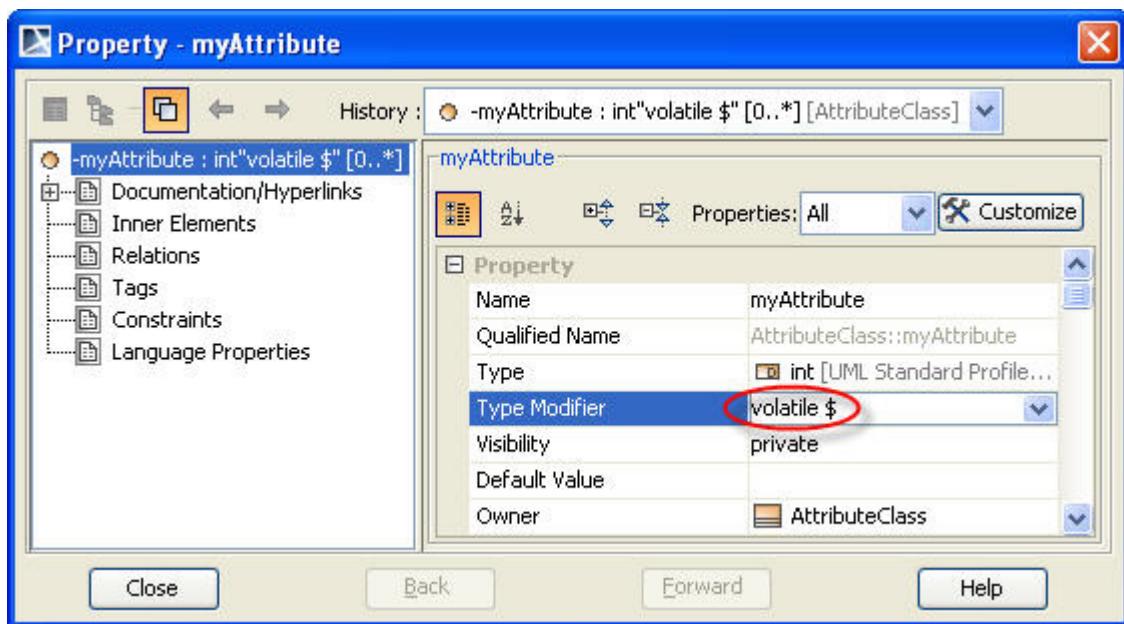


Figure 27 -- Attribute Volatile

Old value	Translation
false	no change.
true	Set Type Modifier to volatile \$.



Attribute - Bit field

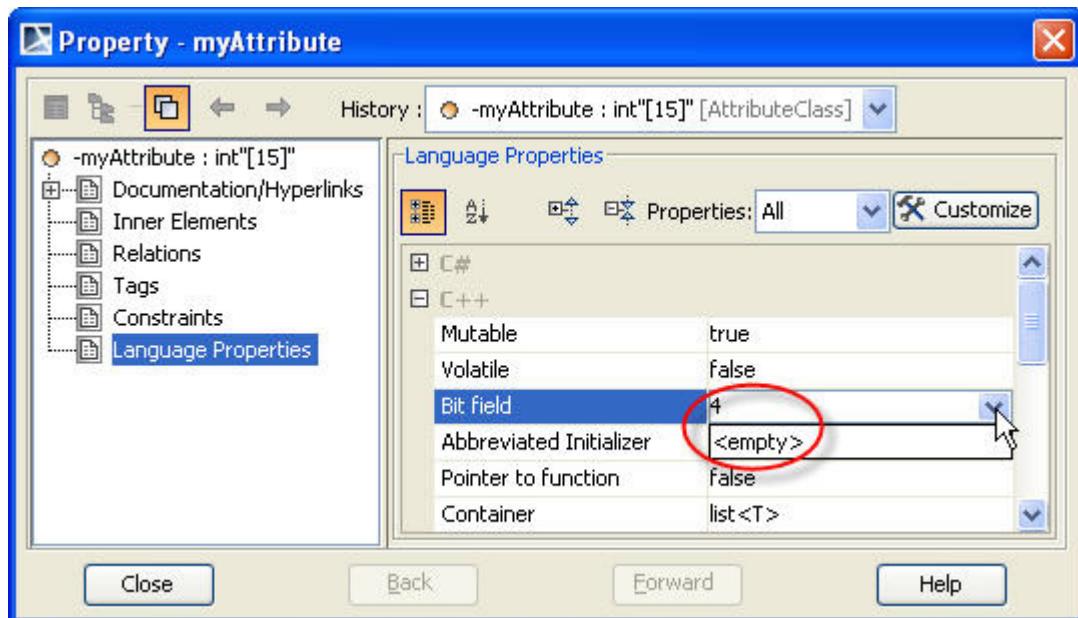
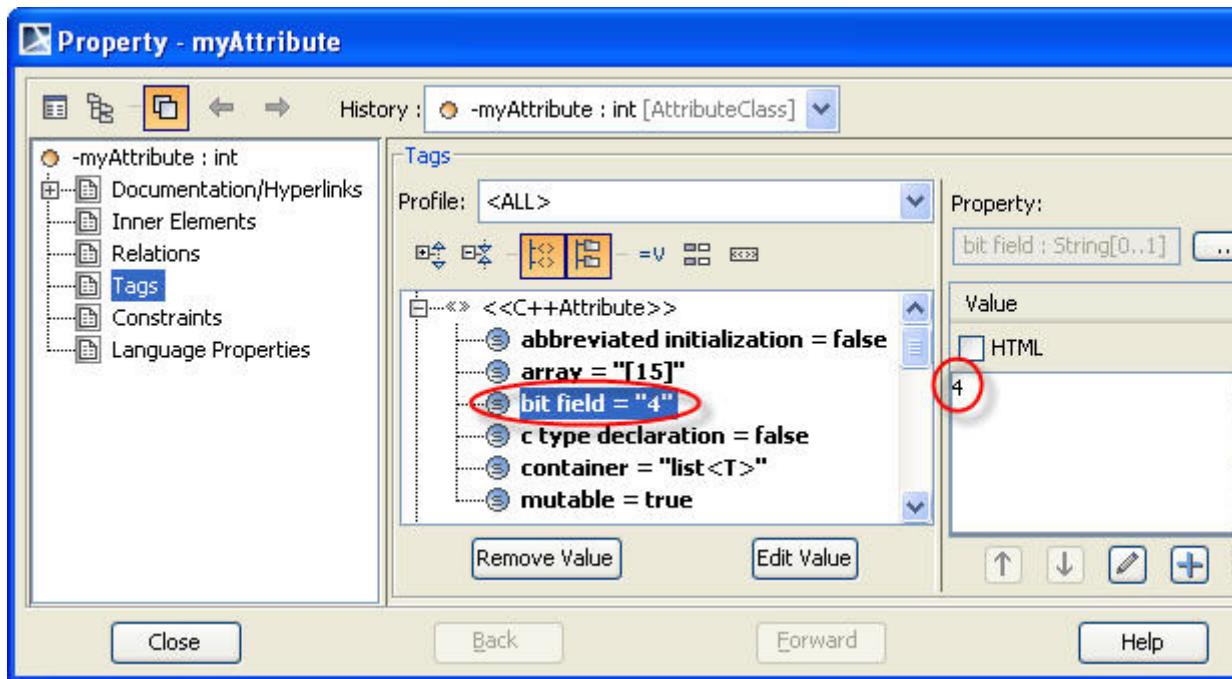


Figure 28 -- Attribute Bit field

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



Attribute - Abbreviated initializer

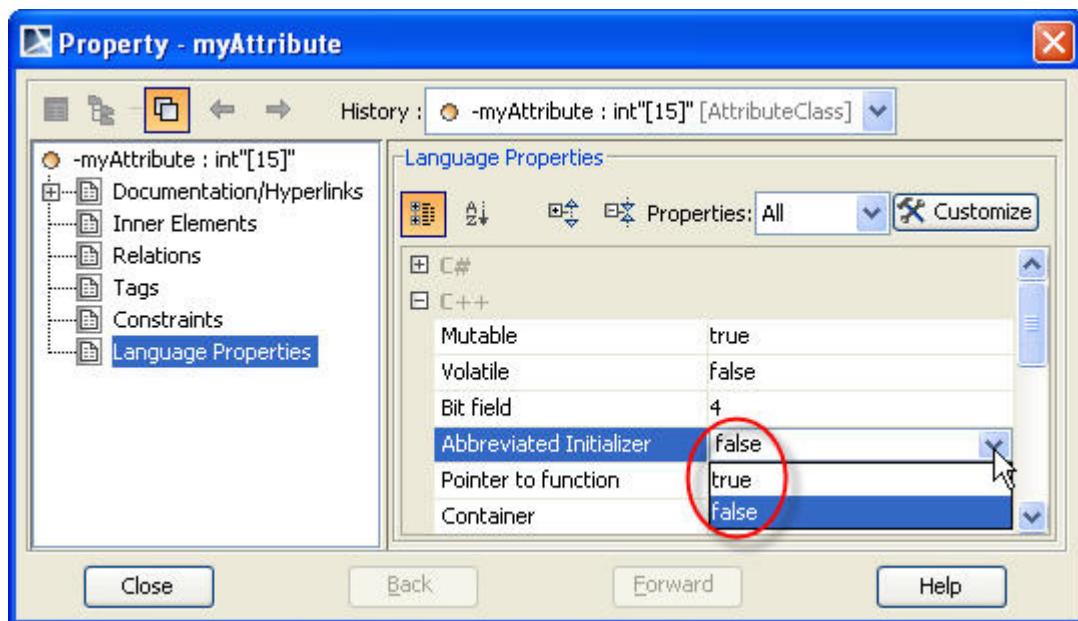
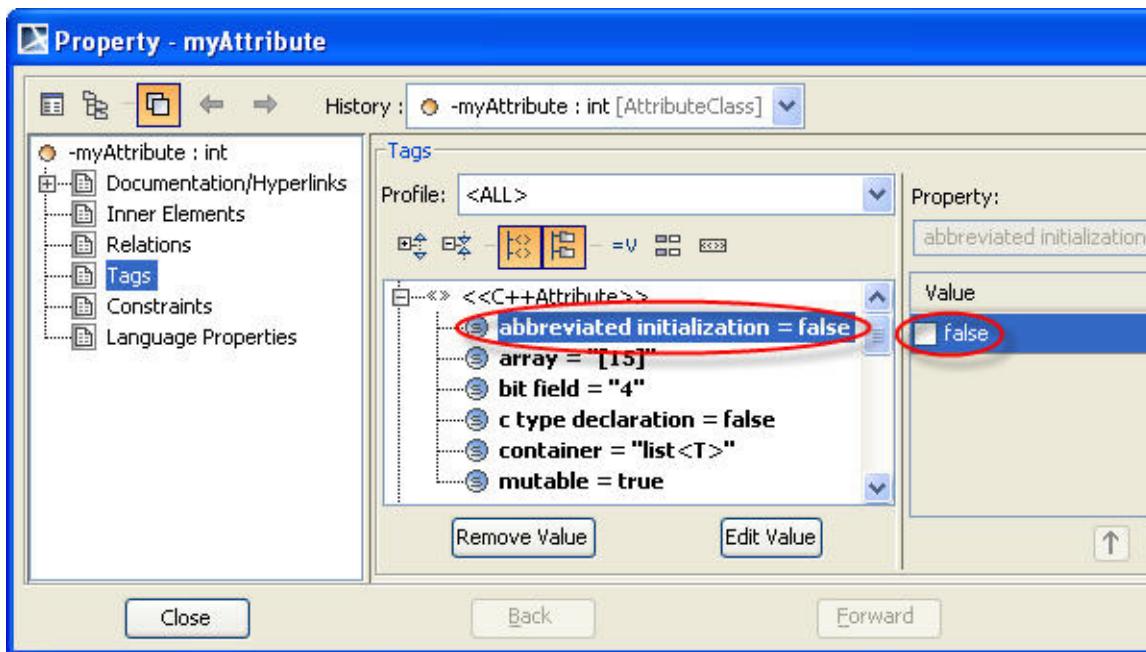


Figure 29 -- Attribute Abbreviated Initializer

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



Attribute - Pointer to function

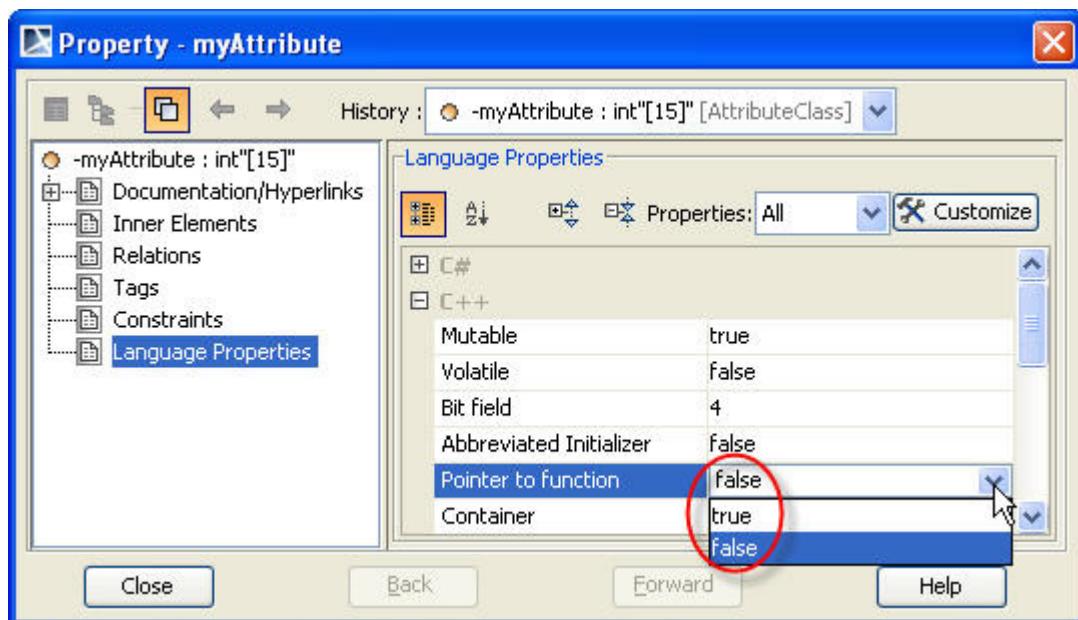


Figure 30 -- Attribute Pointer to function

Old value	Translation
false	no change.
true	no change.

Attribute - Container

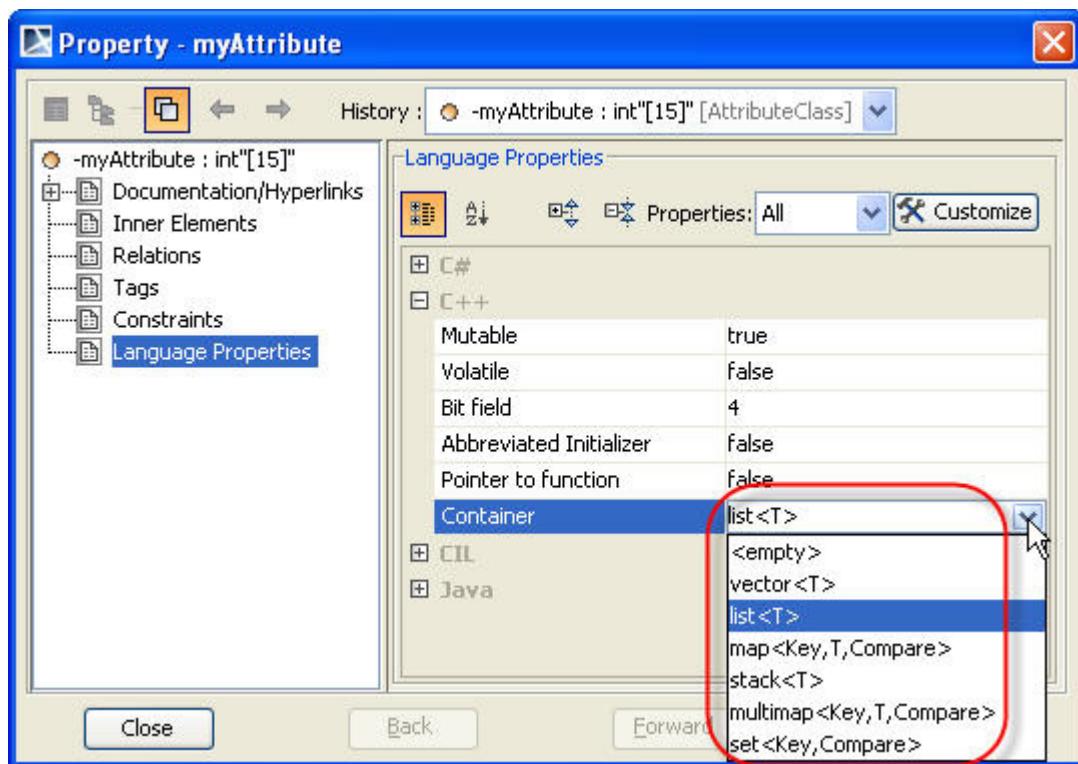
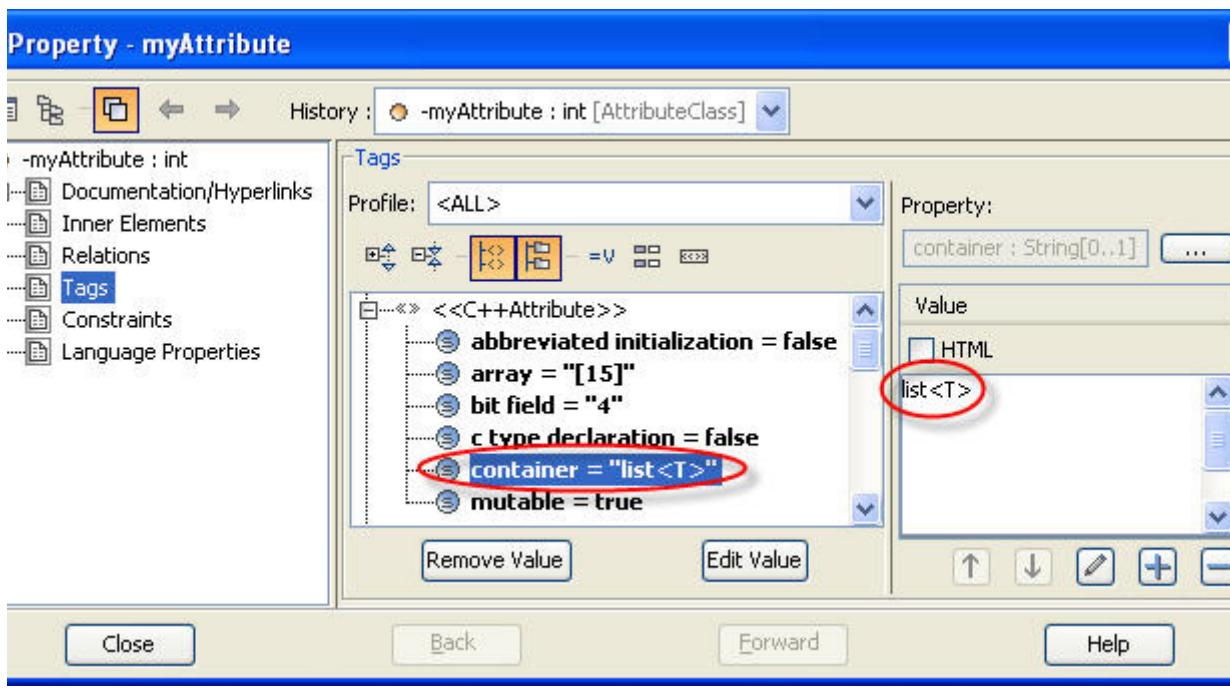


Figure 31 -- Attribute Container

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

set<Key, Compare>

Apply the <<C++Attribute>> stereotype and set container tag value to set<Key, Compare>.



Generalization

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

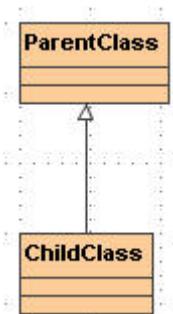


Figure 32 -- Generalization Example in Class Diagram

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

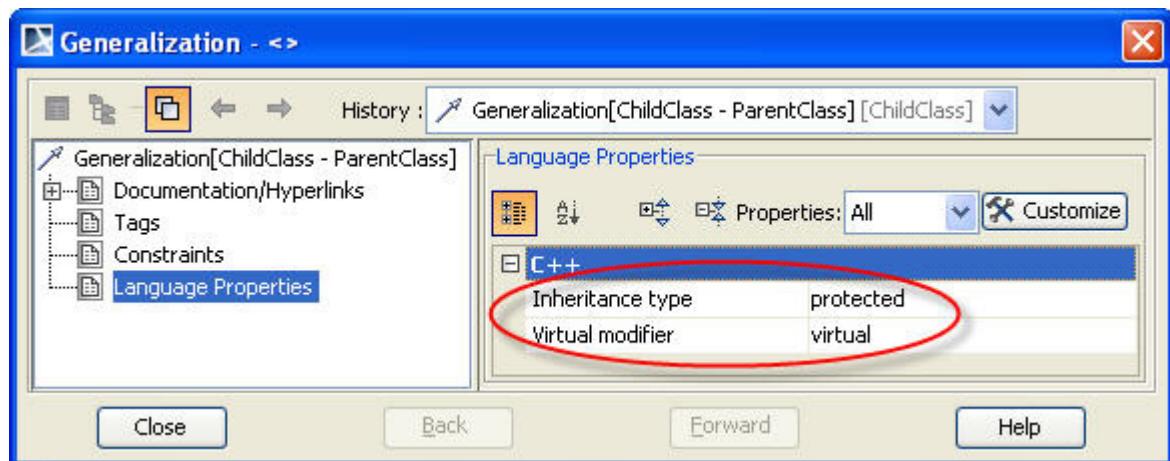


Figure 33 -- Generalization Language Properties

Generalization - Inheritance type

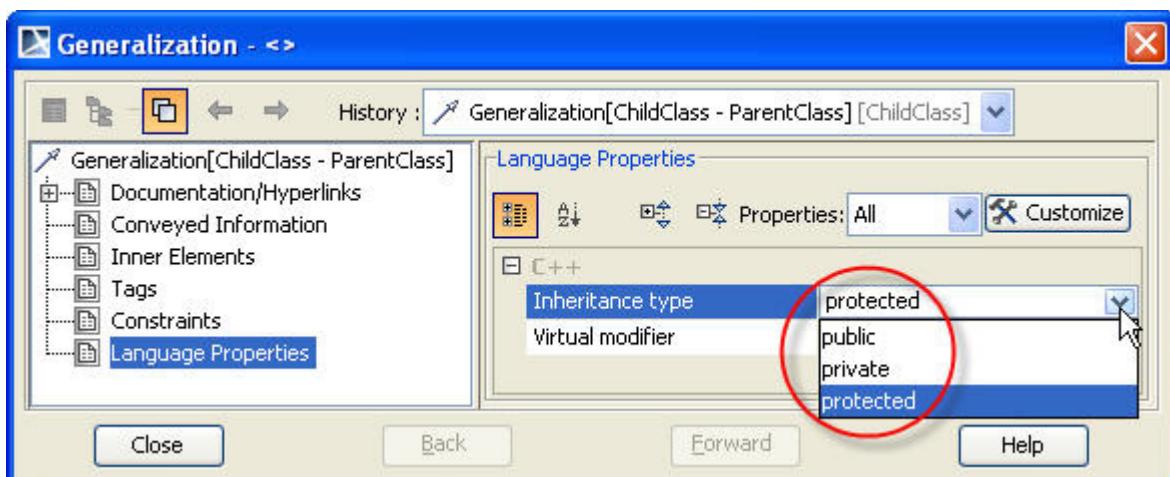
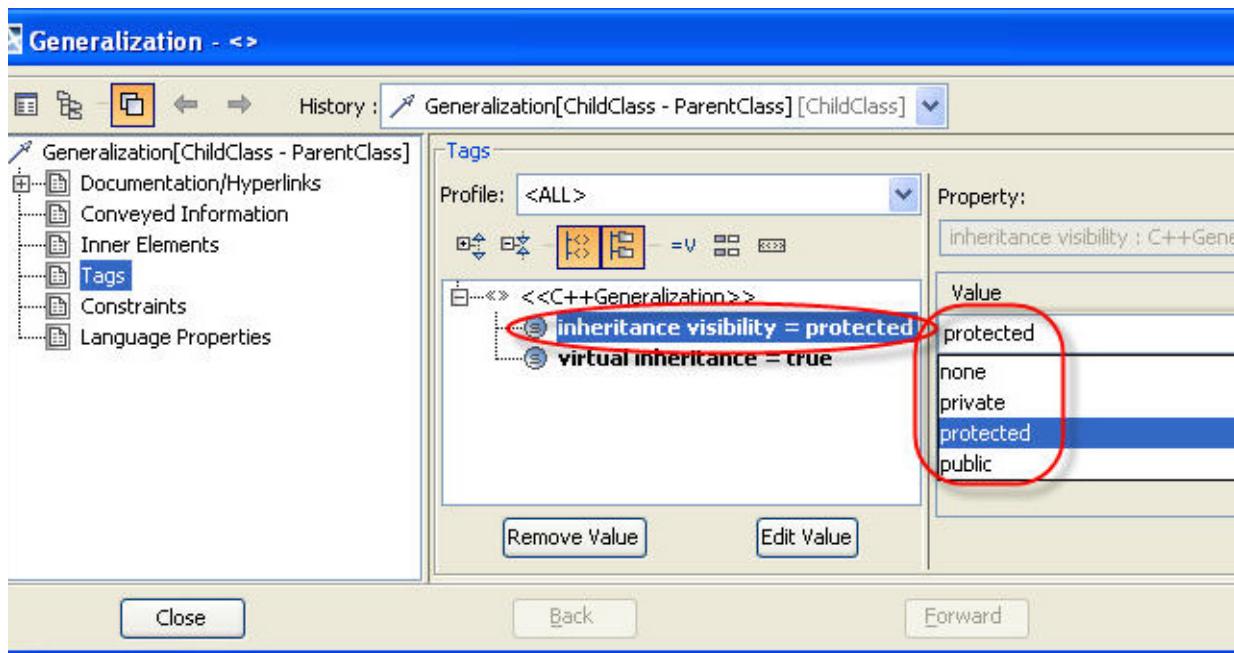


Figure 34 -- Generalization Inheritance type

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

protected	Apply the <<C++Generalization>> stereotype and set inheritance visibility tag value to protected.
private	Apply the <<C++Generalization>> stereotype and set inheritance visibility tag value to private.



Generalization - Virtual modifier

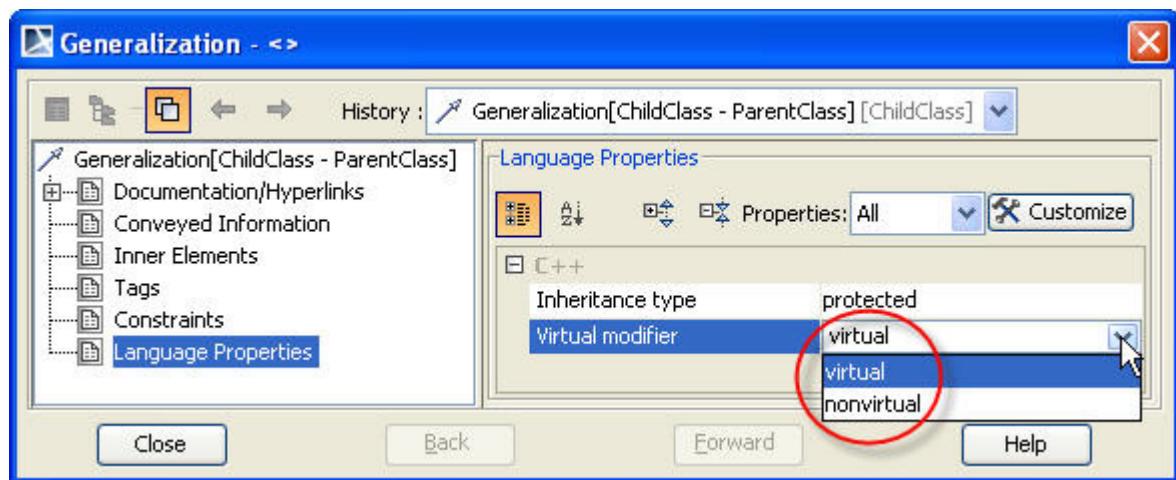
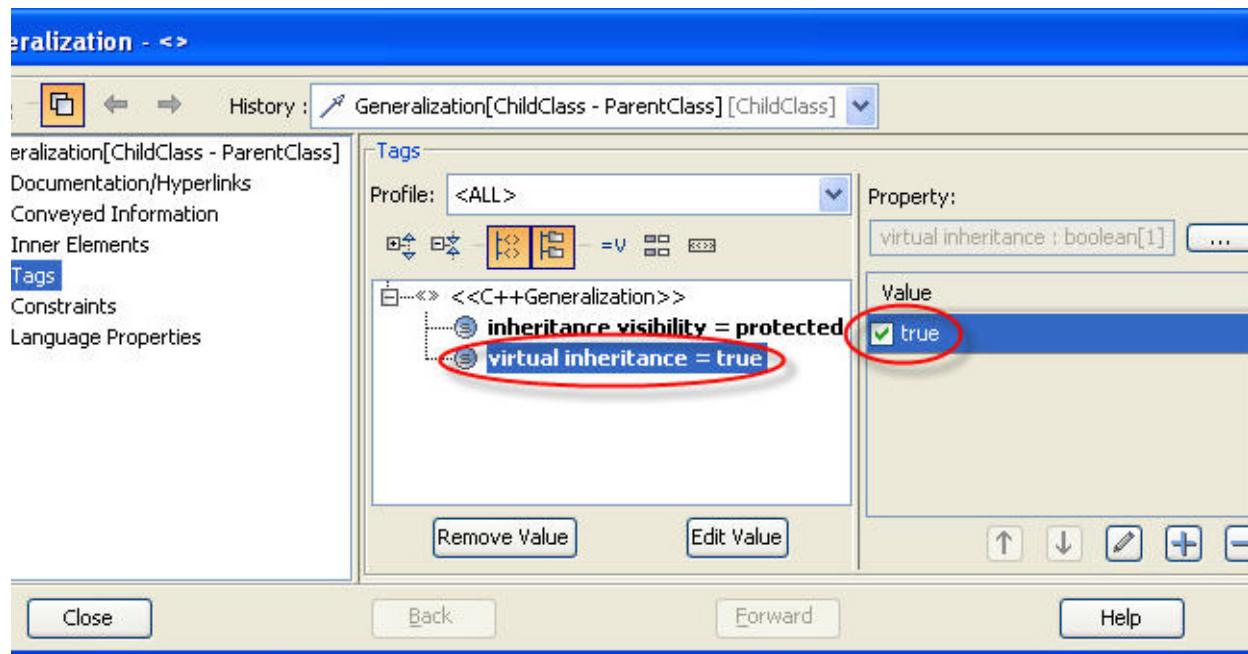


Figure 35 -- Generalization Virtual modifier

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



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.

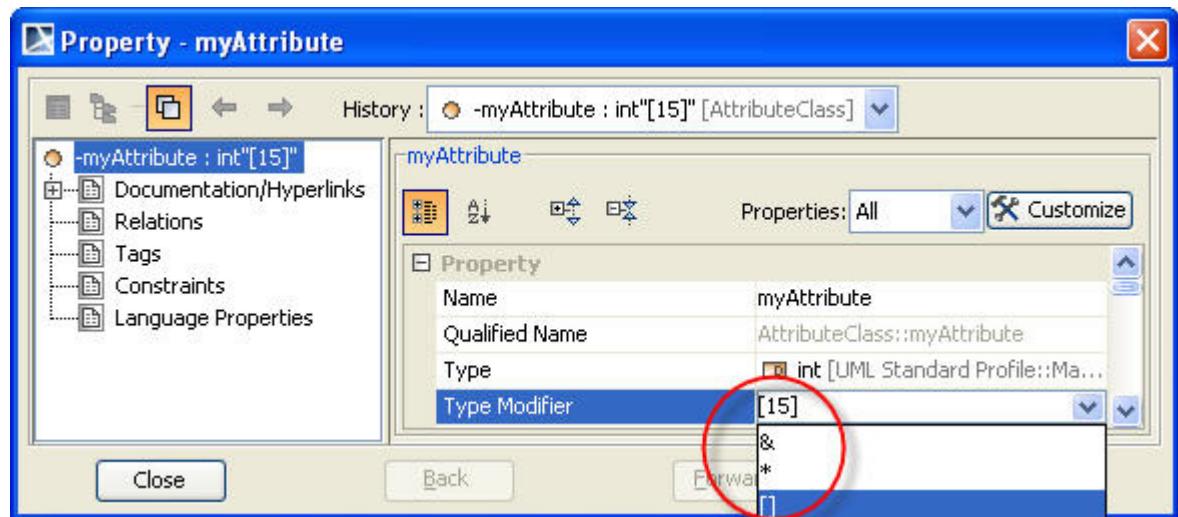


Figure 36 -- Array type modifier in Attribute

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

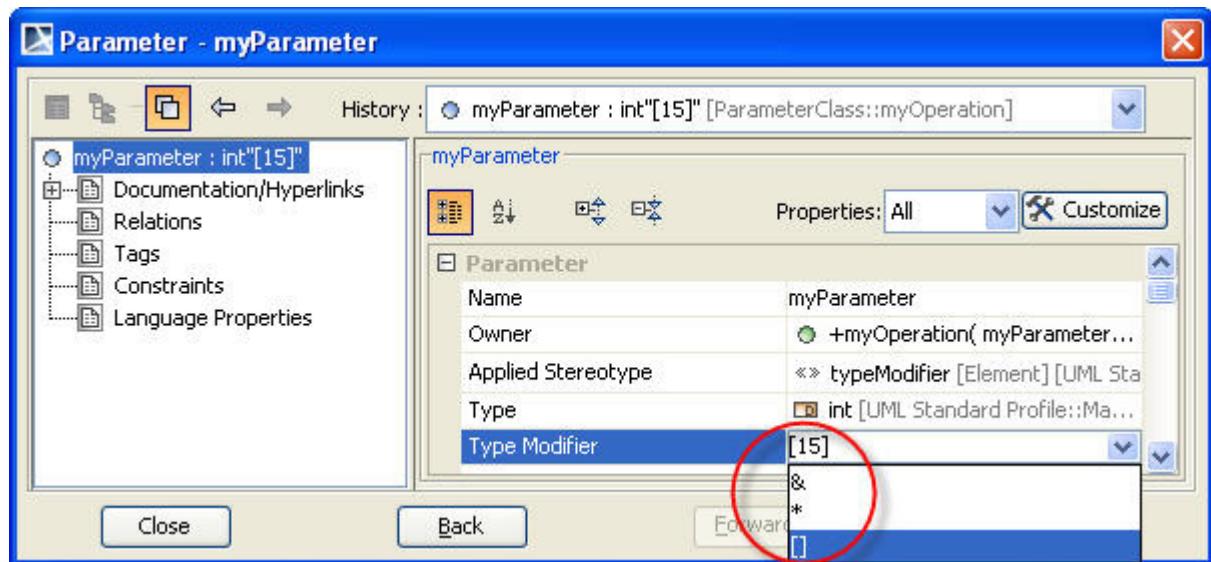
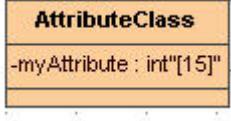
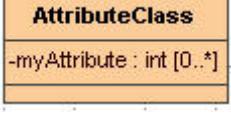
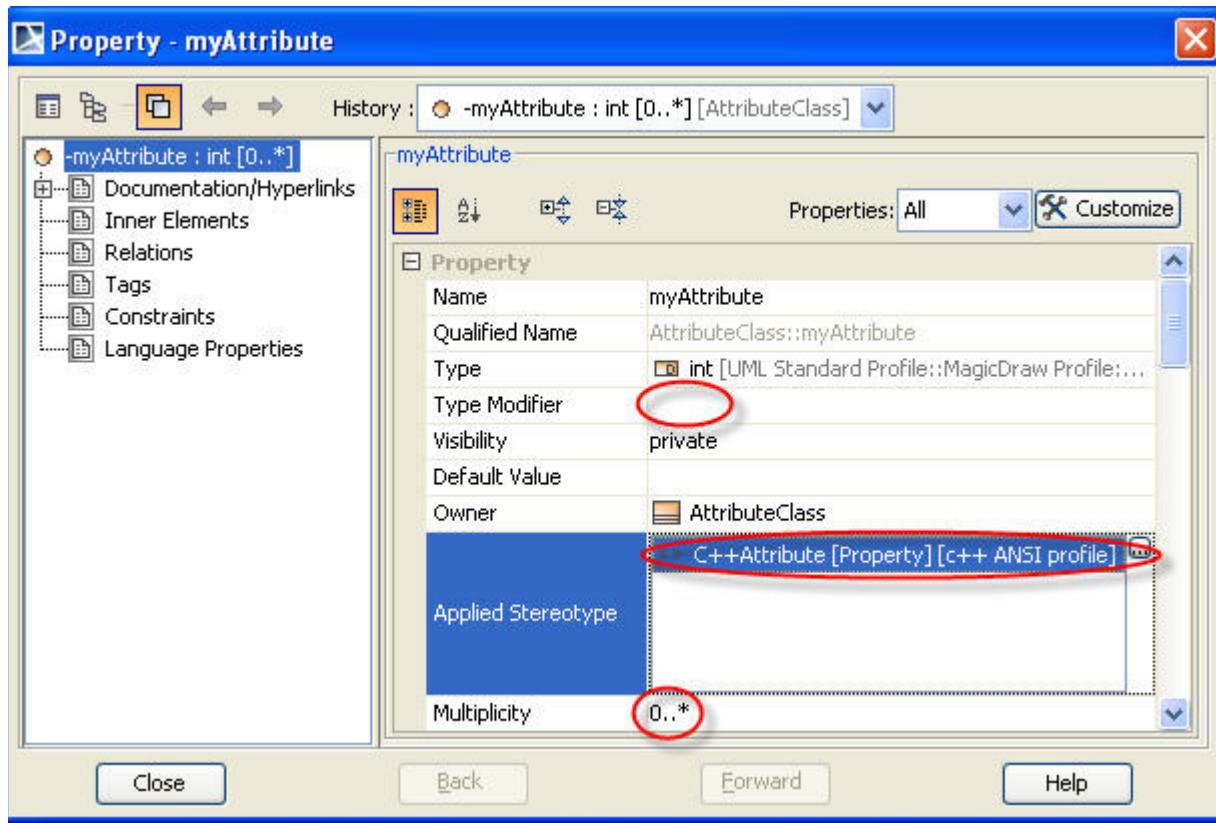
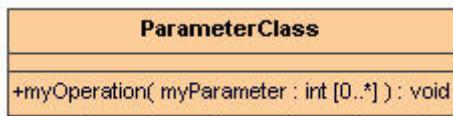
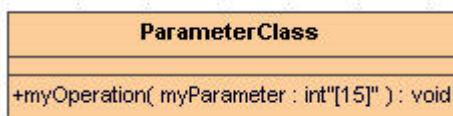
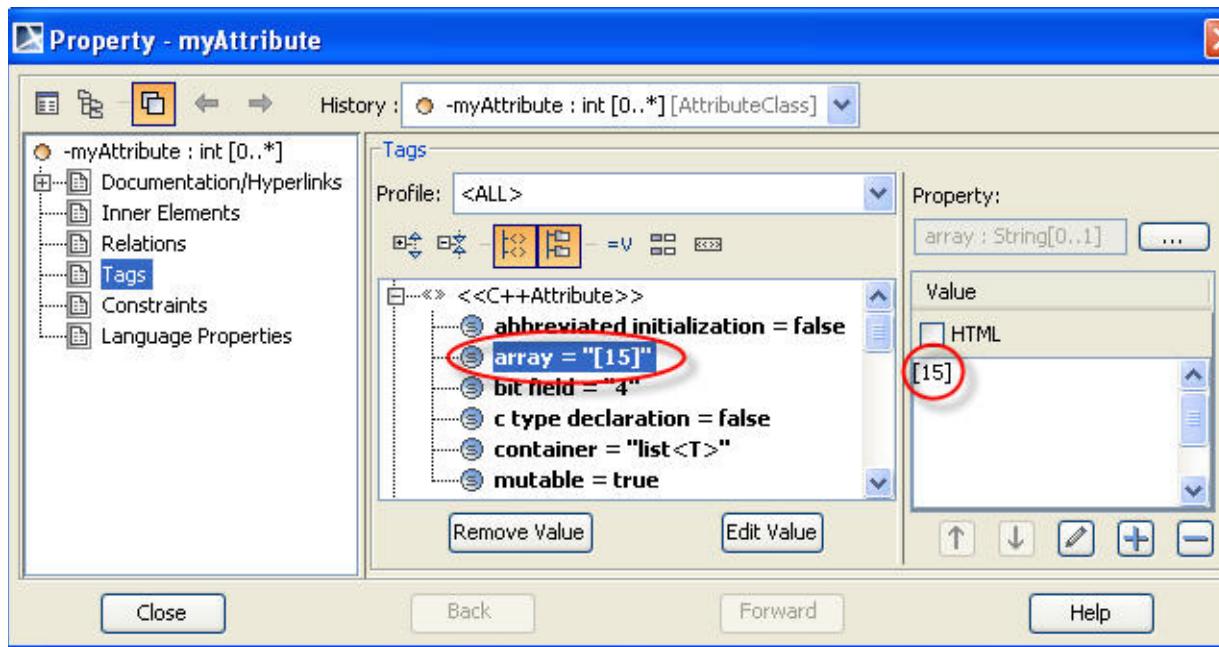


Figure 37 -- Array type modifier in Parameter

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



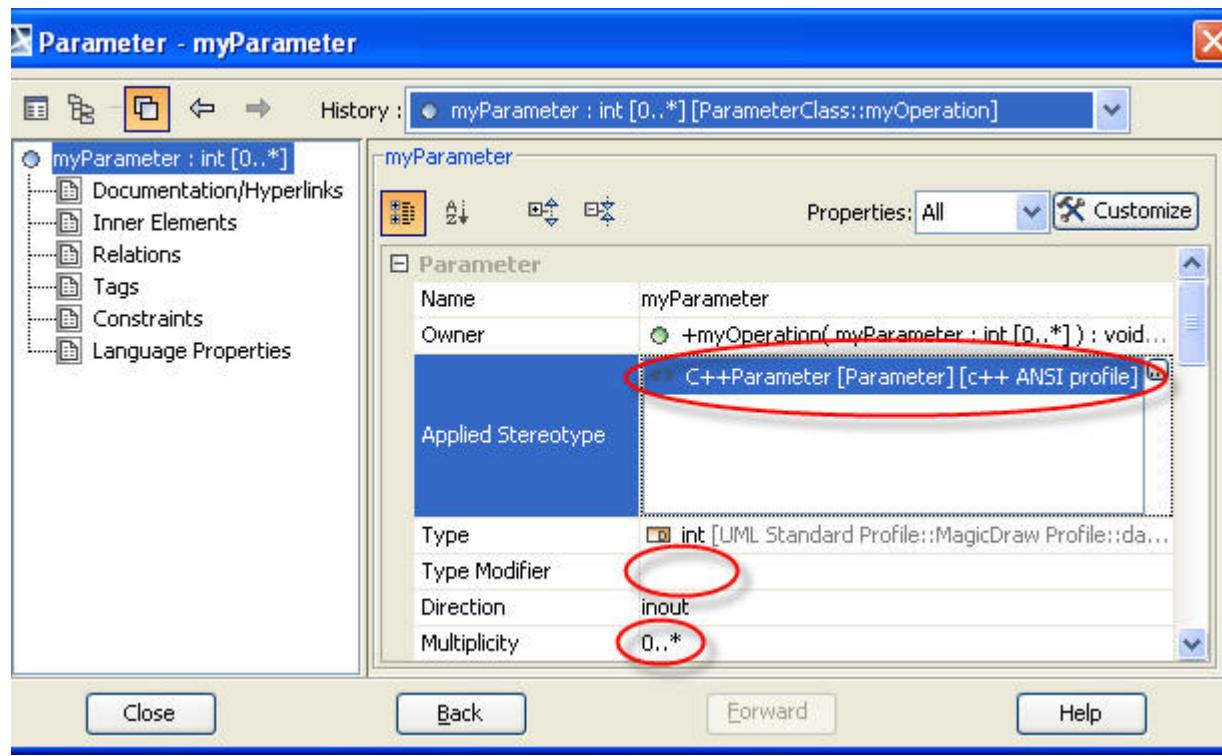


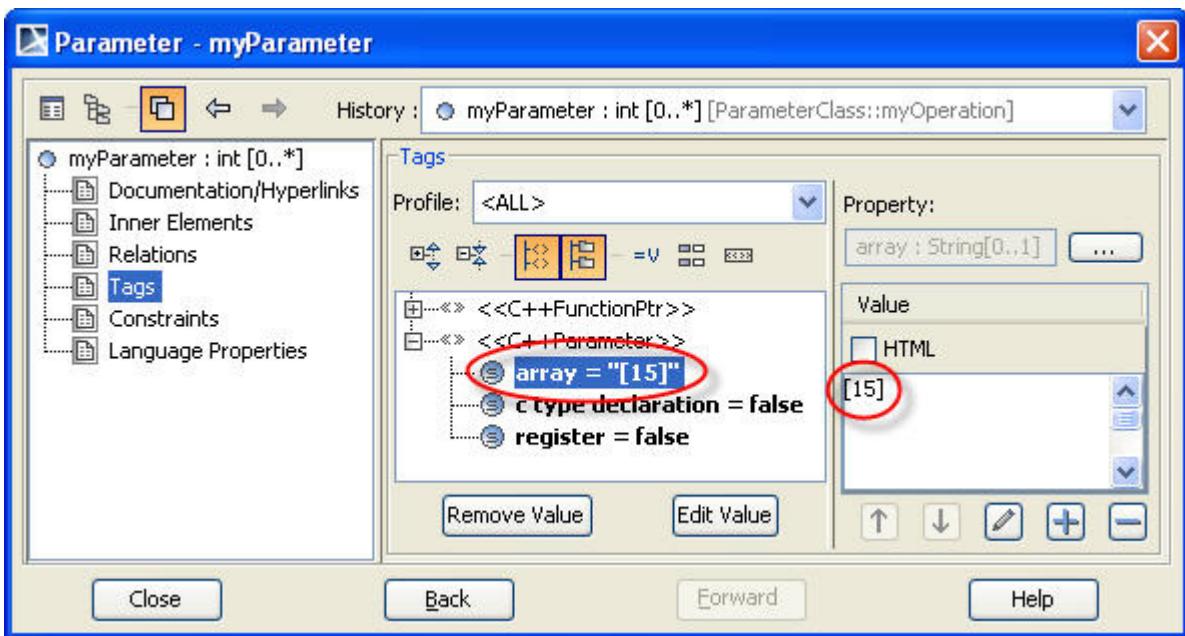
[15]

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**.

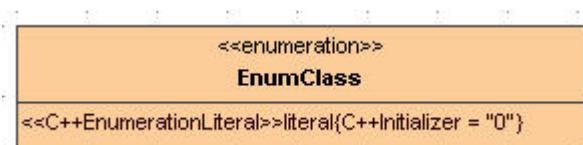


Figure 38 -- <<C++EnumerationLiteral>> stereotype Example in Class Diagram

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

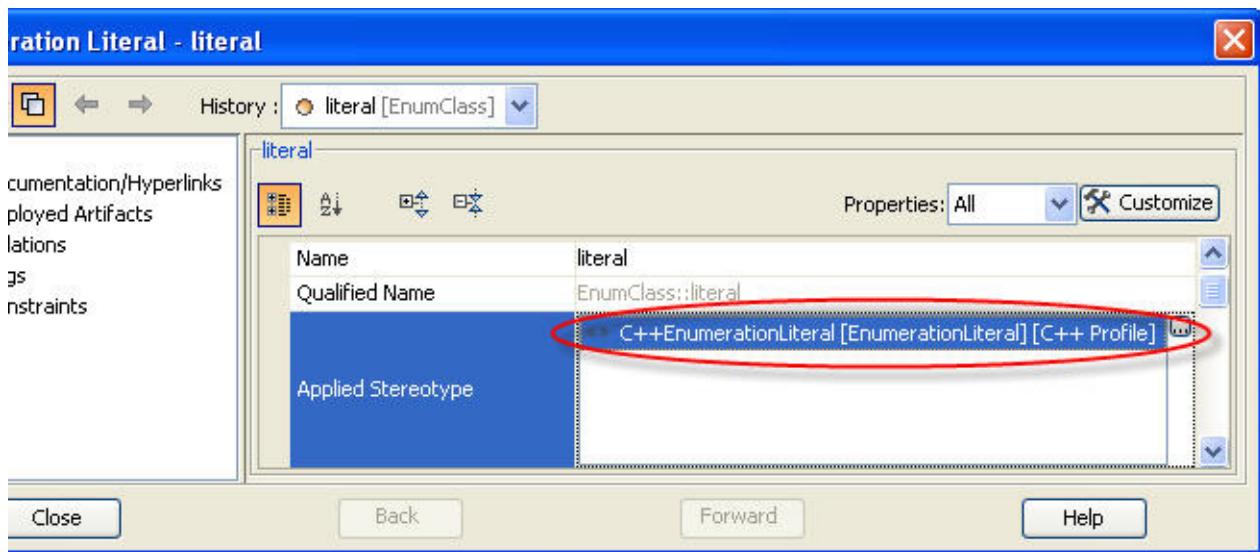


Figure 39 -- <<C++EnumerationLiteral>> stereotype

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

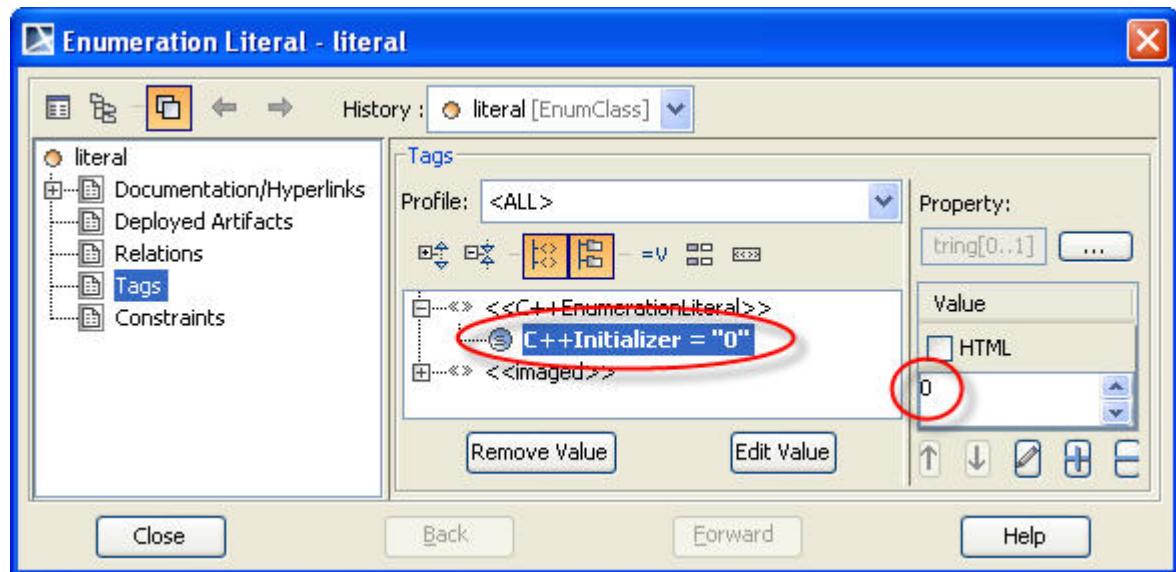
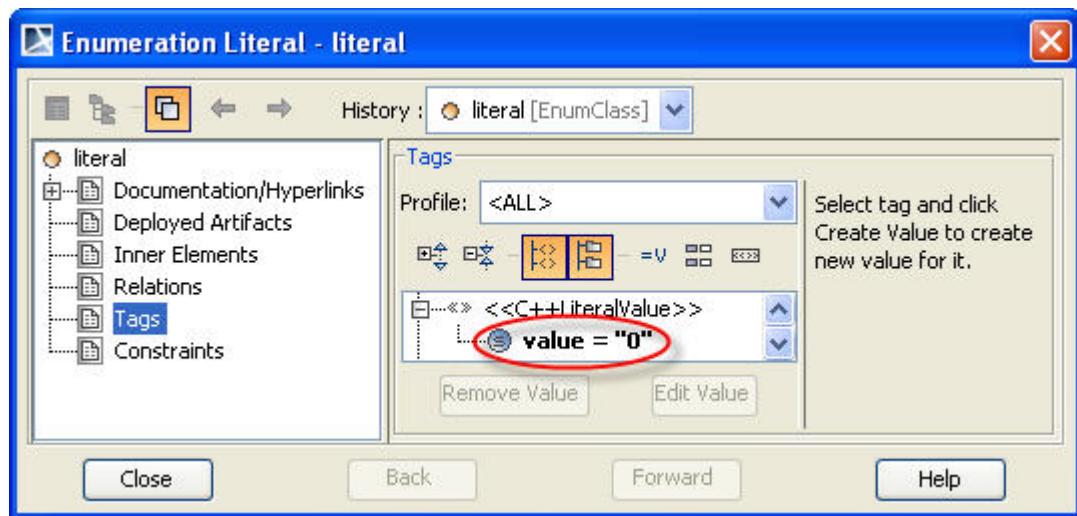
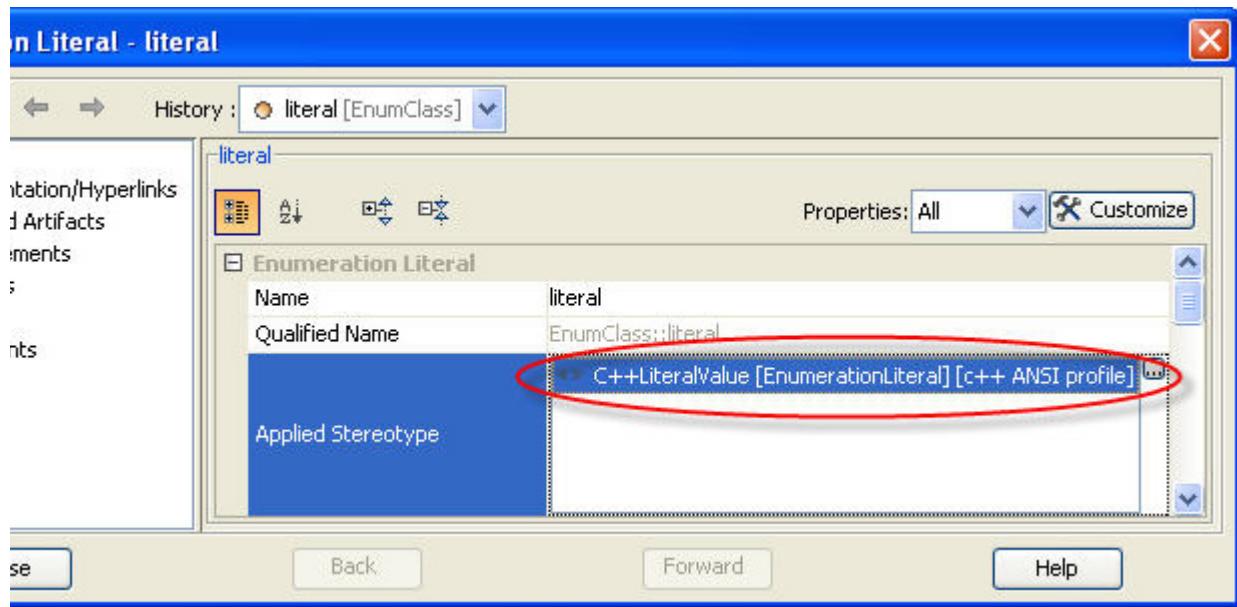


Figure 40 -- Enumeration Literal Tag Value

Old value	Translation
Enumeration Literal applies the <<C++EnumerationLiteral>> stereotype and sets C++Initializer tag value to 0.	Apply the <<C++LiteralValue>> and set value tag value to 0. Remove the <<C++EnumerationLiteral>> stereotype and C++Initializer tag value from Enumeration Literal.



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

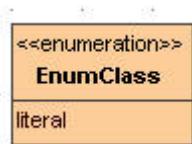


Figure 41 -- Translated Enumeration Literal in Class Diagram

<<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**.

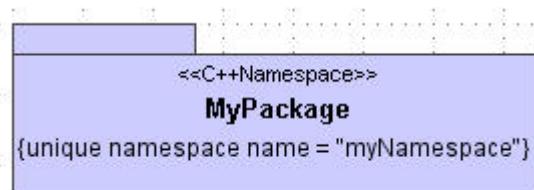


Figure 42 -- <<C++Namespace>> stereotype Example in Class Diagram

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

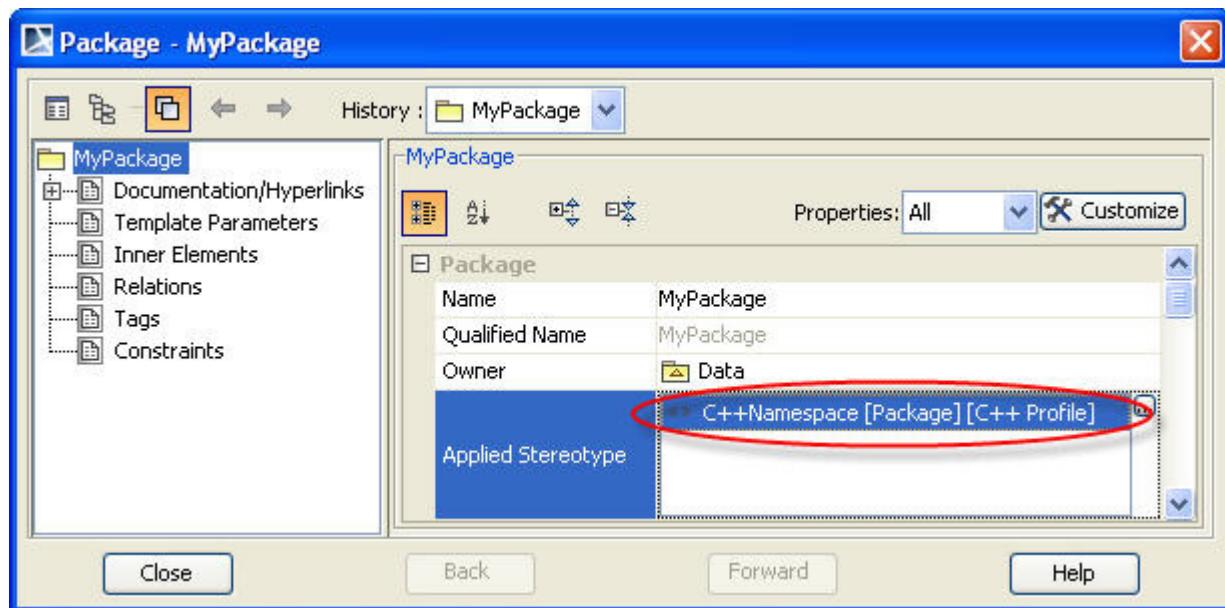


Figure 43 -- <<C++Namespace>> stereotype

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

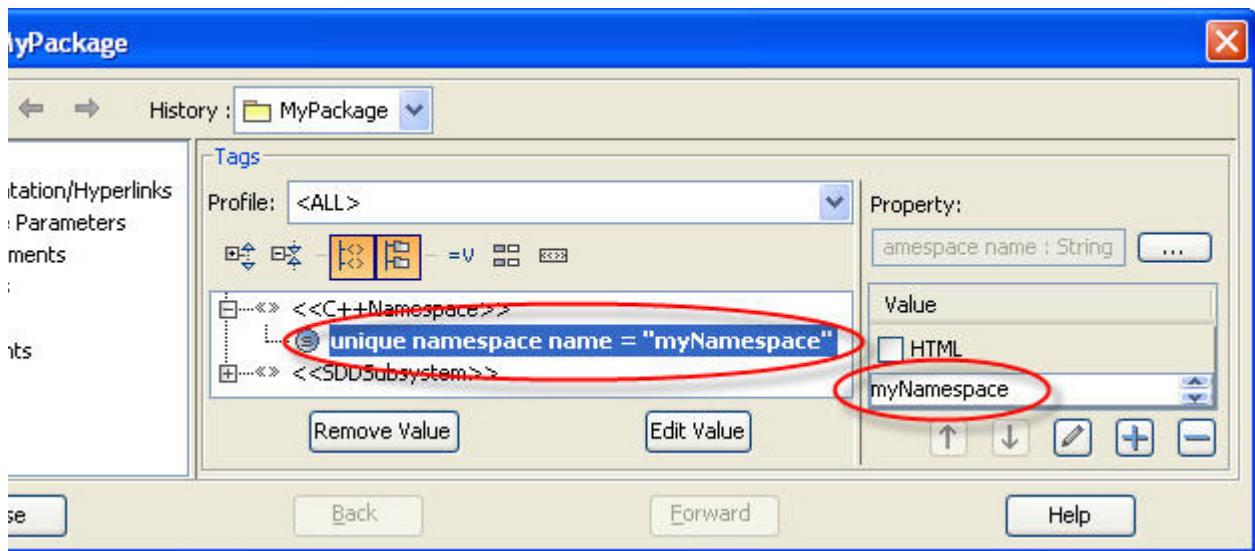
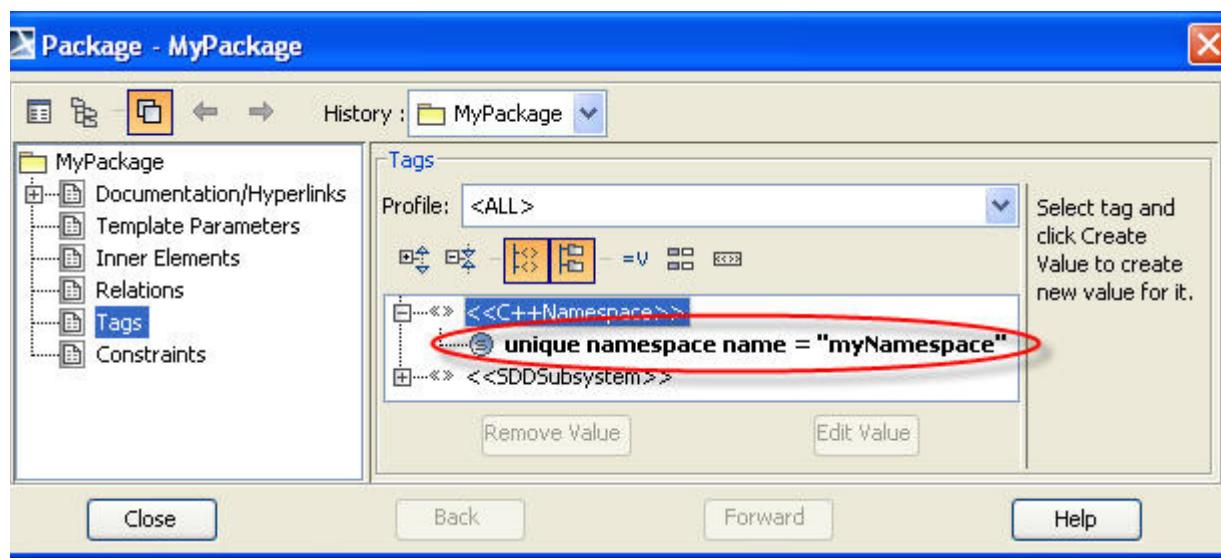
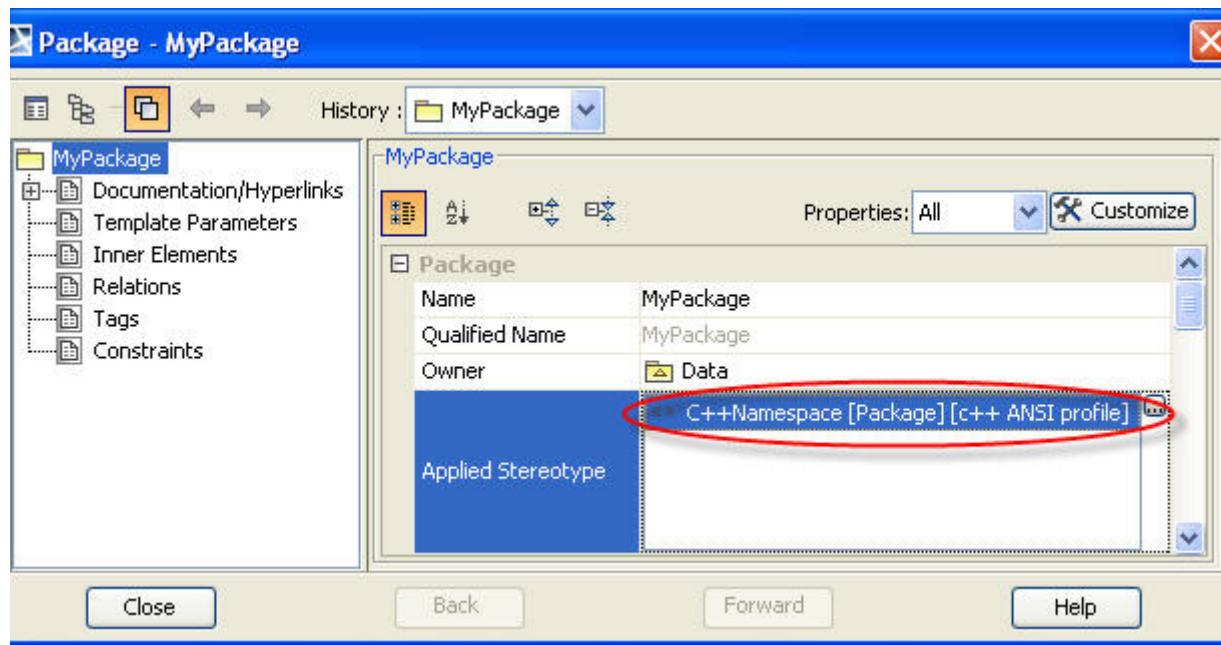


Figure 44 -- unique namespace name tag value

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



<<C++Typename>>

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

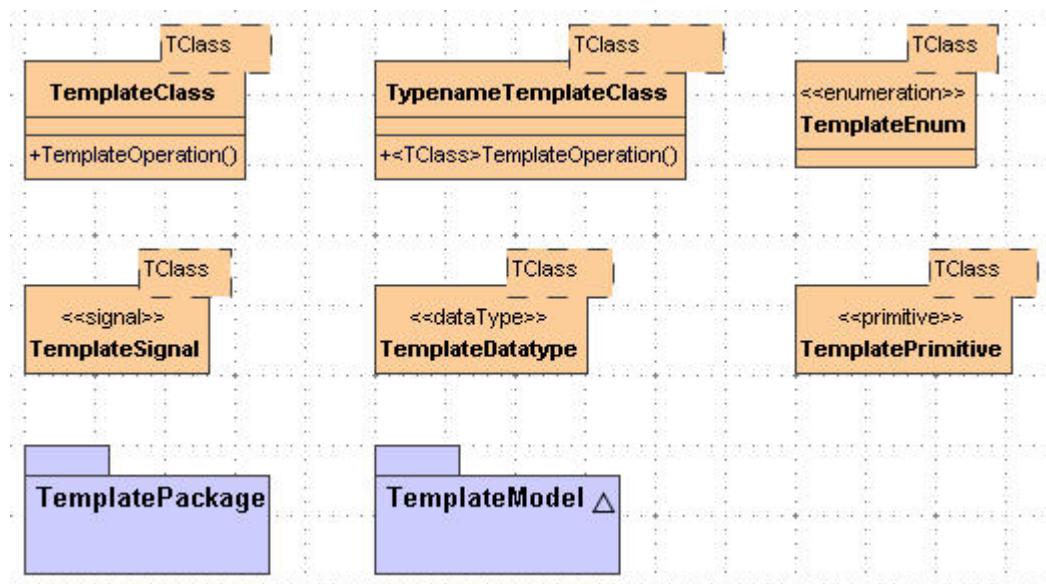


Figure 45 -- Elements that can have template parameters

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**.

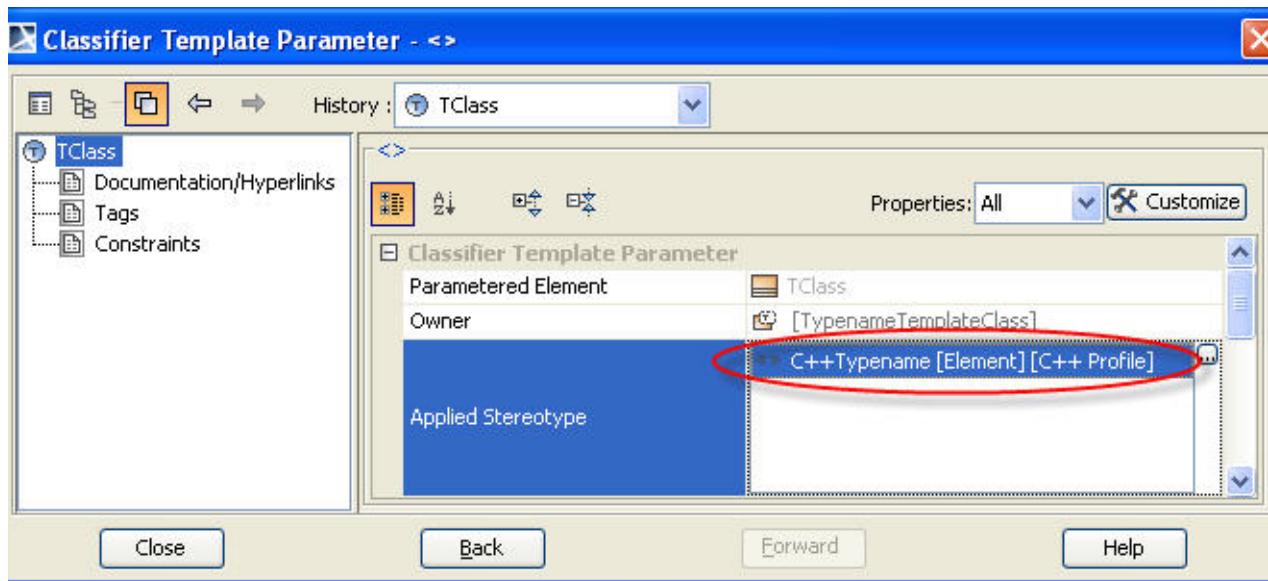


Figure 46 -- <<C++Typename>> stereotype

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

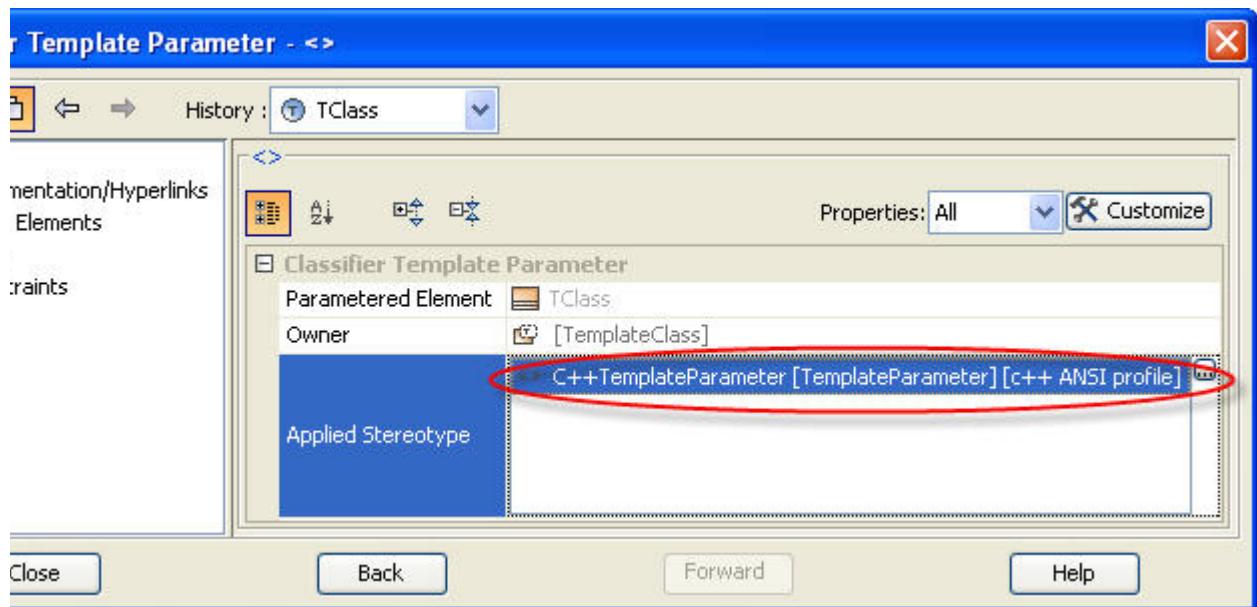
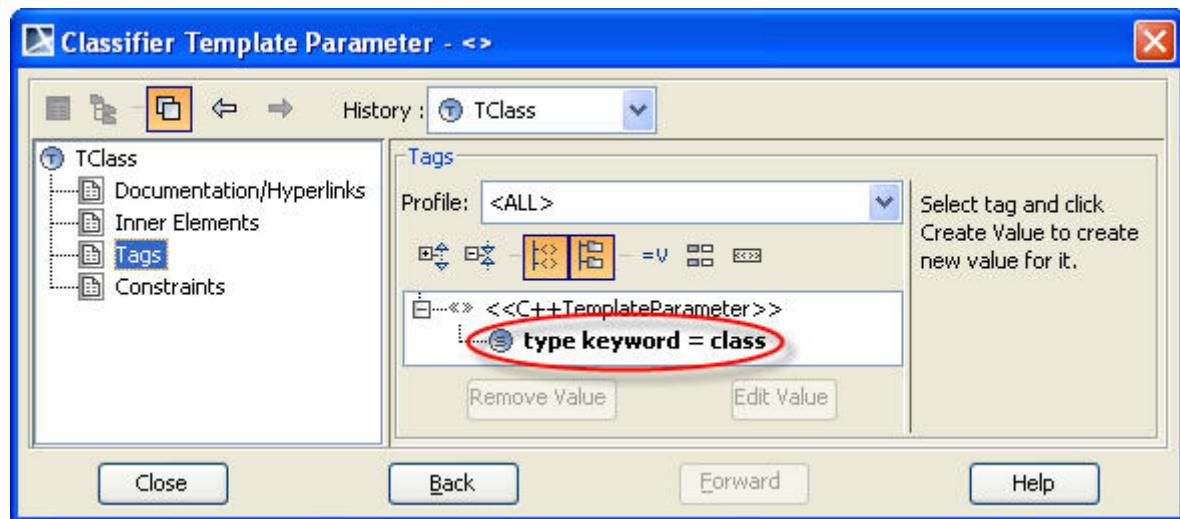


Figure 47 -- <<C++TemplateParamater>> stereotype

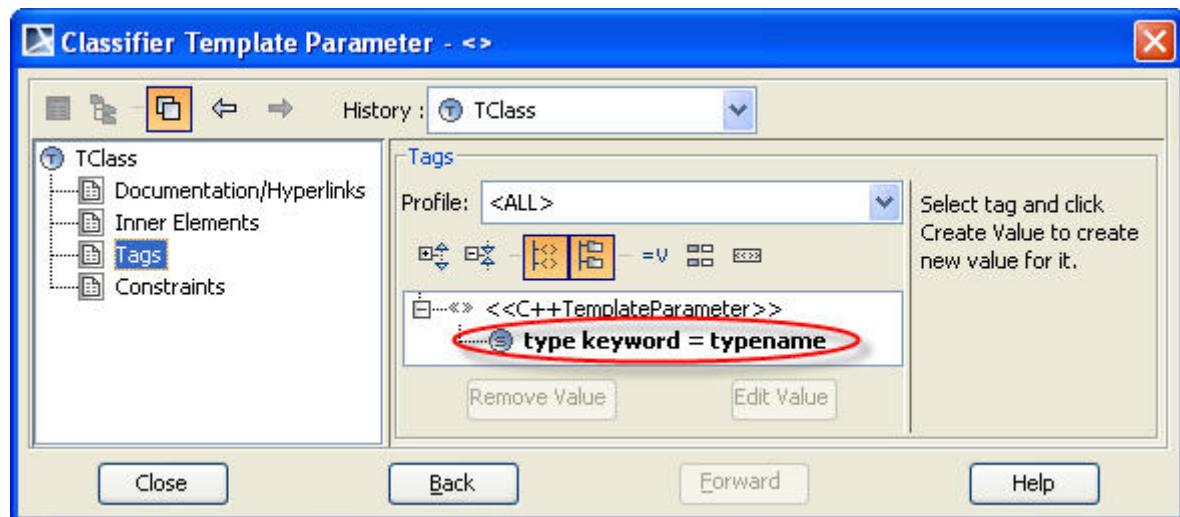
Old value	Translation
Template Parameter does not apply the <<C++Typename>> stereotype.	Apply the <<C++TemplateParameter>> stereotype and set type keyword tag value to class .



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.

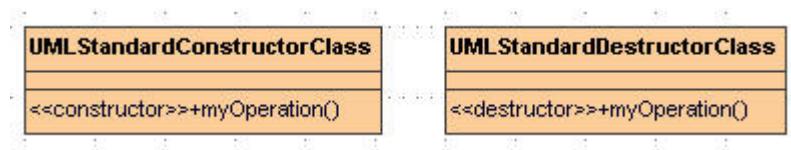


Figure 48 -- <<constructor>> and <<destructor>> stereotype Example in Class Diagram

<<constructor>>

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

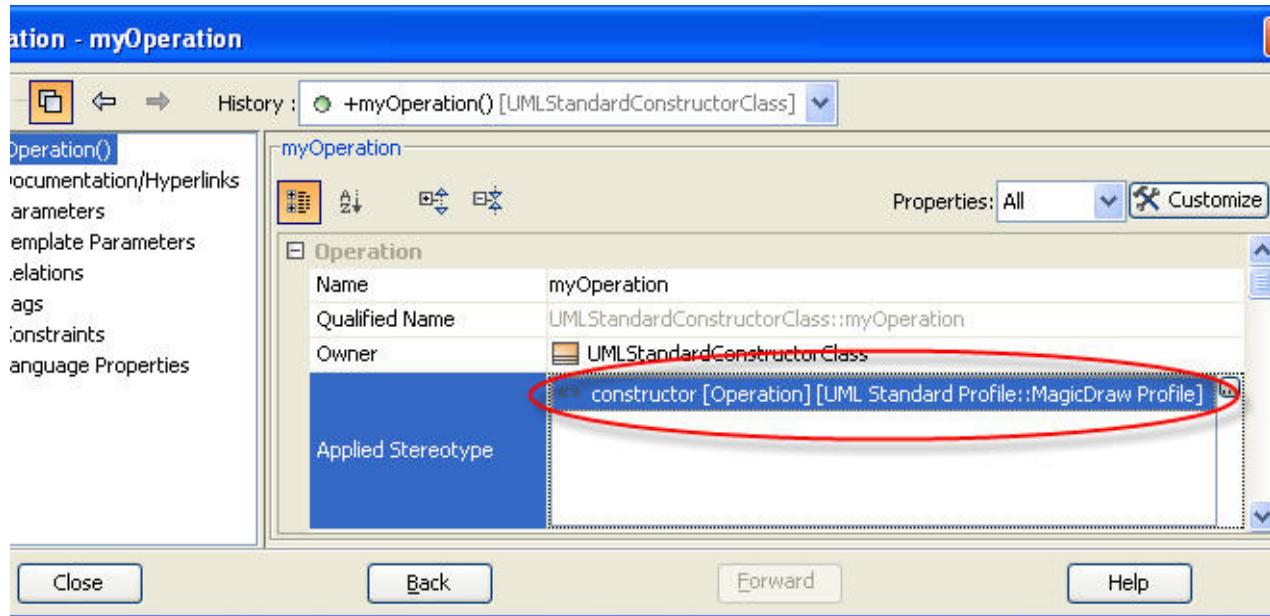
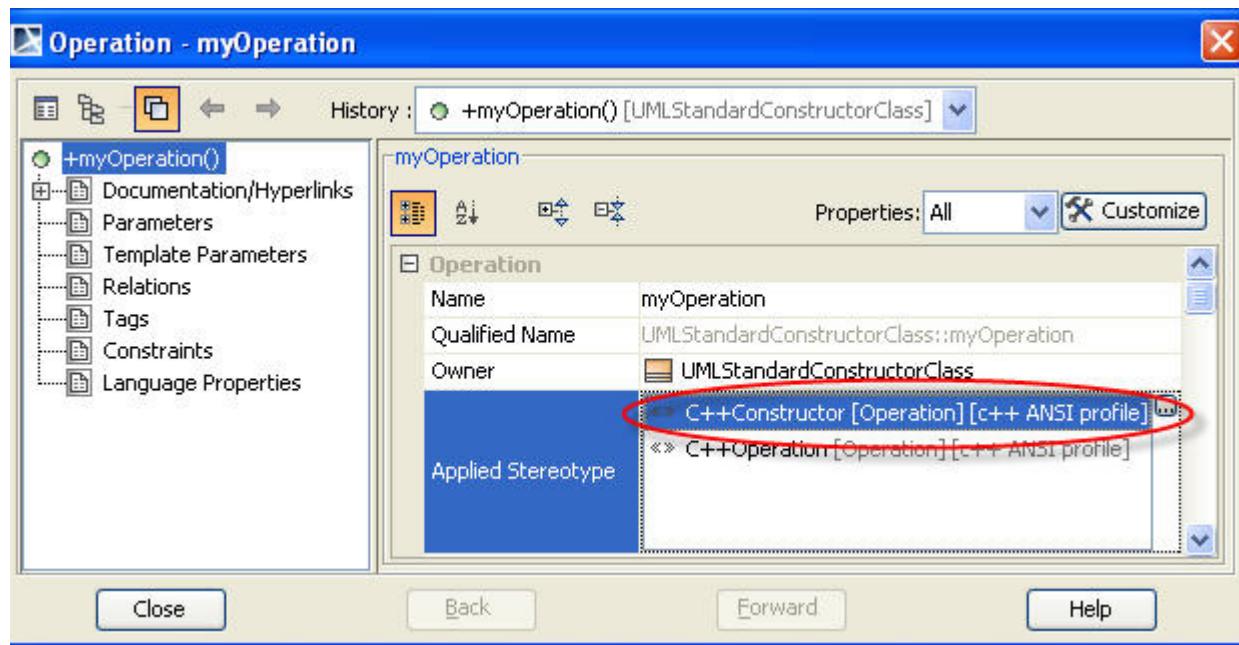


Figure 49 -- <<constructor>> stereotype

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.

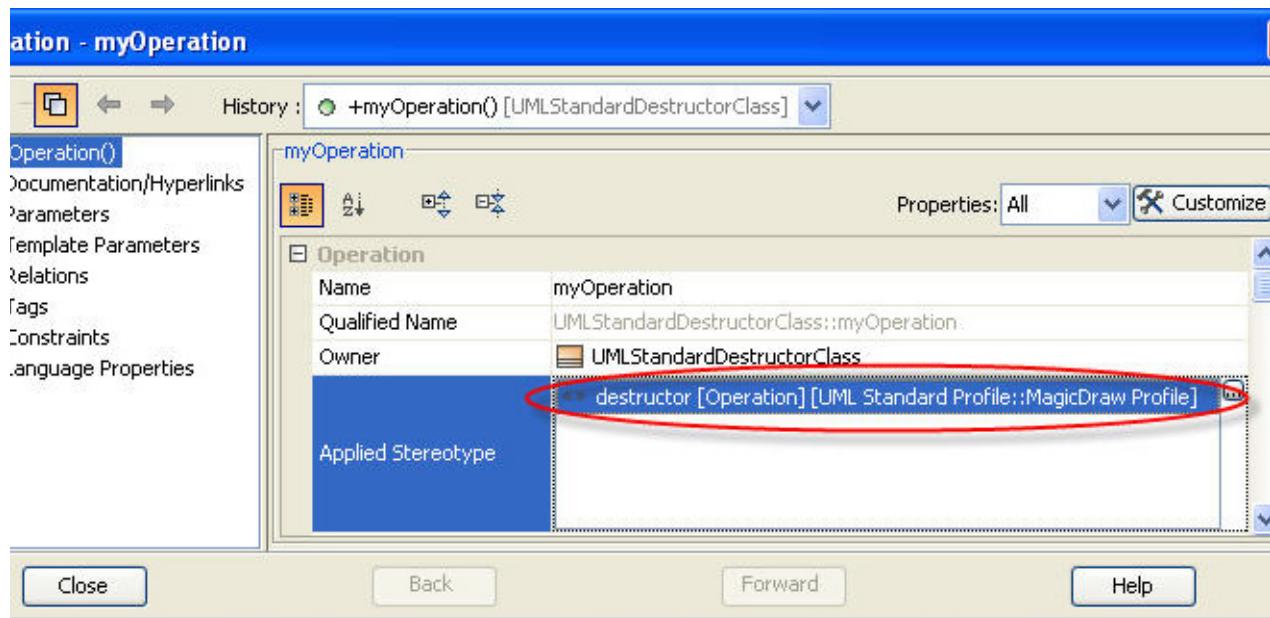
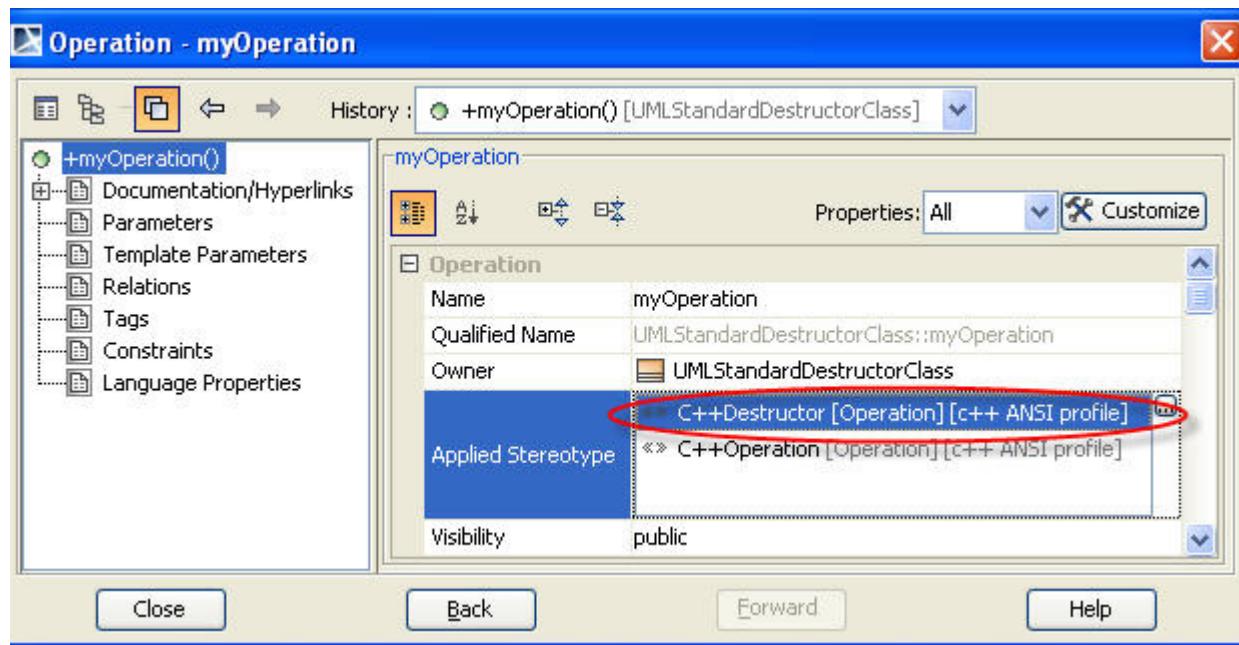


Figure 50 -- <<destructor>> stereotype

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.

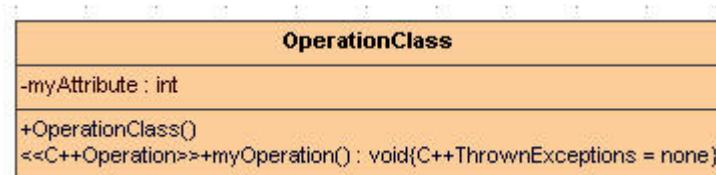


Figure 51 -- C++ThrownExceptions Example in Class Diagram

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.

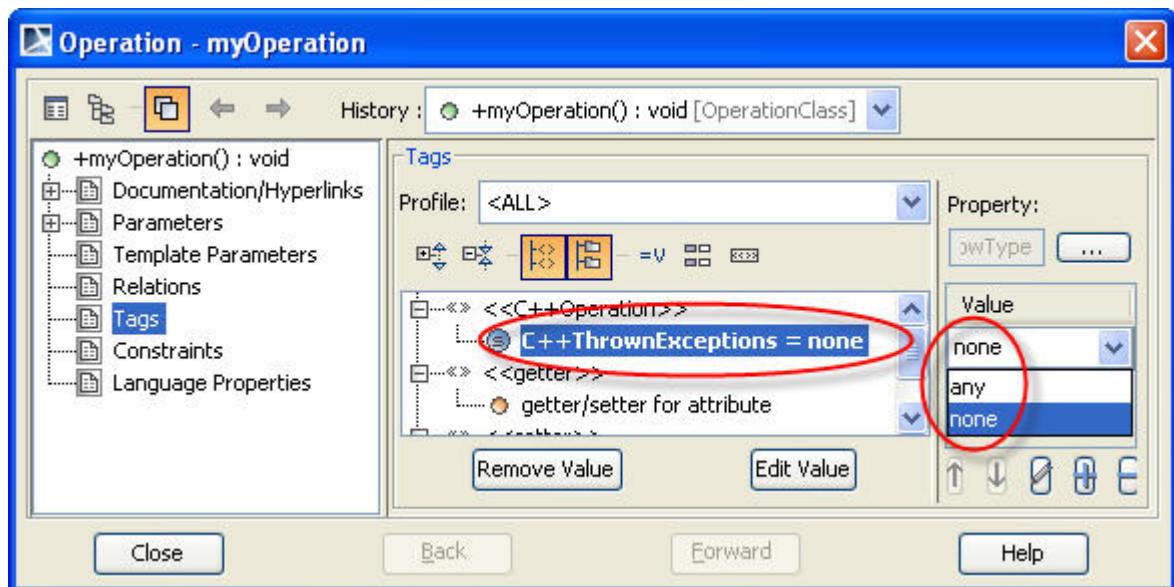
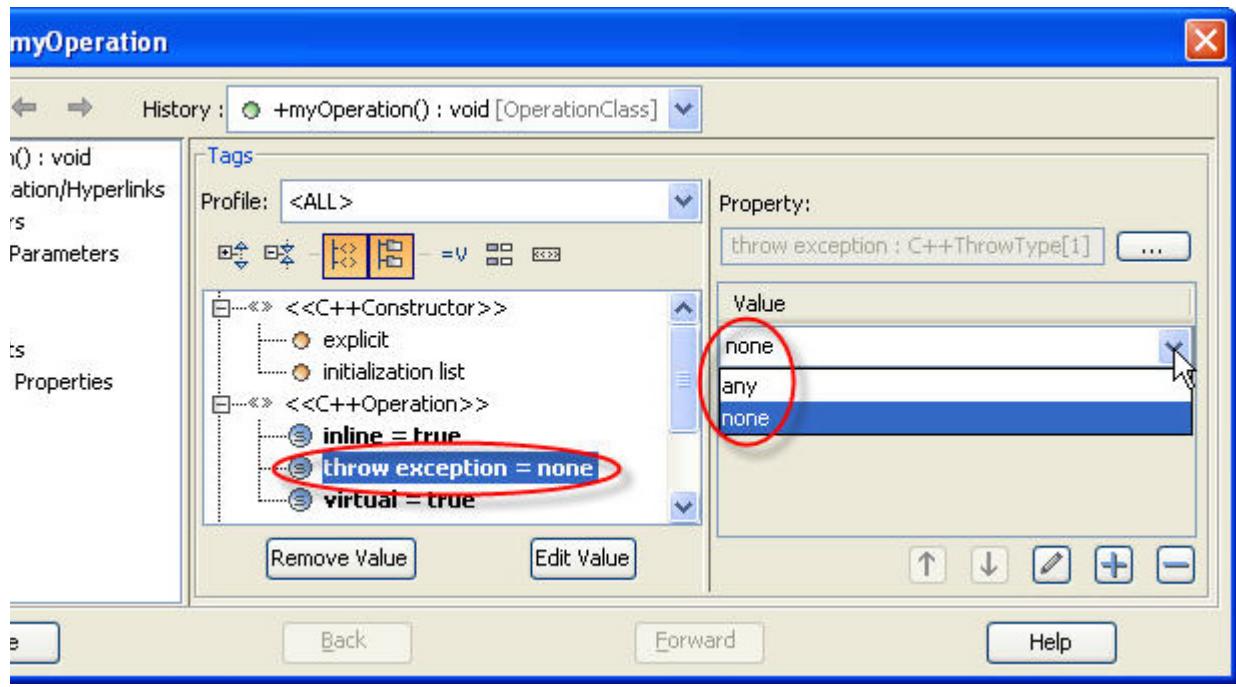


Figure 52 -- C++ThrownExceptions Tag Value

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

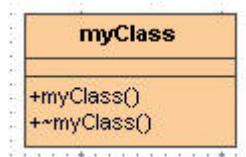


Figure 53 -- Constructor and Destructor name Example in Class Diagram

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

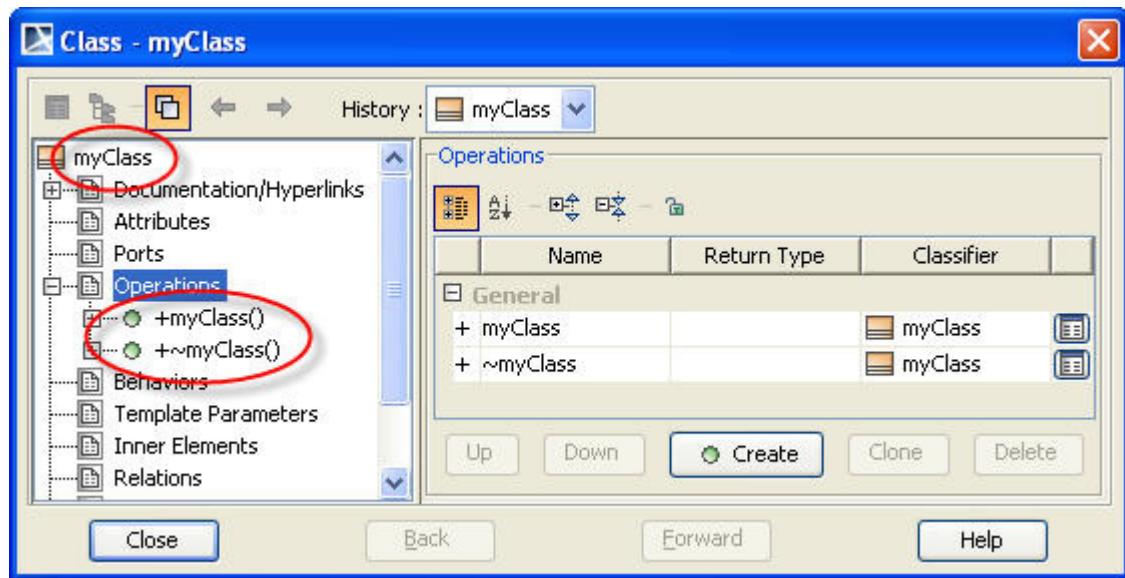
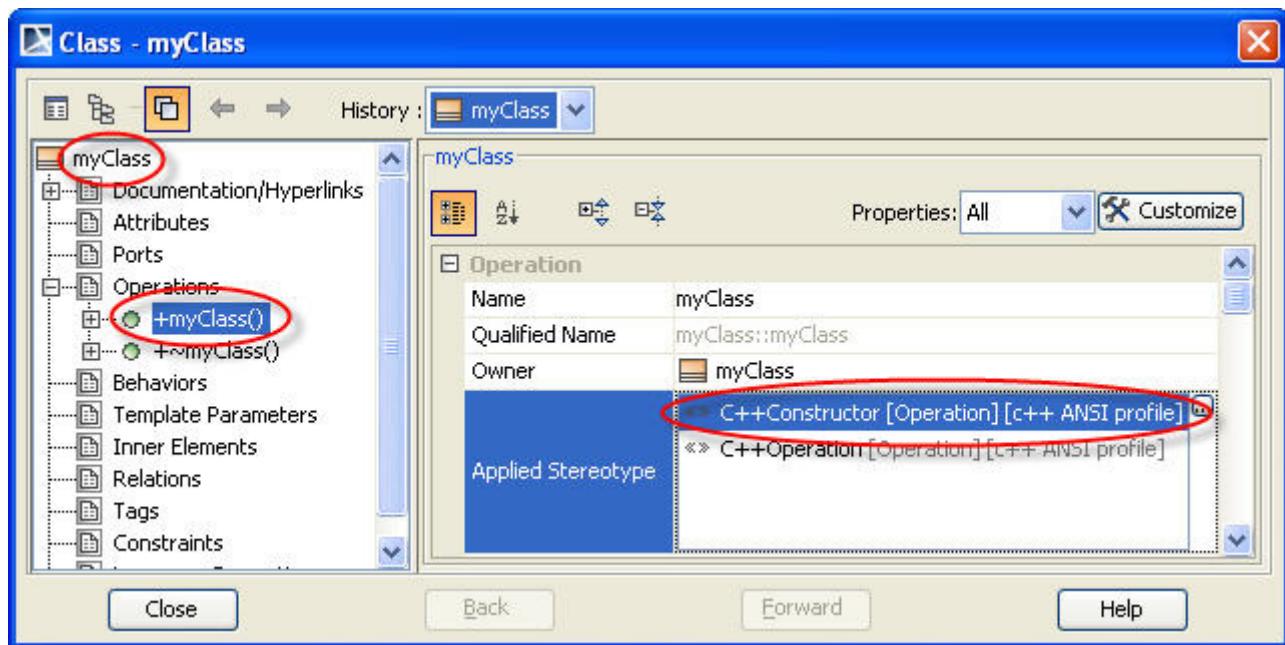
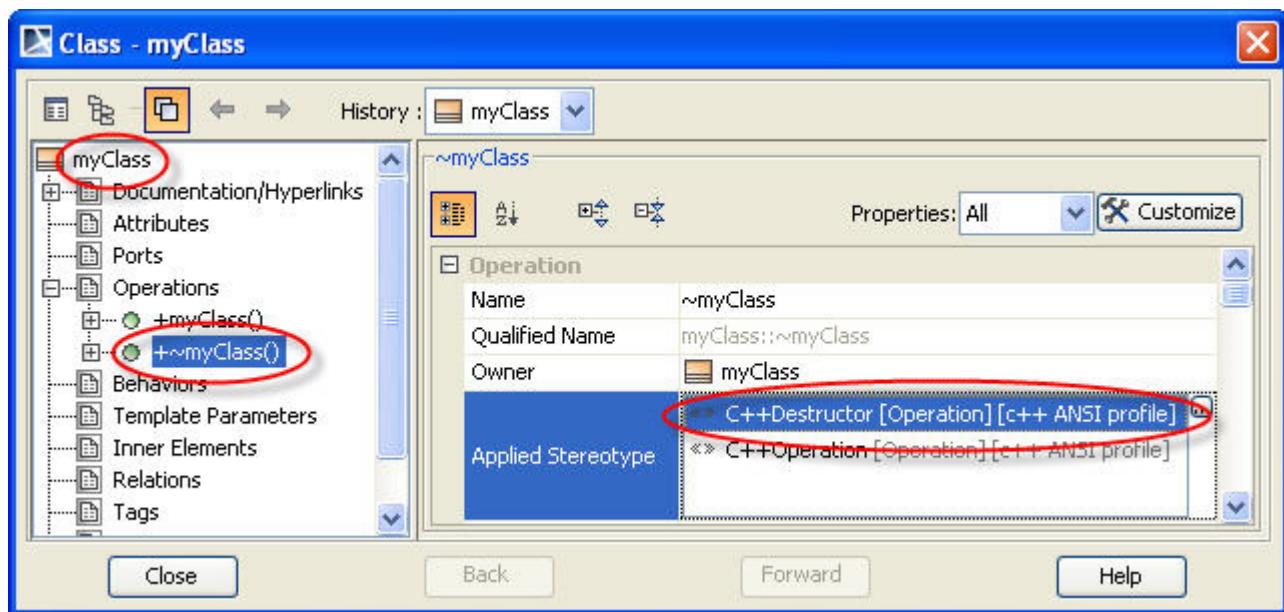


Figure 54 -- Constructor and Destructor name

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.

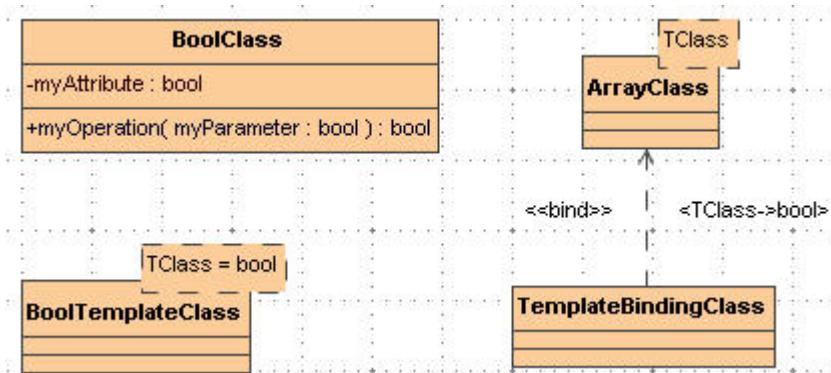


Figure 55 -- bool data type Example in Class Diagram

Attribute type

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

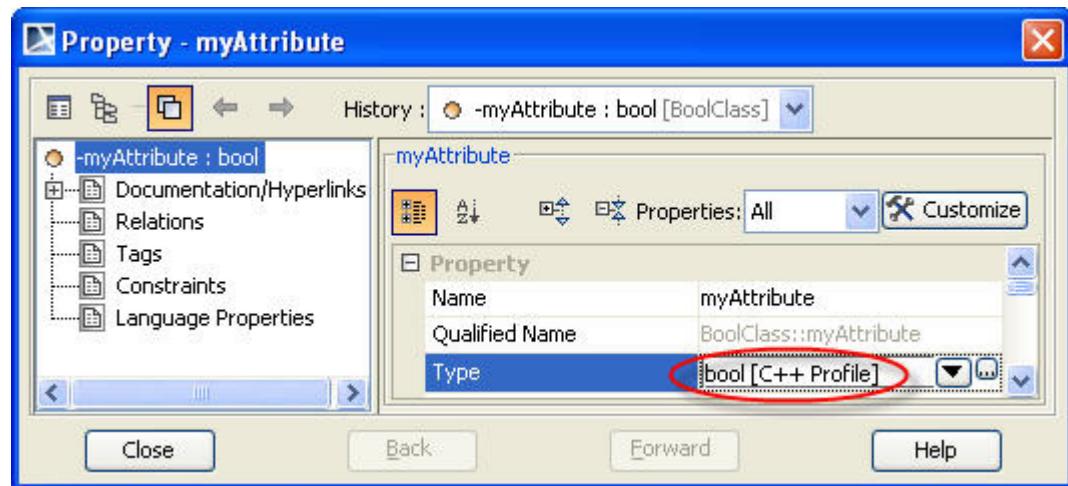
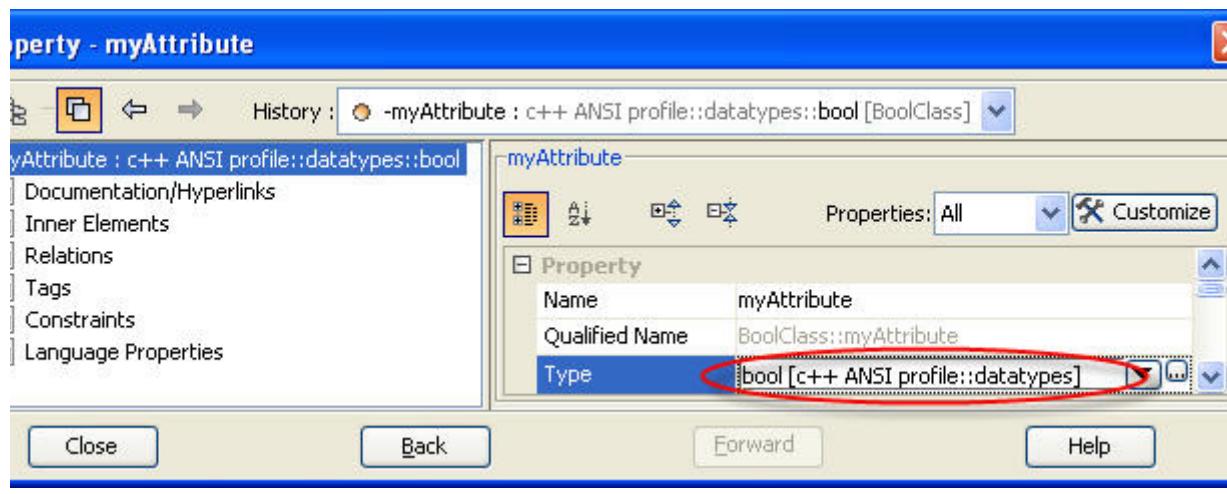


Figure 56 -- old bool data type in Attribute type

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.

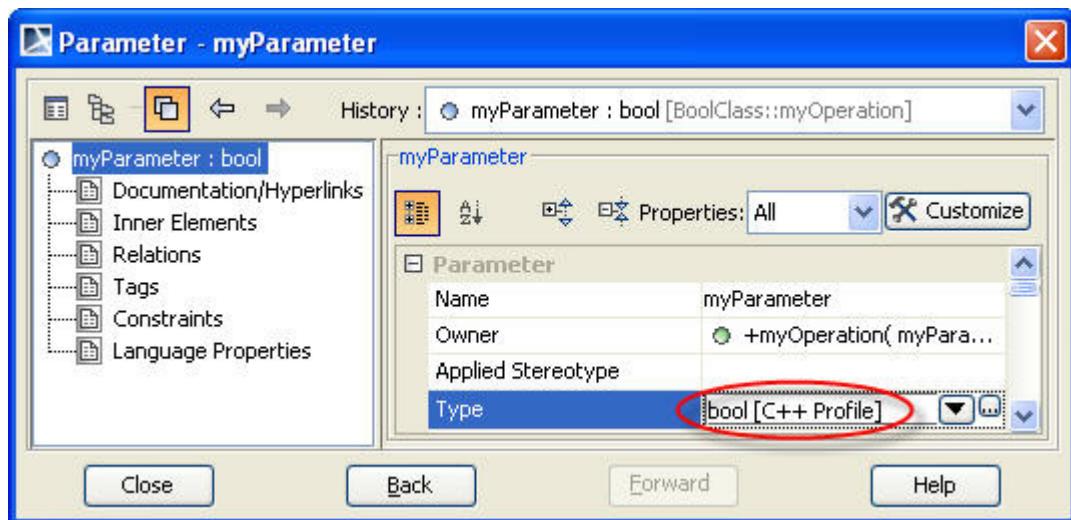
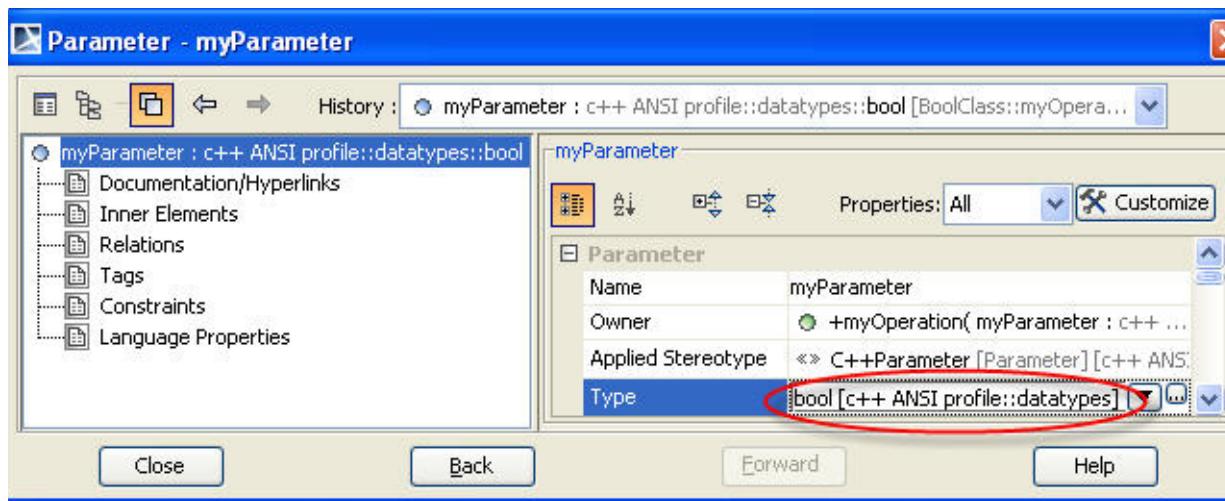


Figure 57 -- old bool data type in Parameter type

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.

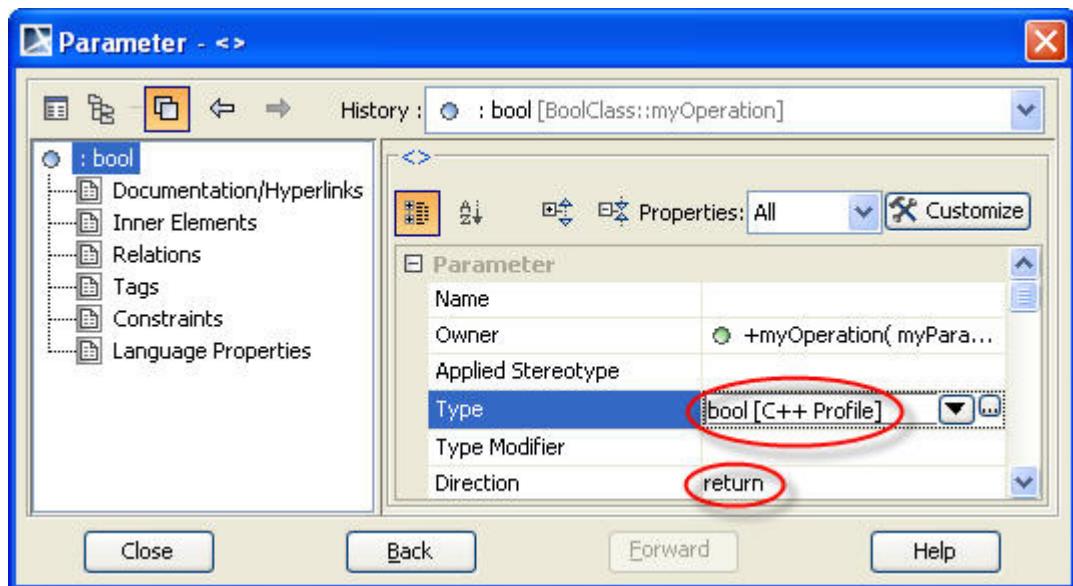
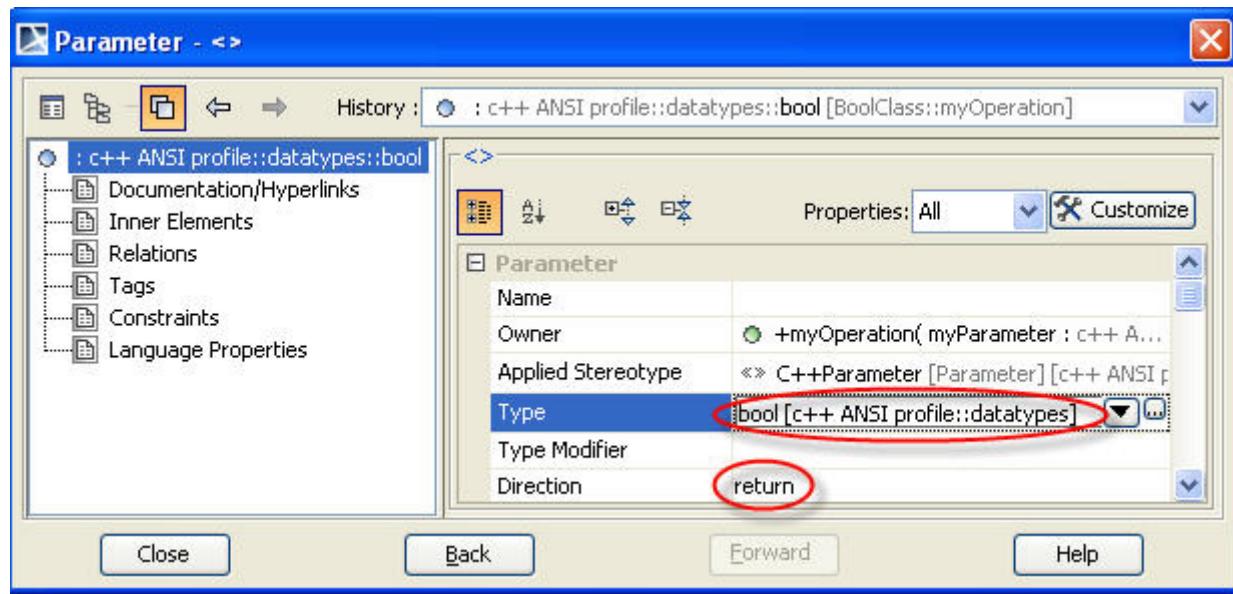


Figure 58 -- Figure 57 -- old bool type in Return type

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.

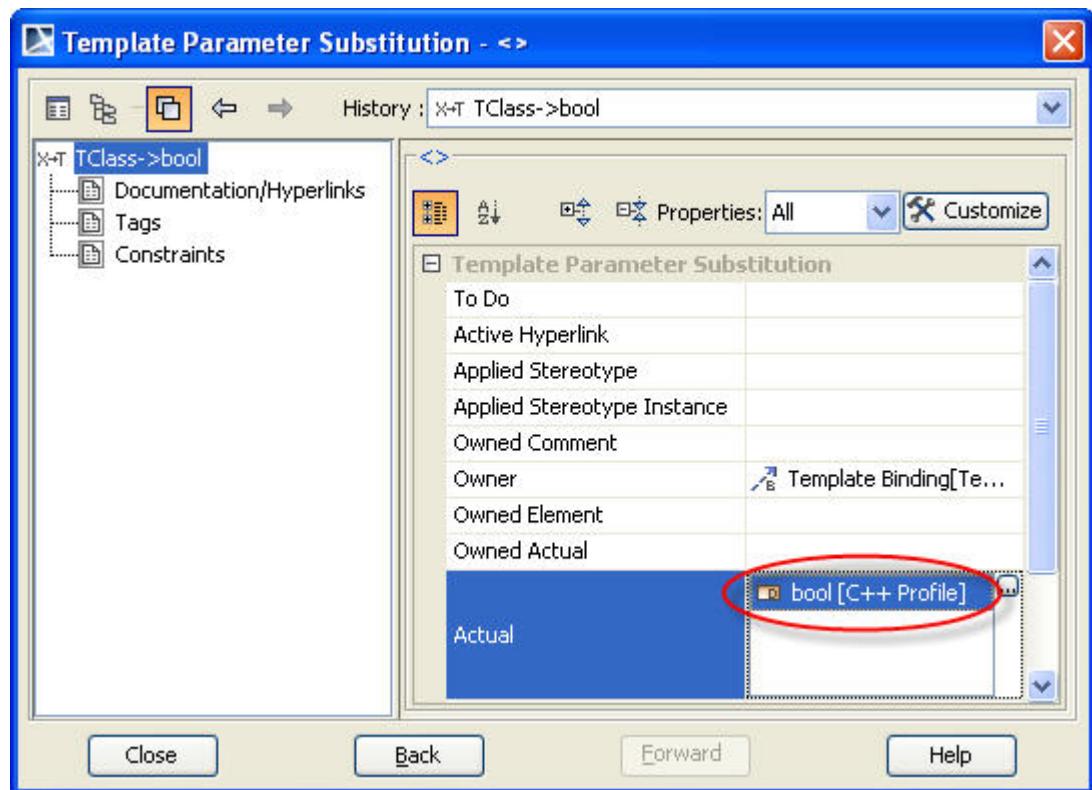
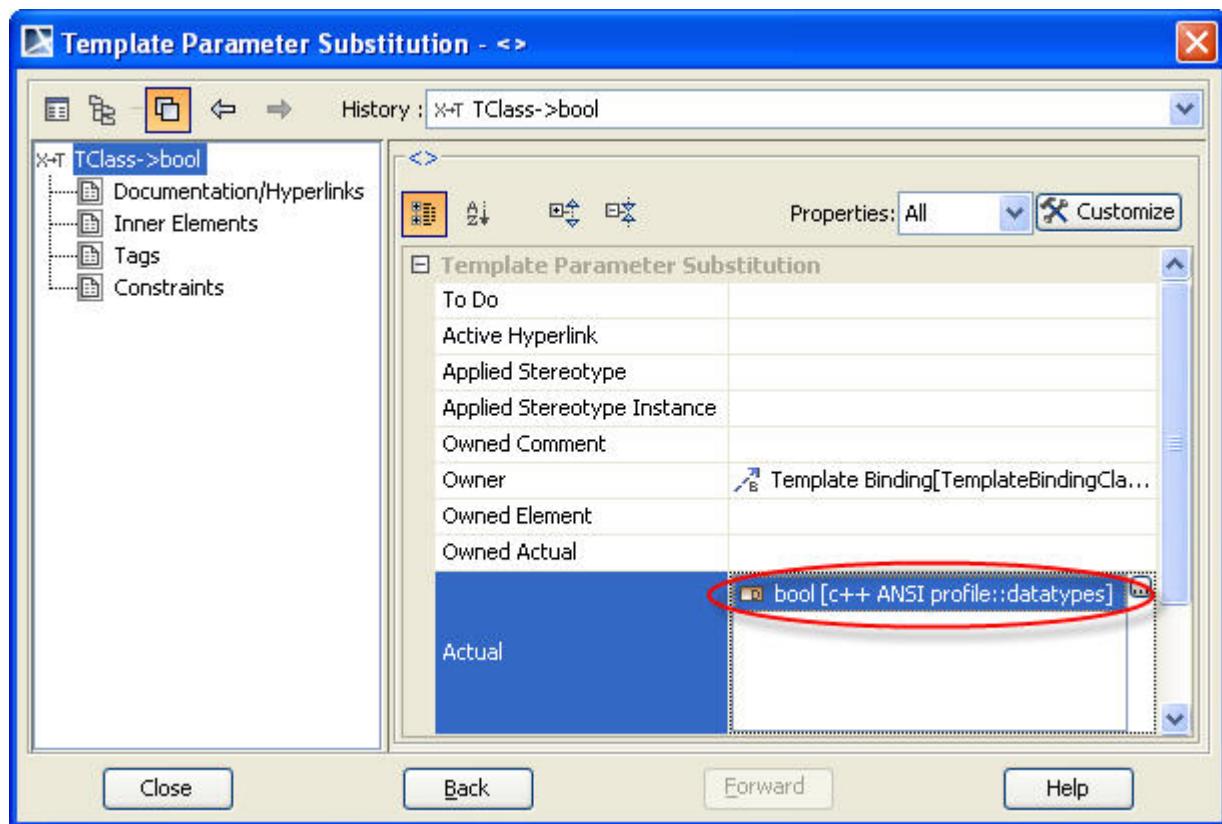


Figure 59 -- old bool data type in Actual in Template Parameter Substitution

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.

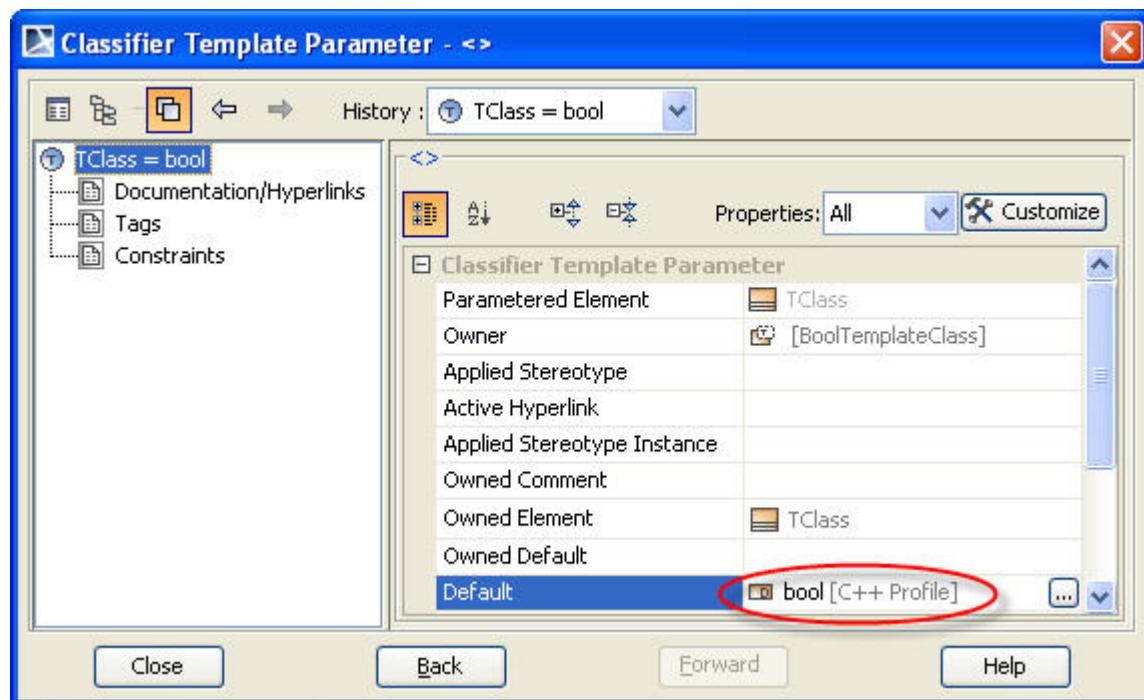
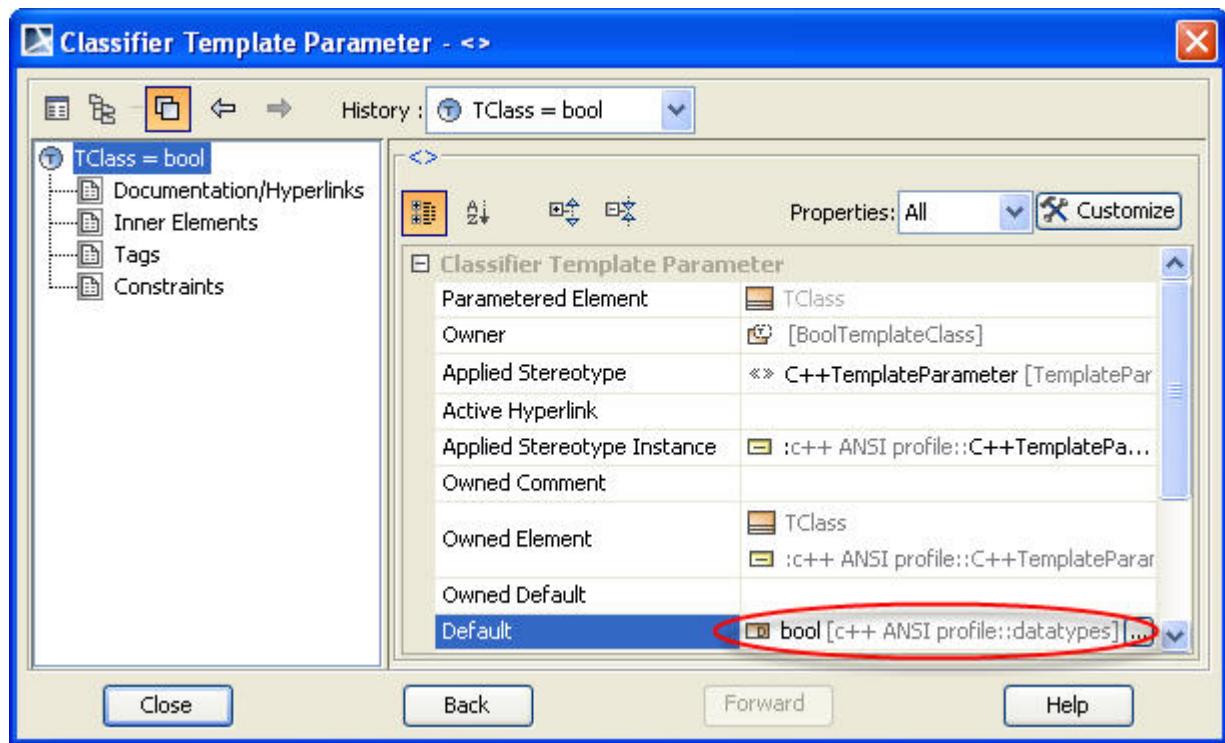


Figure 60 -- old bool data type in Default in Classifier Template Parameter

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.

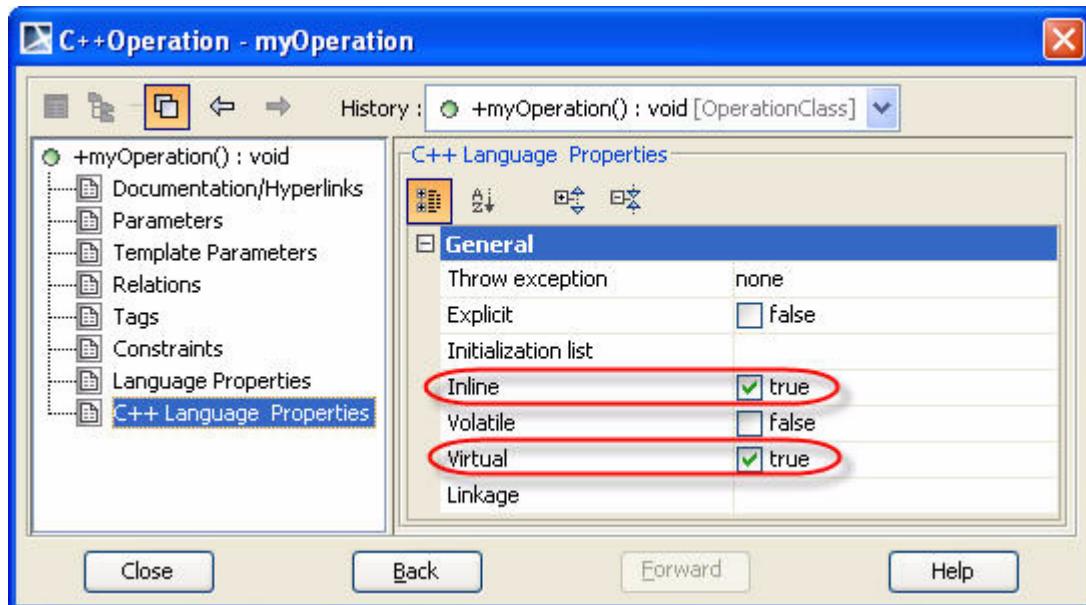


Figure 61 -- tag values in Operation

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

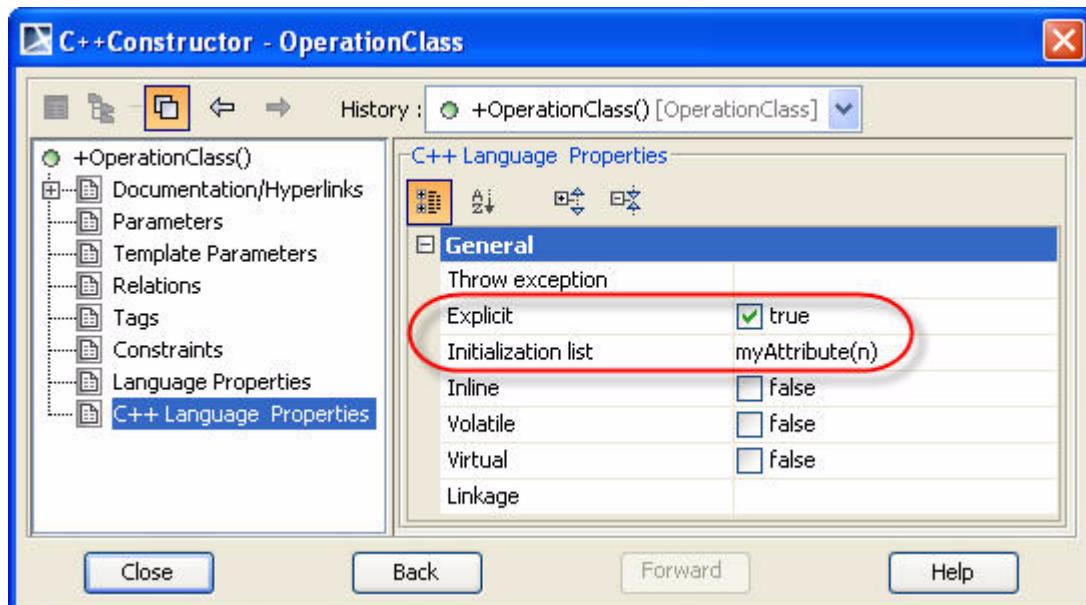


Figure 62 -- tag values in Constructor

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

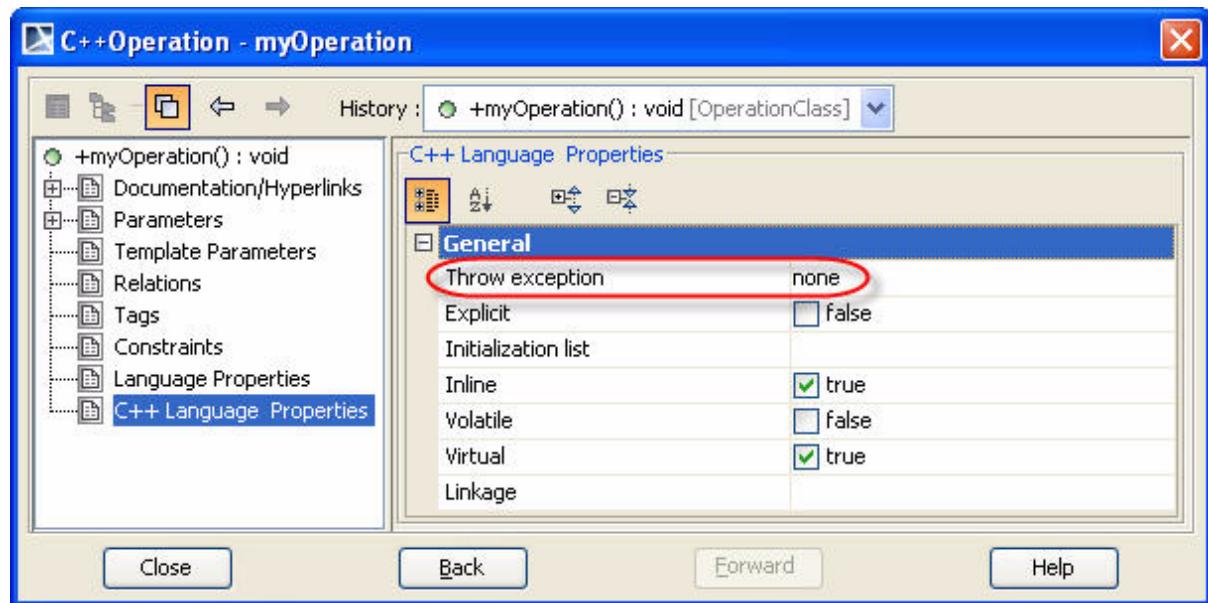


Figure 63 -- Throw exception tag value in Operation

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.

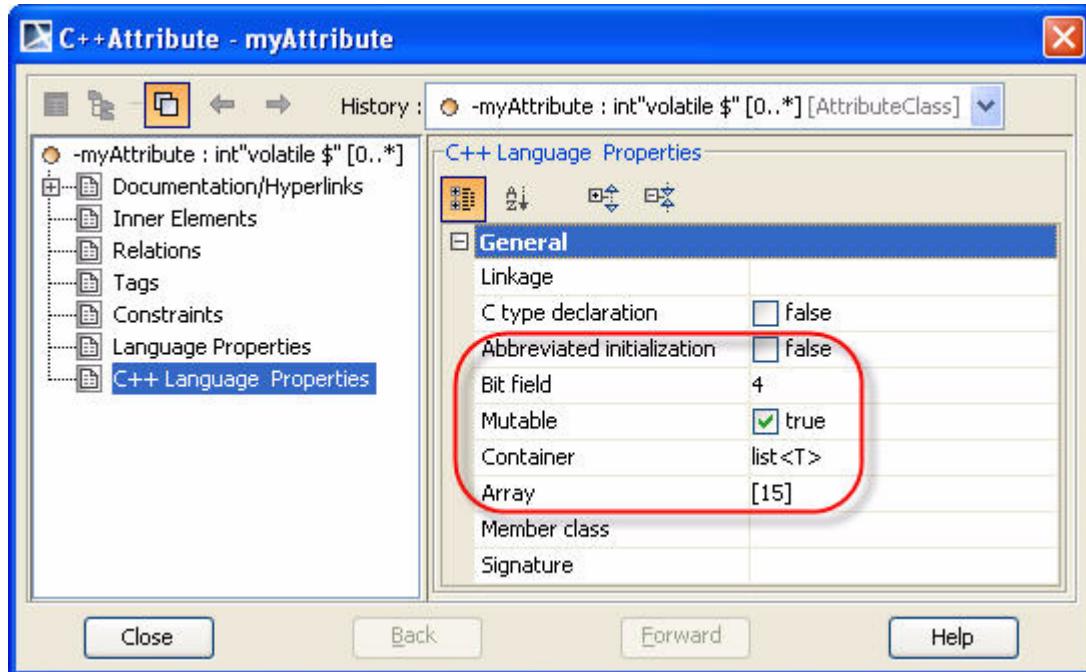


Figure 64 -- tag values in Attribute

Generalization

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

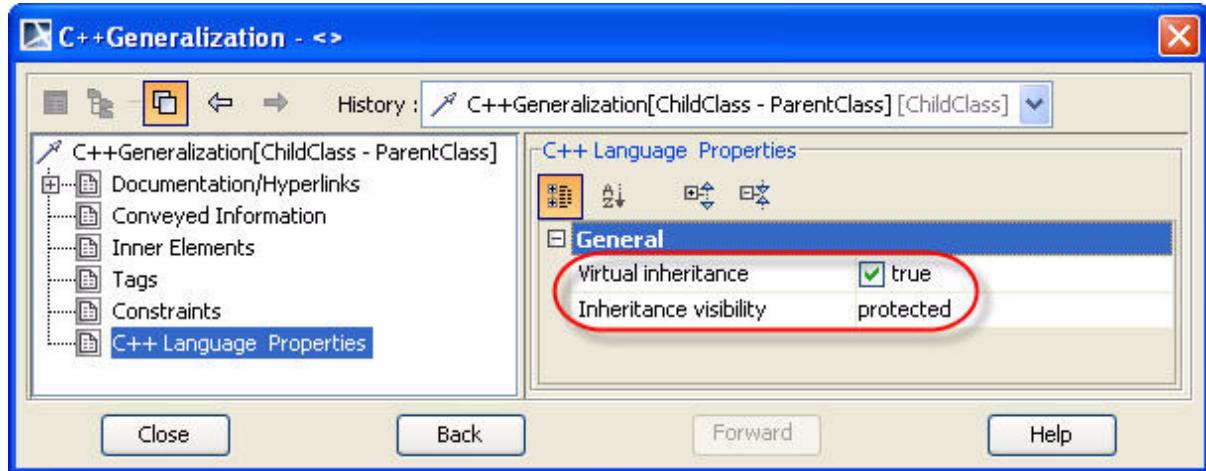


Figure 65 -- tag values in Generalization

Enumeration literal

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

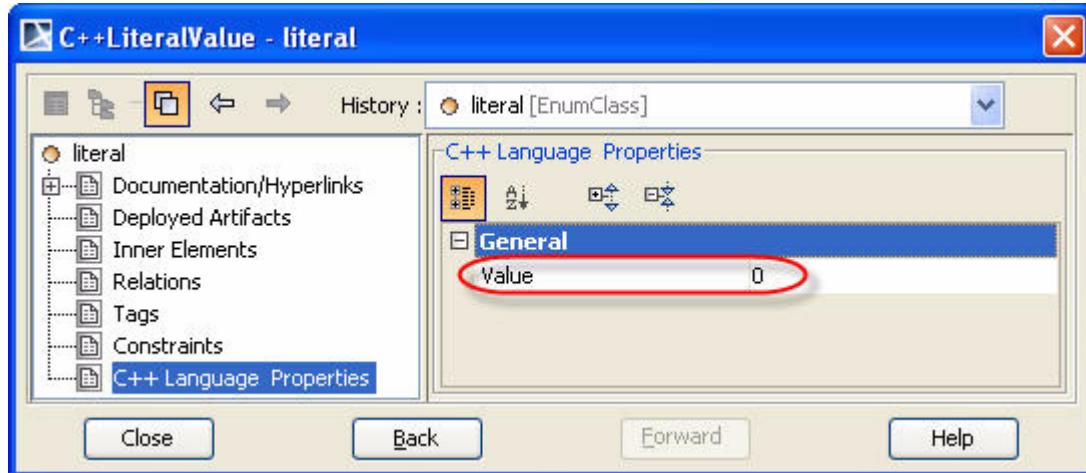


Figure 66 -- tag value in Enumeration literal

Namespace

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

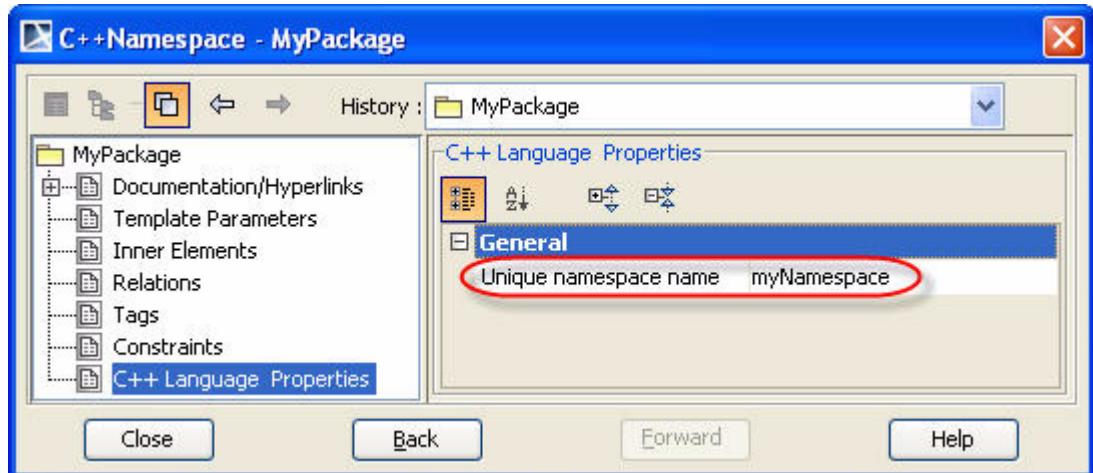


Figure 67 -- tag value in Namespace

Template parameter

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

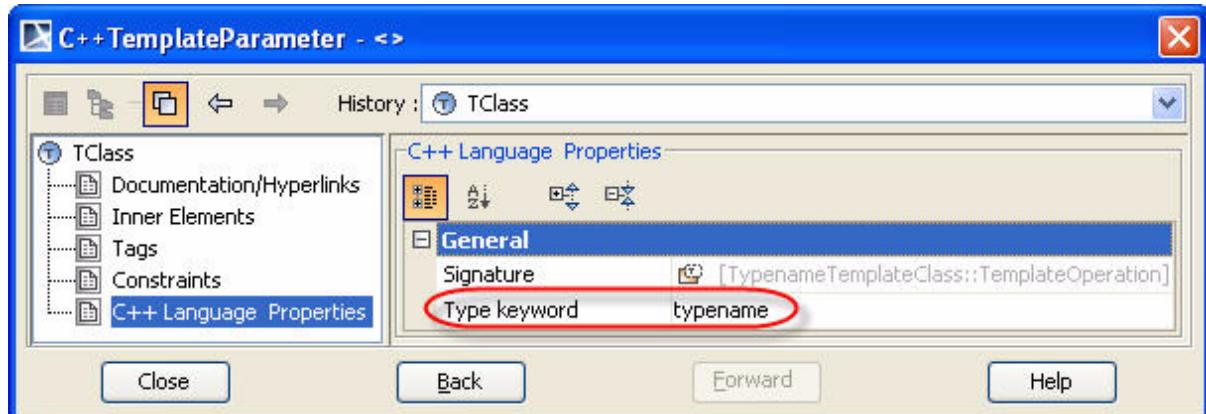


Figure 68 -- tag value in Template parameter

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

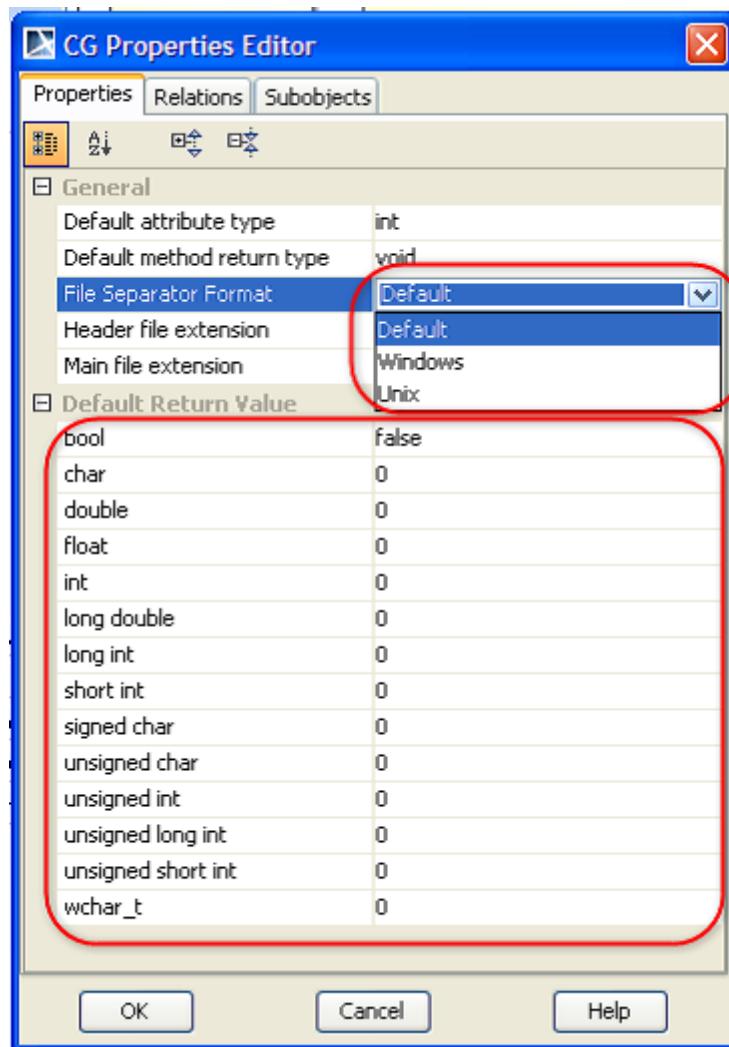


Figure 69 -- New properties

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

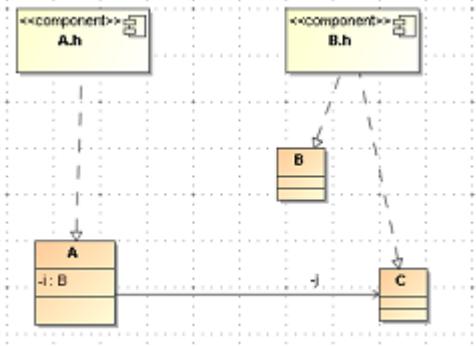
From the previous version of MagicDraw, the #include statement will be generated only at the first generation of a newly created file. According to this, it prevents user from doing roundtrip code engineering. Therefore, #include statement and forward class declaration are mapped to the model in MD 12.1.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

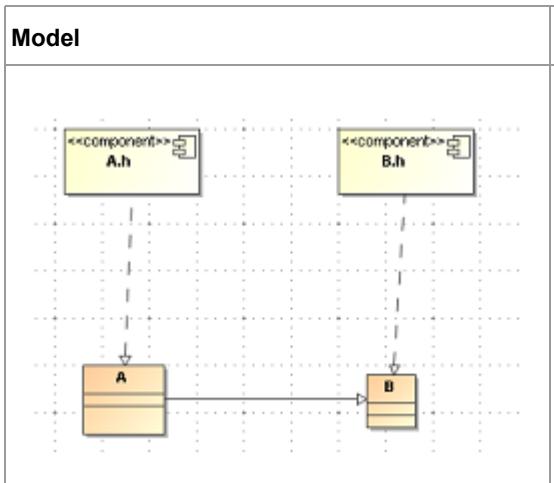
Note: The usage relationship corresponding to the #include statement is also created in the model

Property

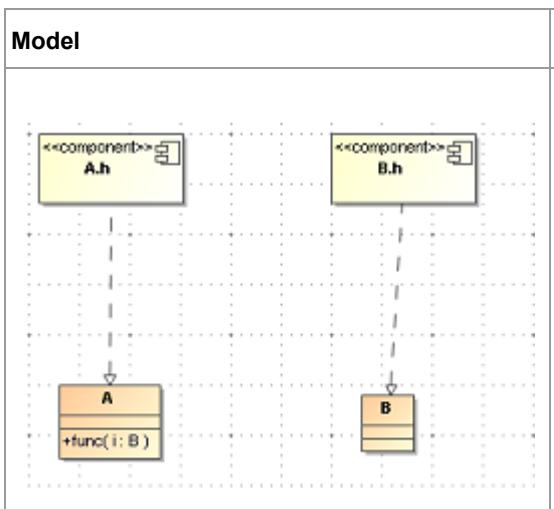
Model	Code
	#include "B.h" class A{ B i; C* j; };

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

Generalization

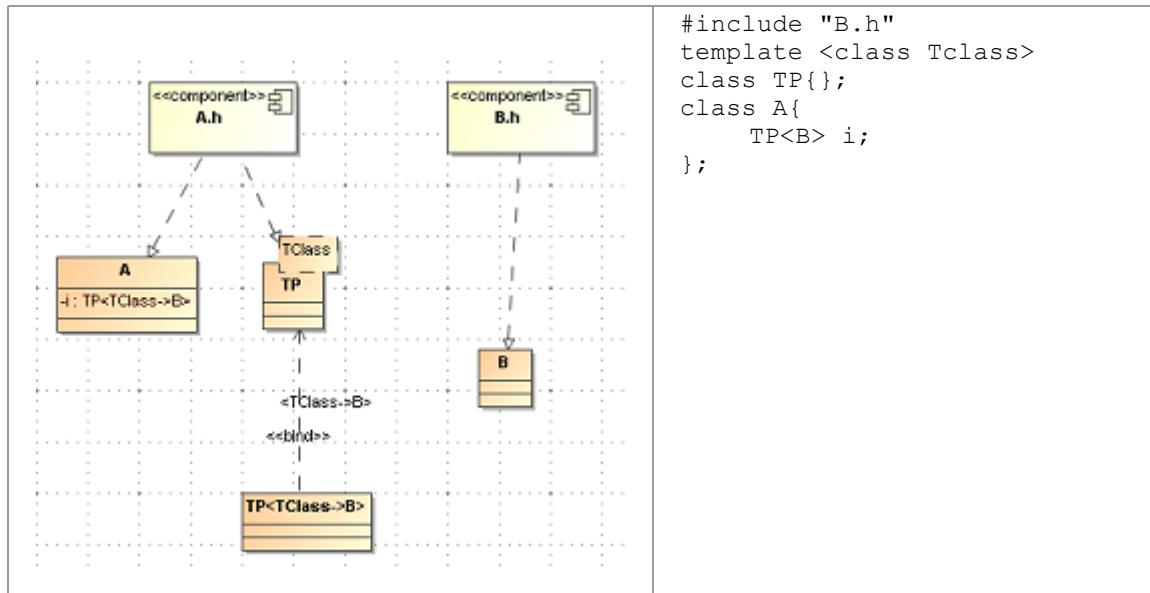
Model	Code
	#include "B.h" class A:B{ };

Parameter

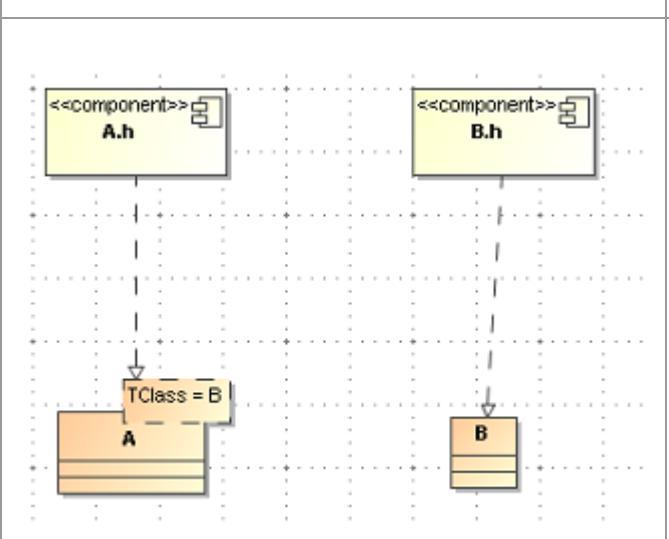
Model	Code
	#include "B.h" class A{ public: void func(B i); };

Template Parameter Substitution

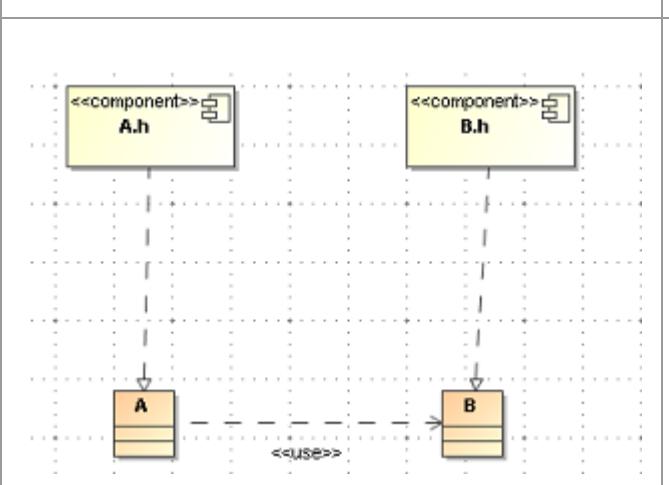
Model	Code
-------	------



Template Parameter Default Value

Model	Code
	<pre>#include "B.h" template <class TClass=B> class A{};</pre>

Usage

Model	Code
	<pre>#include "B.h" class A{ };</pre>

Project Option and Code Generation Options

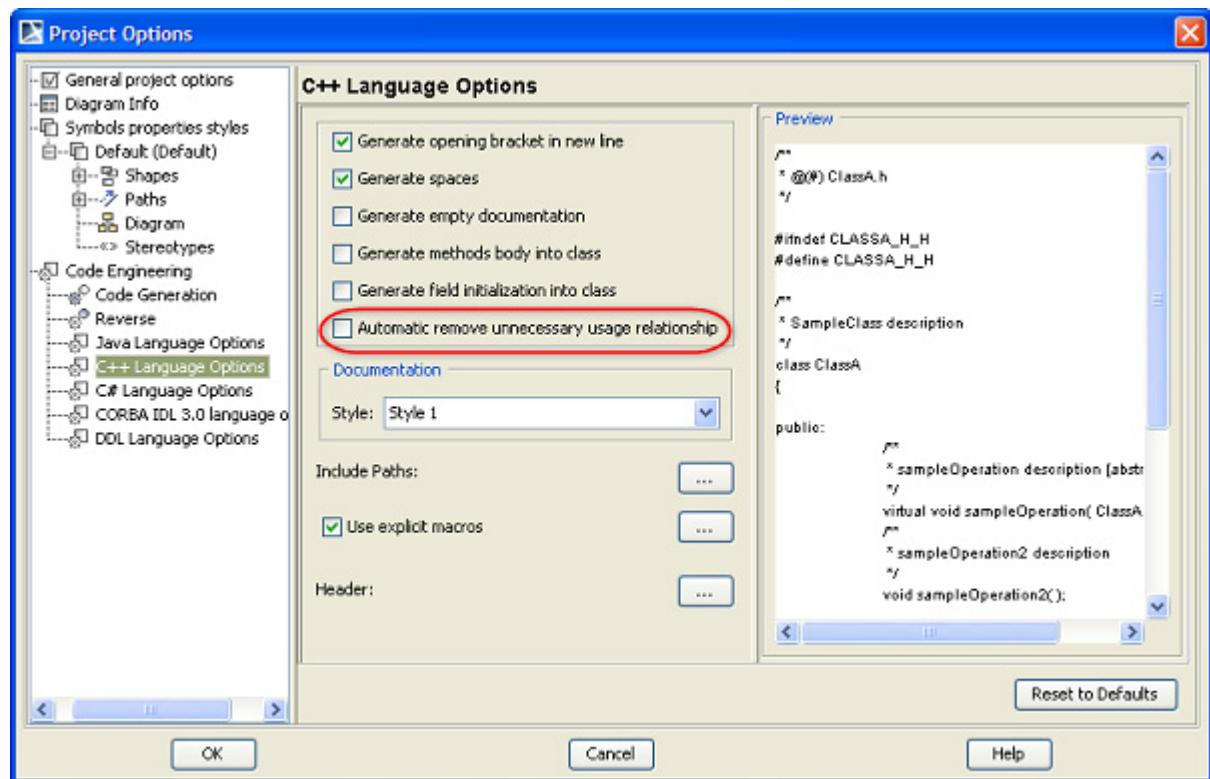


Figure 70 -- New Option in Project Options

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

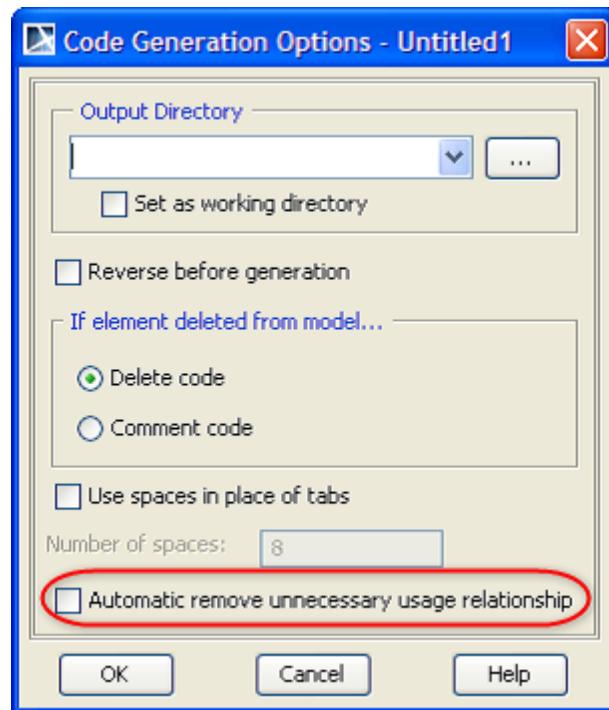
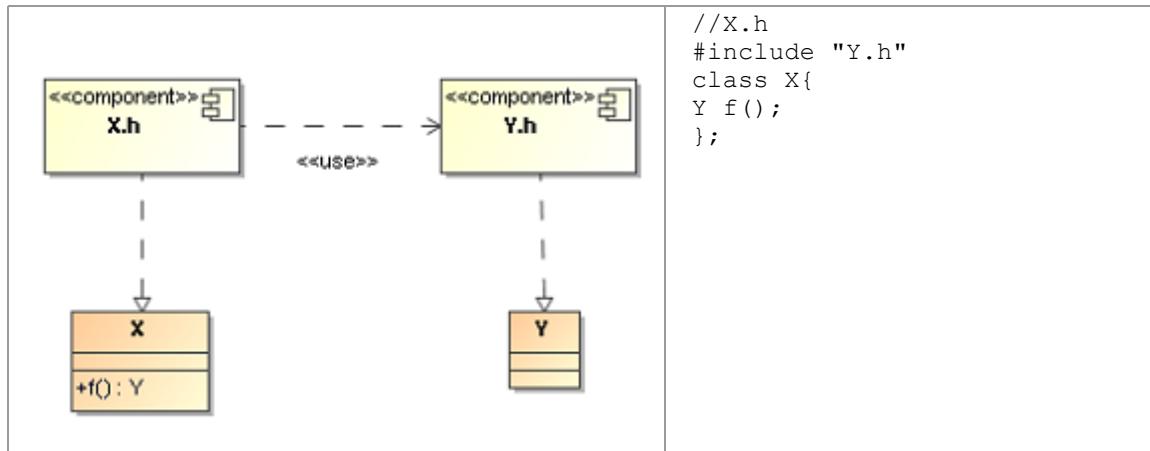


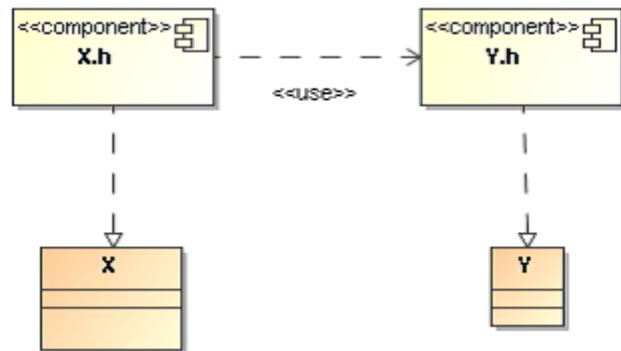
Figure 71 -- New option in Code Generation Options

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
-------	------



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

Selected	<p>The diagram shows two components, X.h and Y.h, each connected to its corresponding implementation class (x and y) via dashed lines. A dashed arrow labeled '<<USE>>' points from X.h to Y.h, indicating a USE dependency. Both the component boxes and the implementation boxes are highlighted in yellow, signifying they are selected.</p>	//X.h #include "Y.h" class X{};
Not Selected	<p>The diagram shows two components, X.h and Y.h, each connected to its corresponding implementation class (x and y) via dashed lines. A dashed arrow labeled '<<USE>>' points from X.h to Y.h, indicating a USE dependency. Only the component box for X.h is highlighted in yellow, signifying it is selected. The implementation box x and the component box Y.h are not highlighted.</p>	//X.h class X{};

New in MagicDraw 14.0

Support C++ dialects

C++ code engineerint 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 Secification base 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.

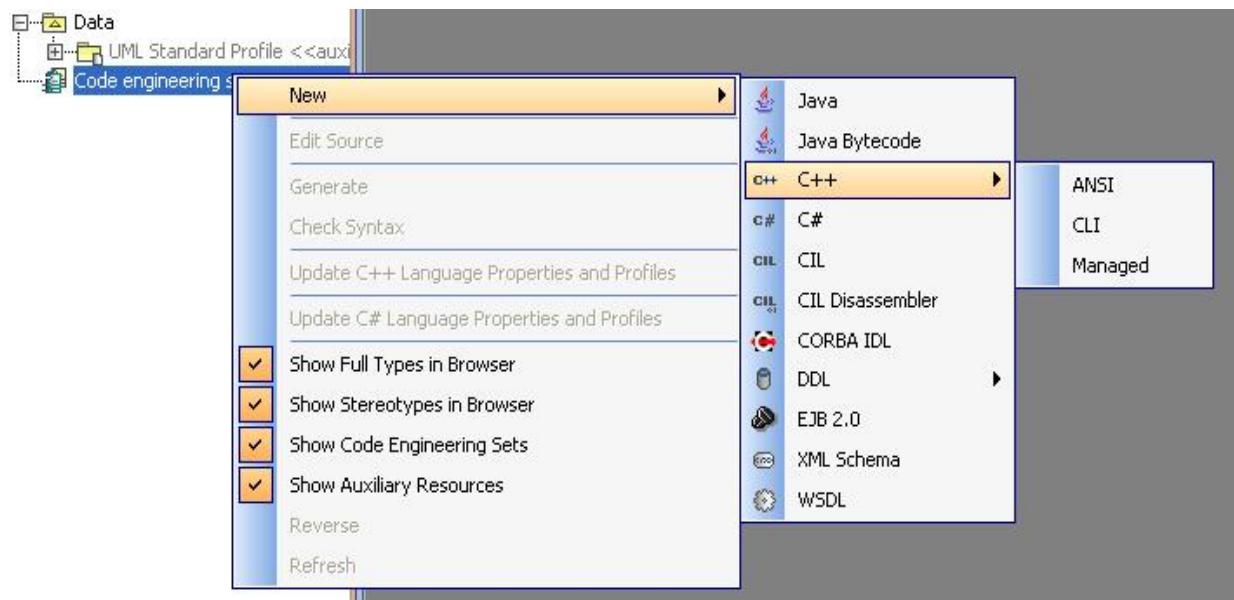


Figure 72 -- New dialects in C++ CE

CG Properties Editor

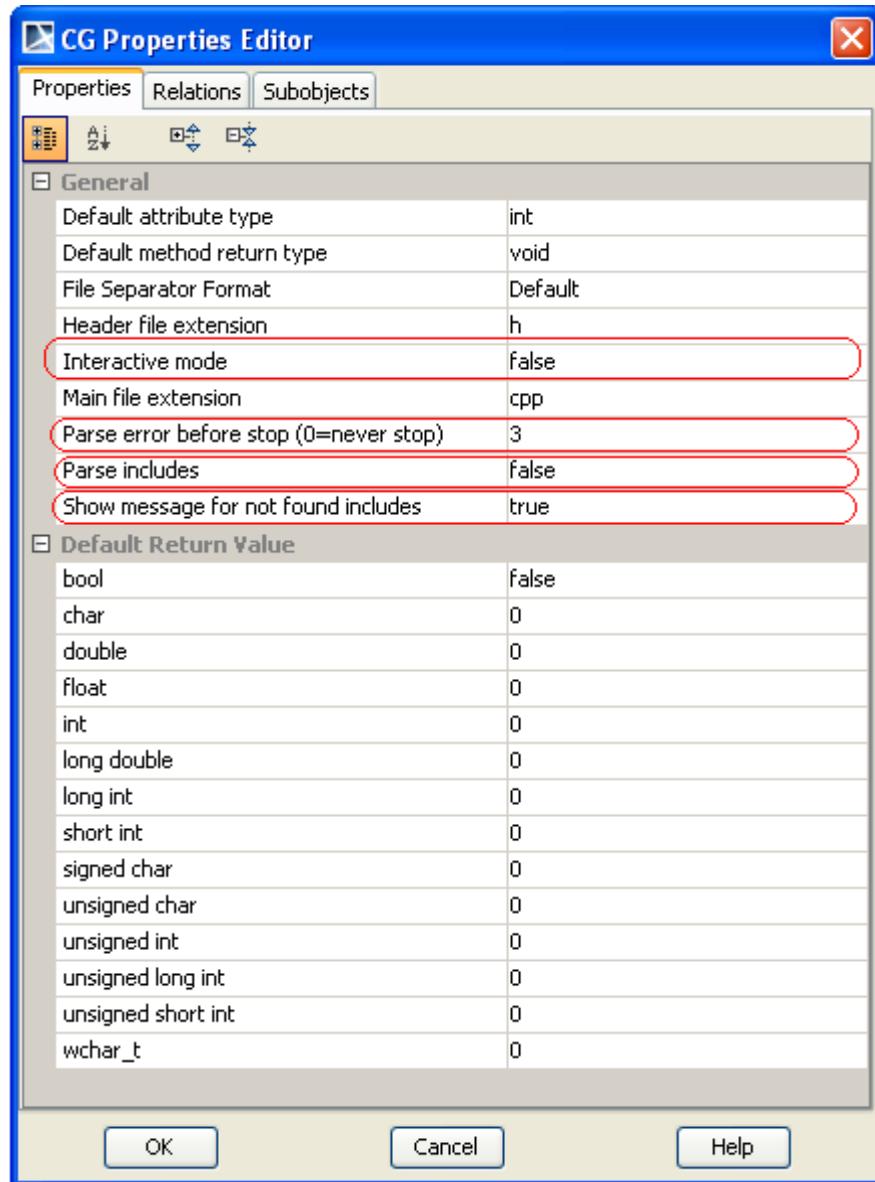


Figure 73 -- New properties

Property name	Description
Interactive mode	If enabled, user will be questioned during the reverse process when the parser cannot decide whether the symbol is a class or namespace. For example If you reverse the following code
	<pre>class A { B::X i; };</pre>
	The parser does not have enough information to decide whether B is class or namespace.

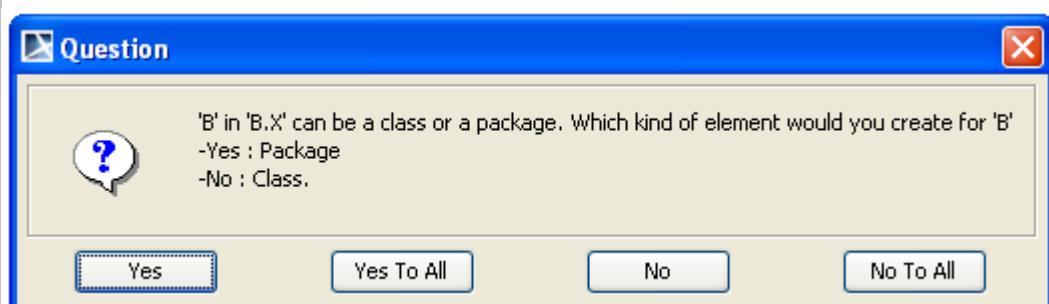


Figure 74 -- Question dialog for interactive mode

NOTE If the default mode, not interactive mode, the reverse module will automatically do the best guess according to the information in code.

Parse error before stop	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.
Parse Includes	If set to false, the reverse module will reverse only the selected files and ignore all #include statements.
Show message for not found includes	Used to display messages in the message window when the reverse module cannot find the files included in the #include statement. NOTE You have to set the property Parse Includes to true in order to use the property Show message for not found includes .

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.

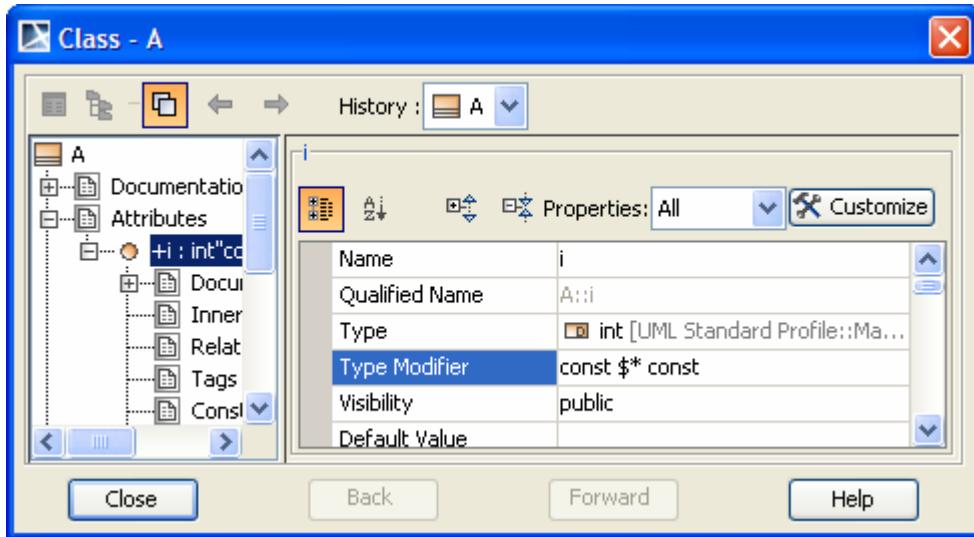


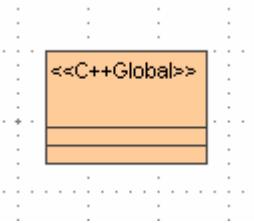
Figure 75 -- Type Modifier Example

Global Member

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

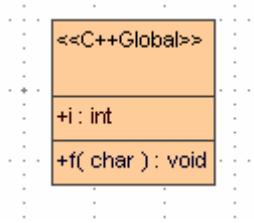
1. Create a class

[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 115 for example.

TypeDef

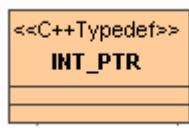
With new C++ ANSI profile, you can model both normal TypeDef and TypeDef of function pointer.

Normal TypeDef

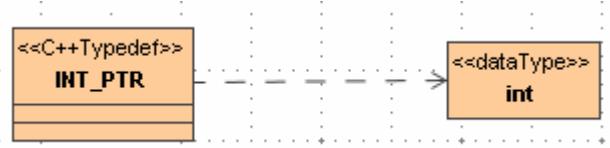
The steps below show you how to create the model that can be used to generate the following code.

```
typedef int* INT_PTR;
```

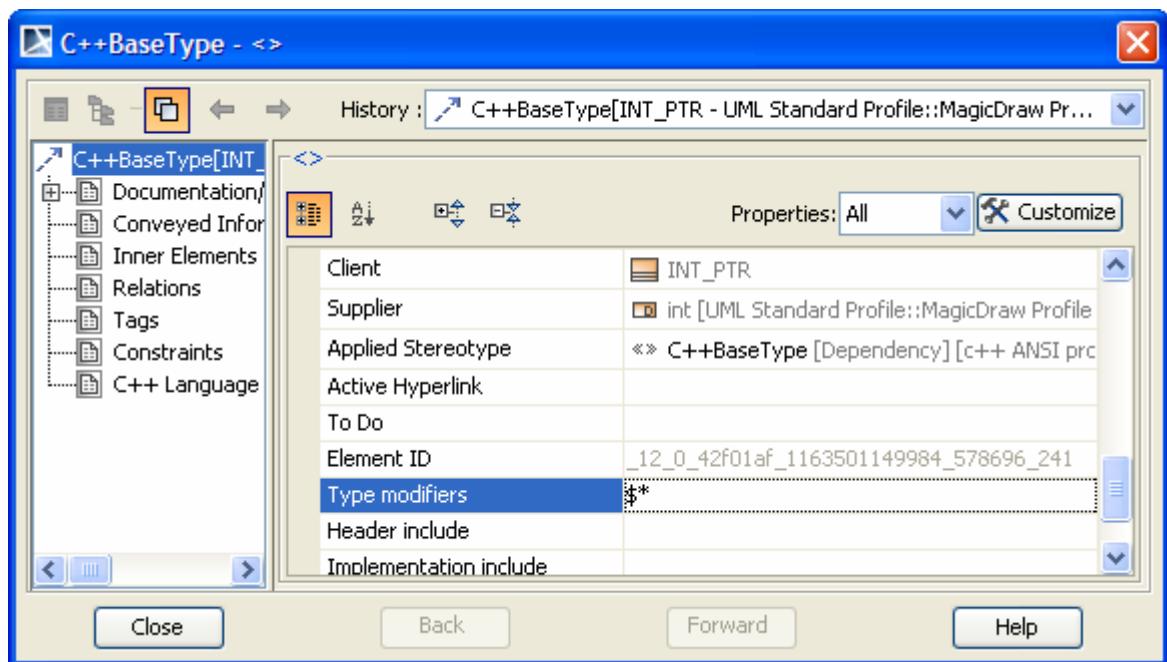
1. Create a class, apply stereotype <<C++Typedef>> and name it with typedef name. In this case, INT_PTR.

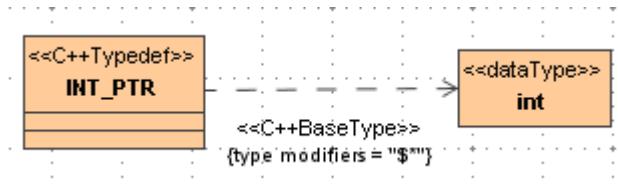


2. Draw Dependency relationship from such class to base type element. In this case, int.



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



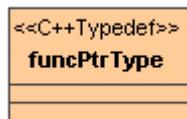


Typedef of function pointer

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

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

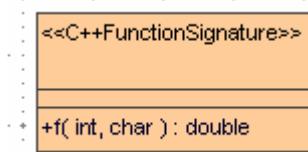
1. Create a class, apply stereotype <<C++Typedef>> and name it with typedef name. In this case, funcPtrType.



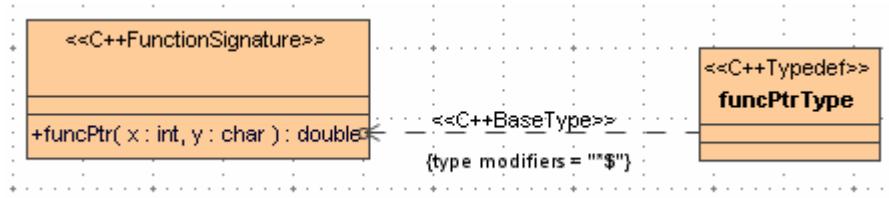
2. Create an operation with function signature type of function pointer in <<C++FunctionSignature>> class. The operation name does not have any effect on code generation.

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

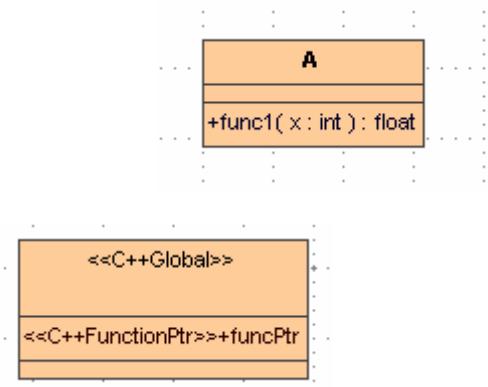
The steps below show you how to create the model that can be used to generate the following code.

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

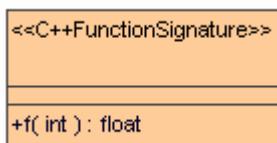
float (A*funcPtr) (int);
```

Suppose that the model for class A is already created.

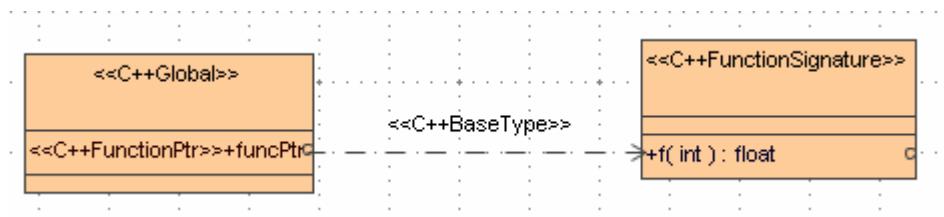
1. Create attribute with name, in this case, **funcPtr** and apply `<<C++FunctionPtr>>` stereotype. In this example, **funcPtr** is created in `<<C++Global>>` class means that **funcPtr** is global member.



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.



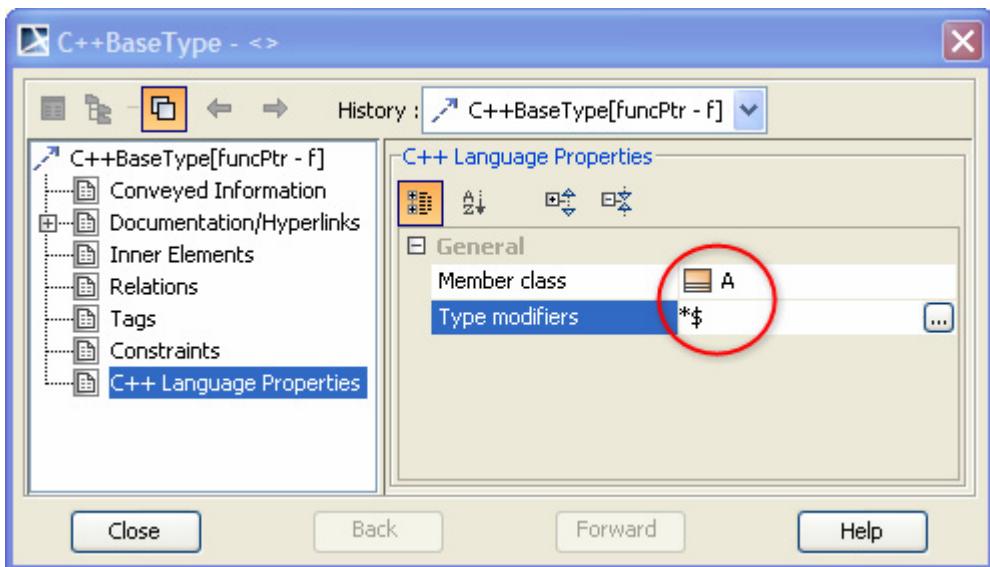
3. Draw a Dependency from funcPtr to f that is just created in <<C++FunctionSignature>> class and set <<C++BaseType>> stereotype to the Dependency.

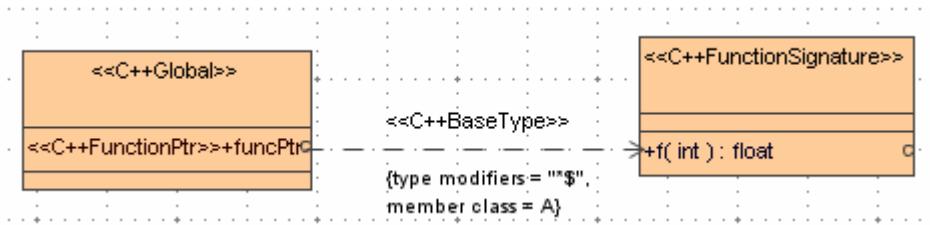


4. Open specification dialog of <<C++BaseType>> Dependency.

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

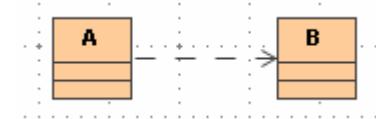
With new C++ ANSI profile, you can model C++ friend, both friend class and friend function.

Friend Class

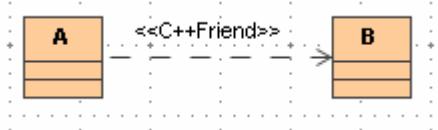
The steps below show you how to create the model that can be used to generate the following code.

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

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



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



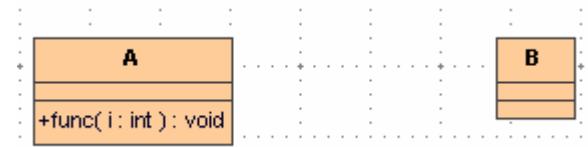
Friend Function

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

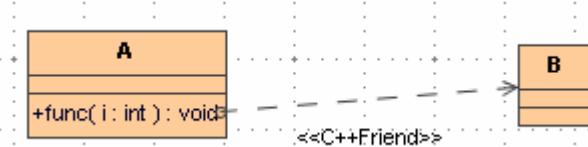
```
class A
{
    void func(int i);
};

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**.



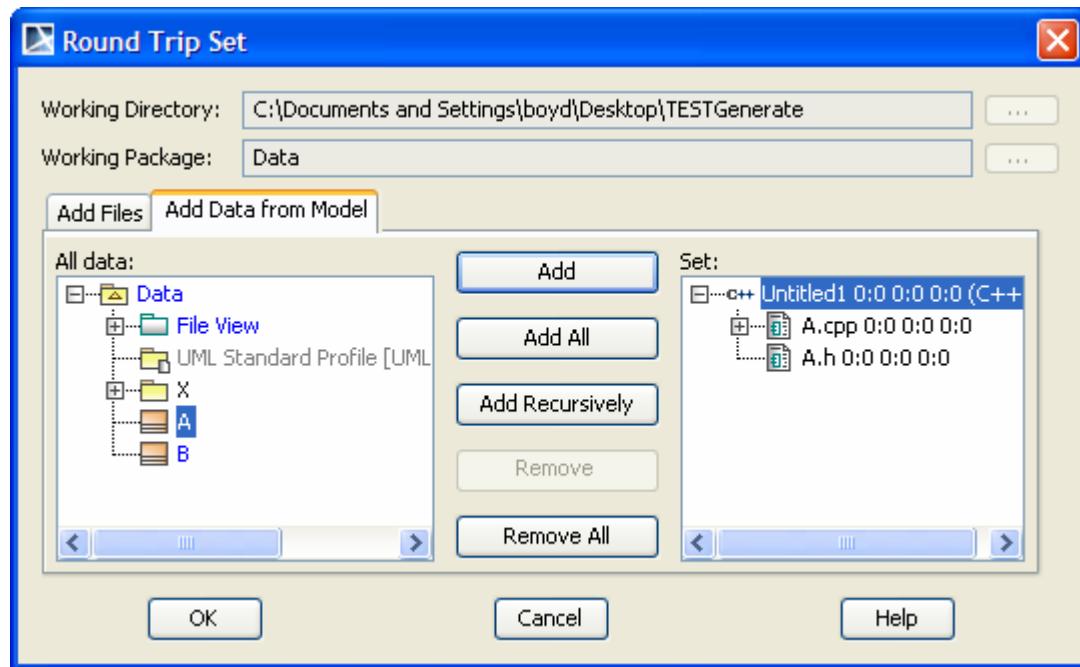
2. Apply stereotype <<C++Friend>> to the dependency.



See Friend declaration for mapping example.

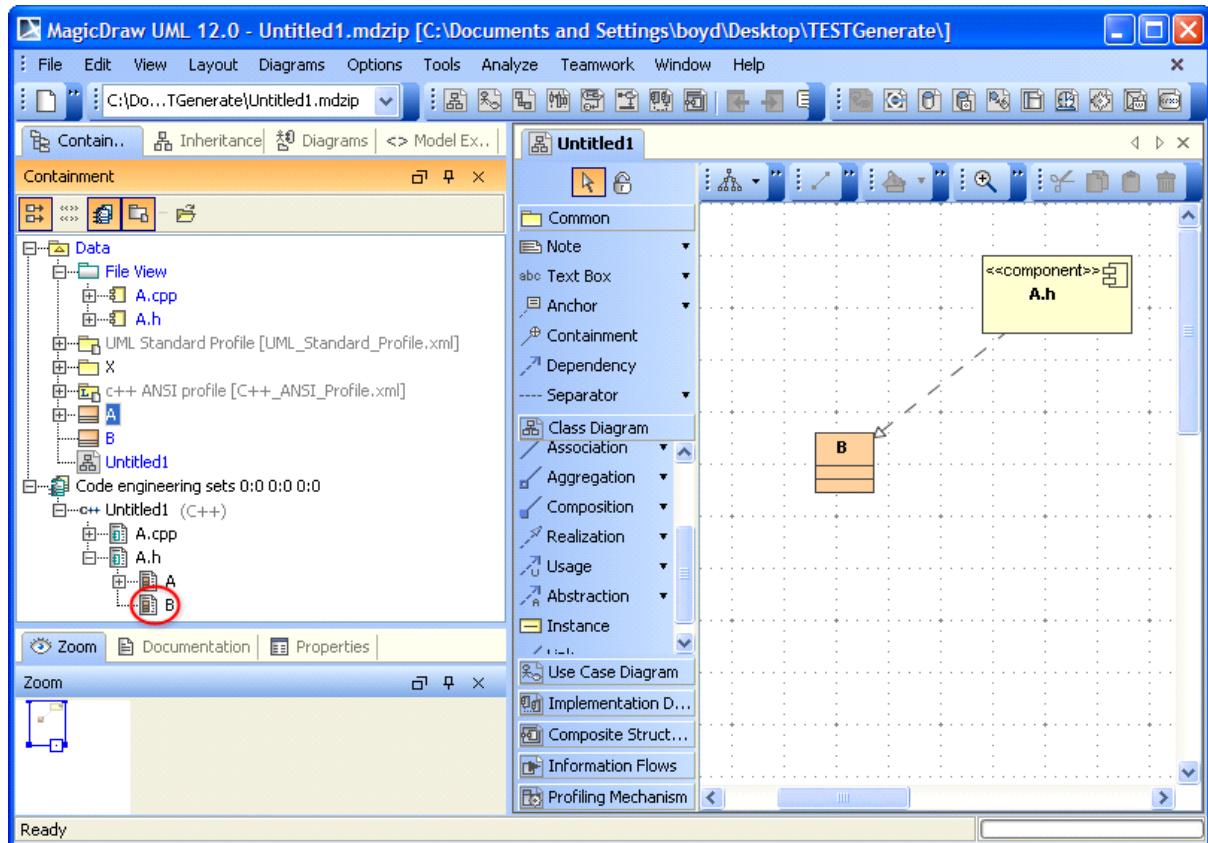
How to specify component to generate code to

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.



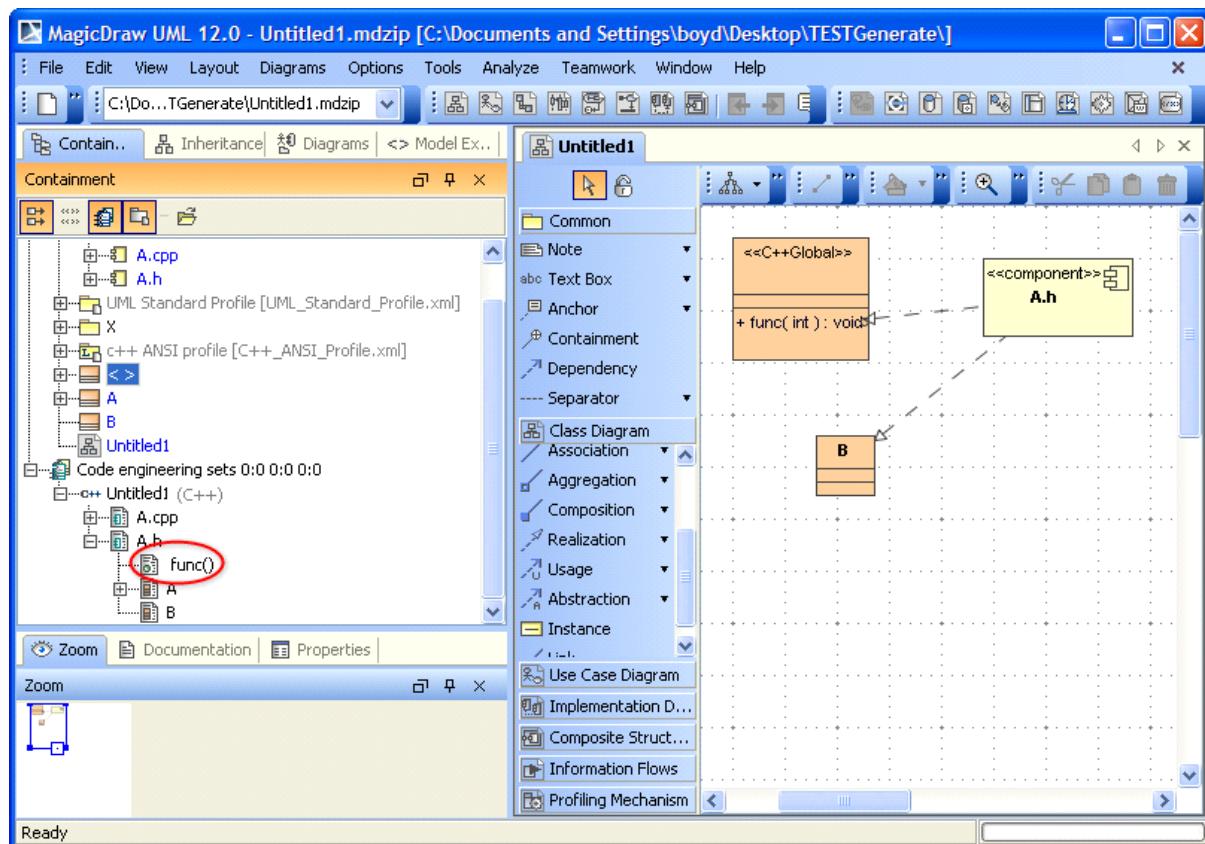
However, MagicDraw allows user to specify the component to generate the model into by using Realization.

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.



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>           include system file
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 }
```

Figure 76 -- *Include system file example*

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

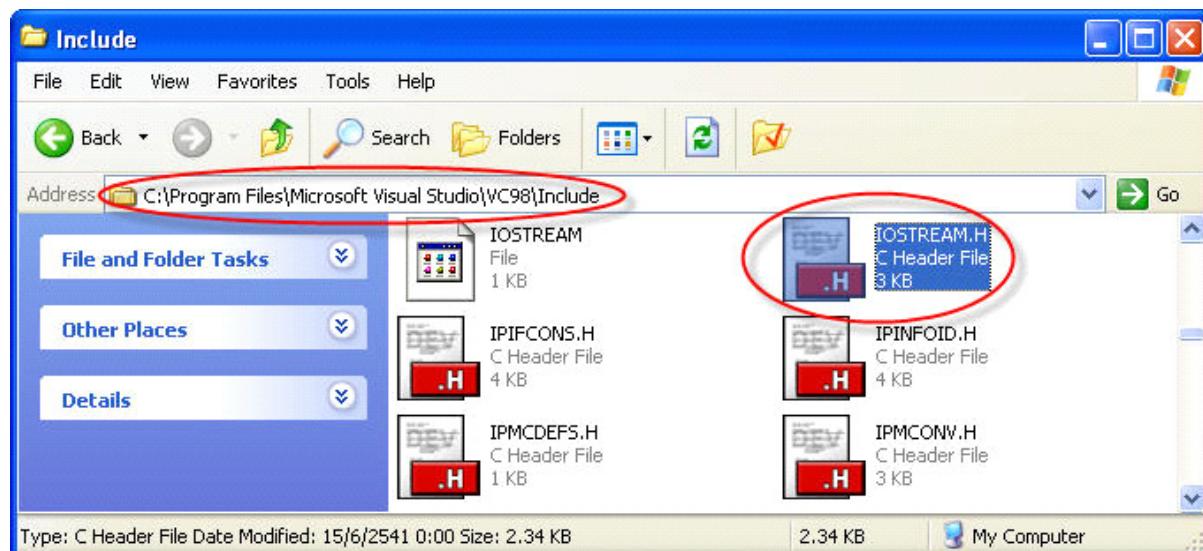


Figure 77 -- System file paths

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.

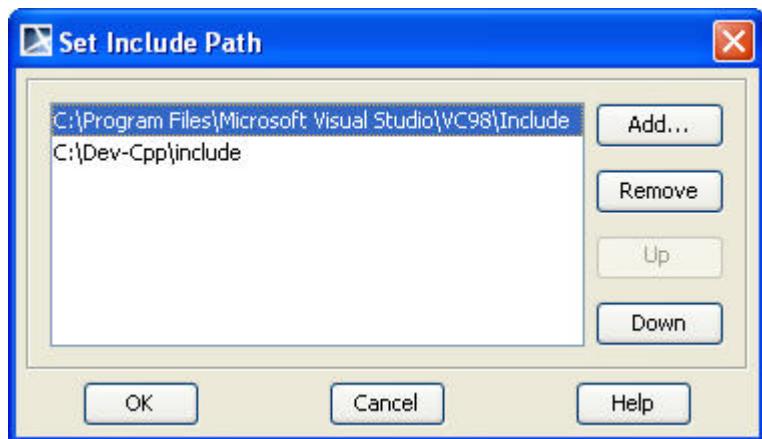
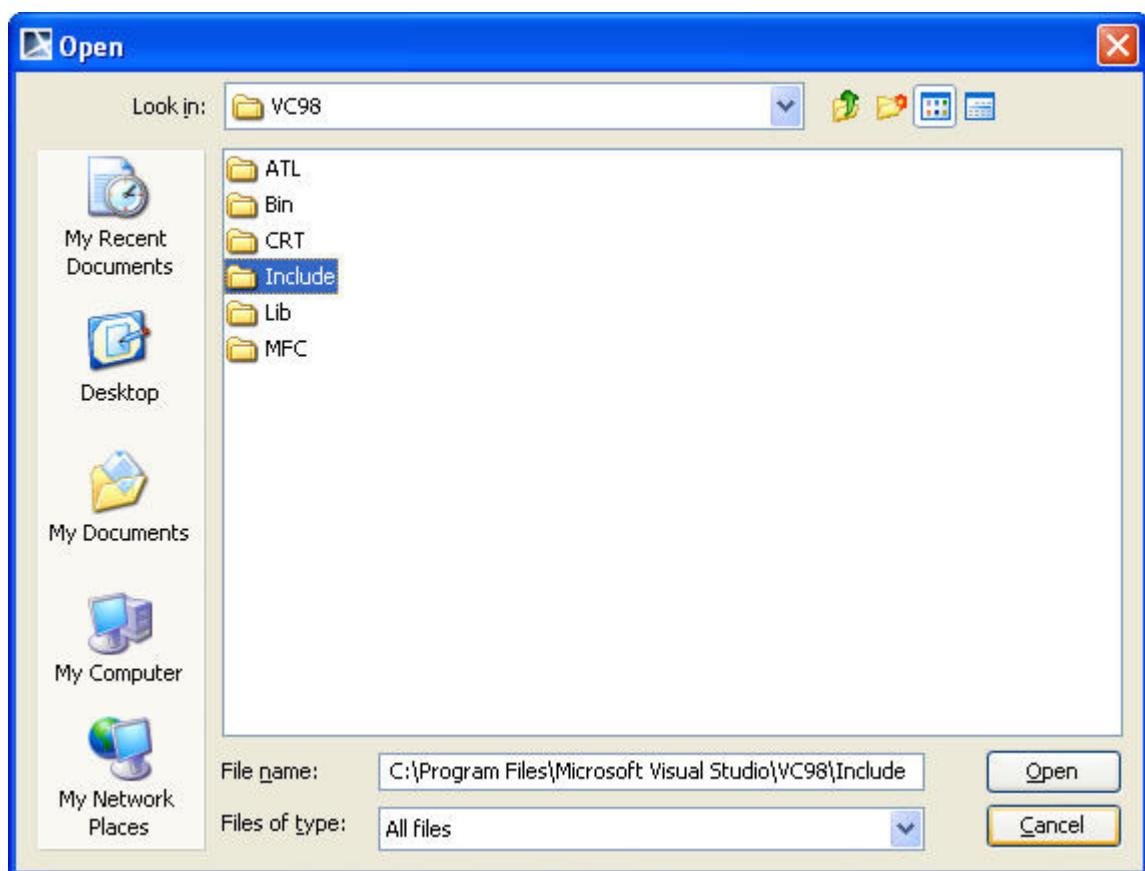


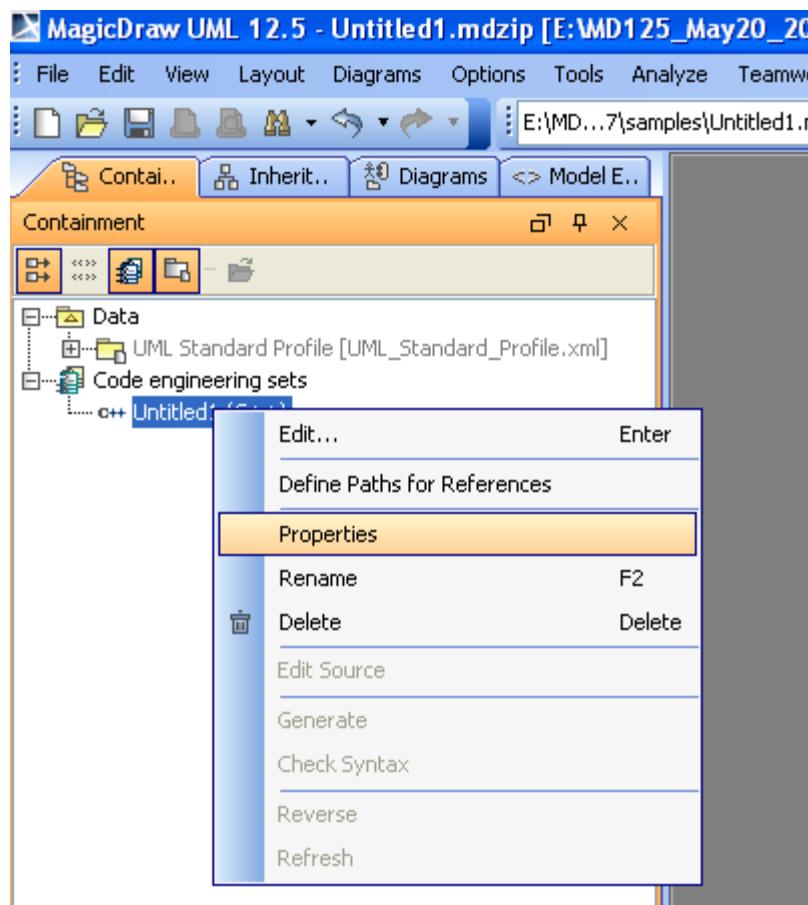
Figure 78 -- Set Include Path dialog box

Button	Action
Add	The Open dialog box appears for select directory path.

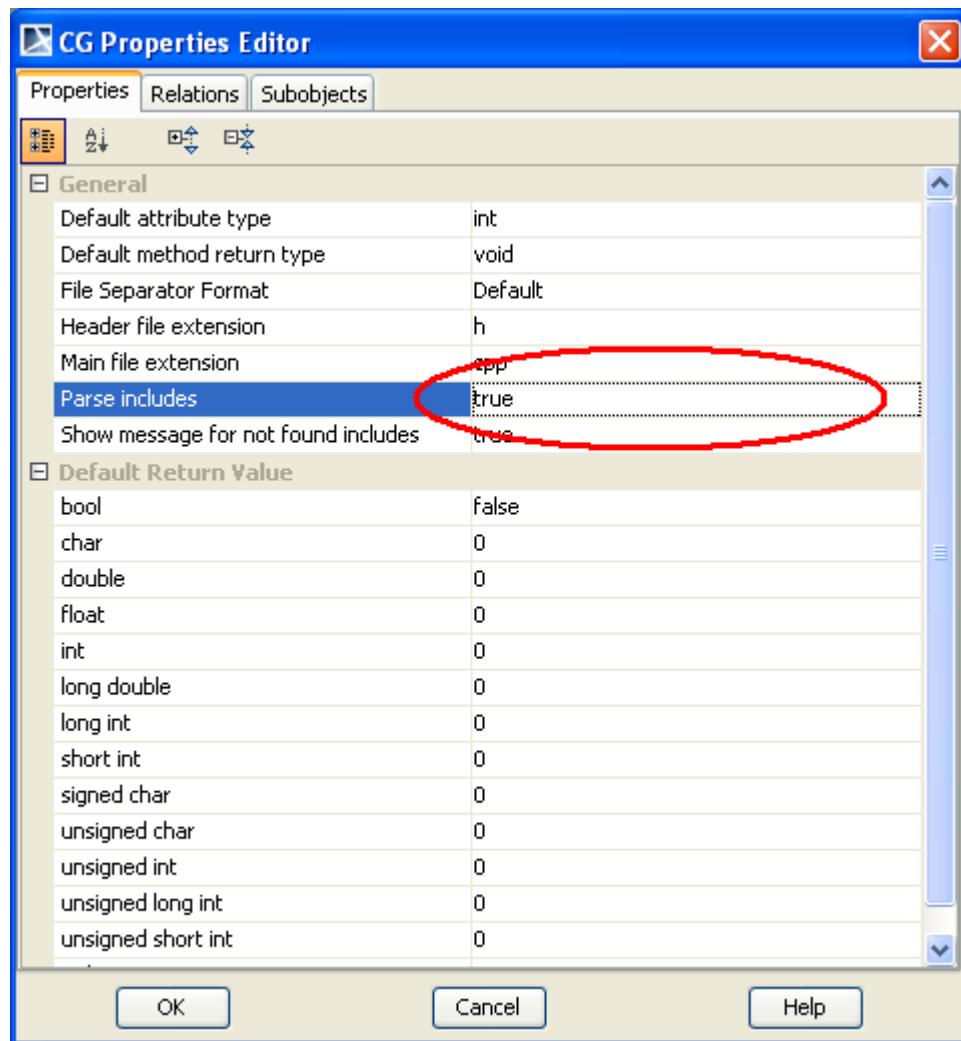


Remove	Remove the selected Include path.
Up	Move the selected Include path upward.
Down	Move the selected Include path downward.

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



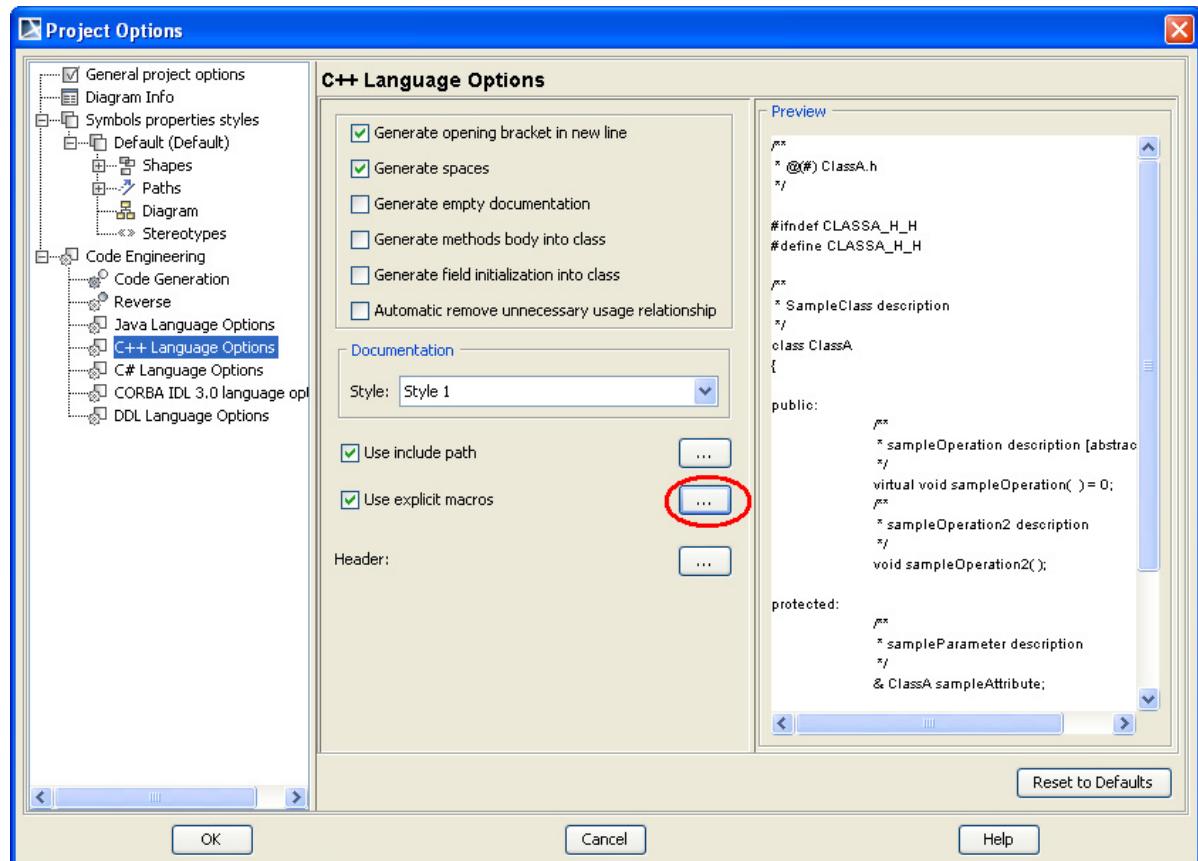
Finally set the “Parse includes” property to “true”.

**Note**

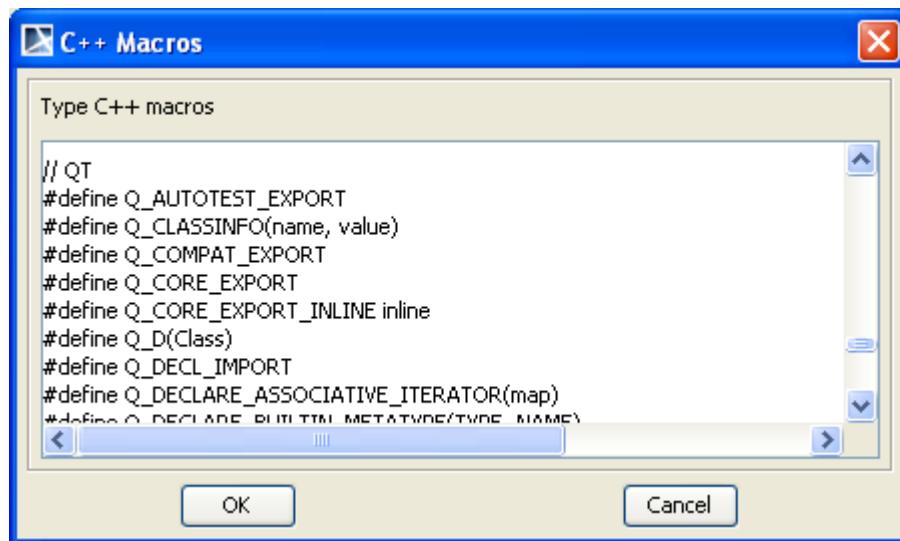
If the user sets the “Parse Includes” property to “false”, then during reverse engineering of the C++ code, MagicDraw will not parse inside the header file and all the unknown type will be created in the “Default” package of the model. This way the user can solve some keyword specific problems in the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

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

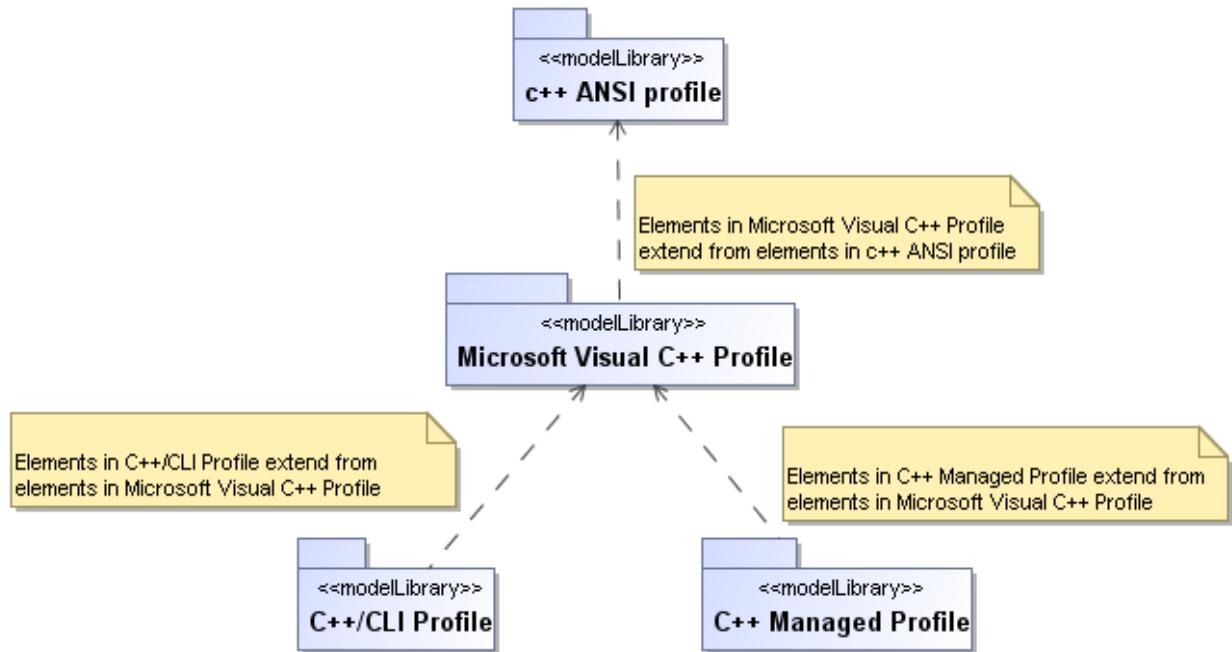


Figure 79 -- Profile Dependency

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.

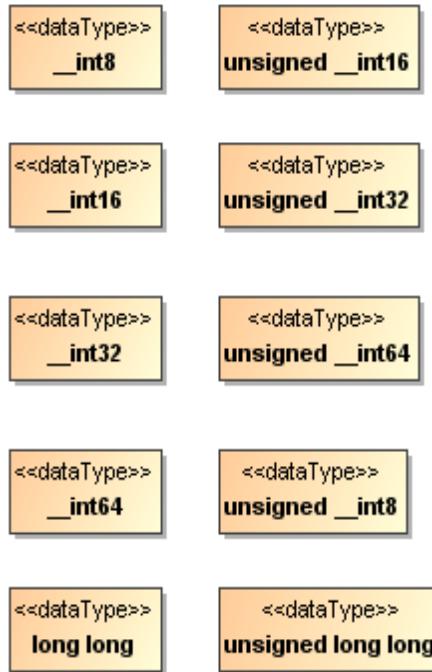


Figure 80 -- Microsoft Visual C++ Data Type

Stereotype

NOTE

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

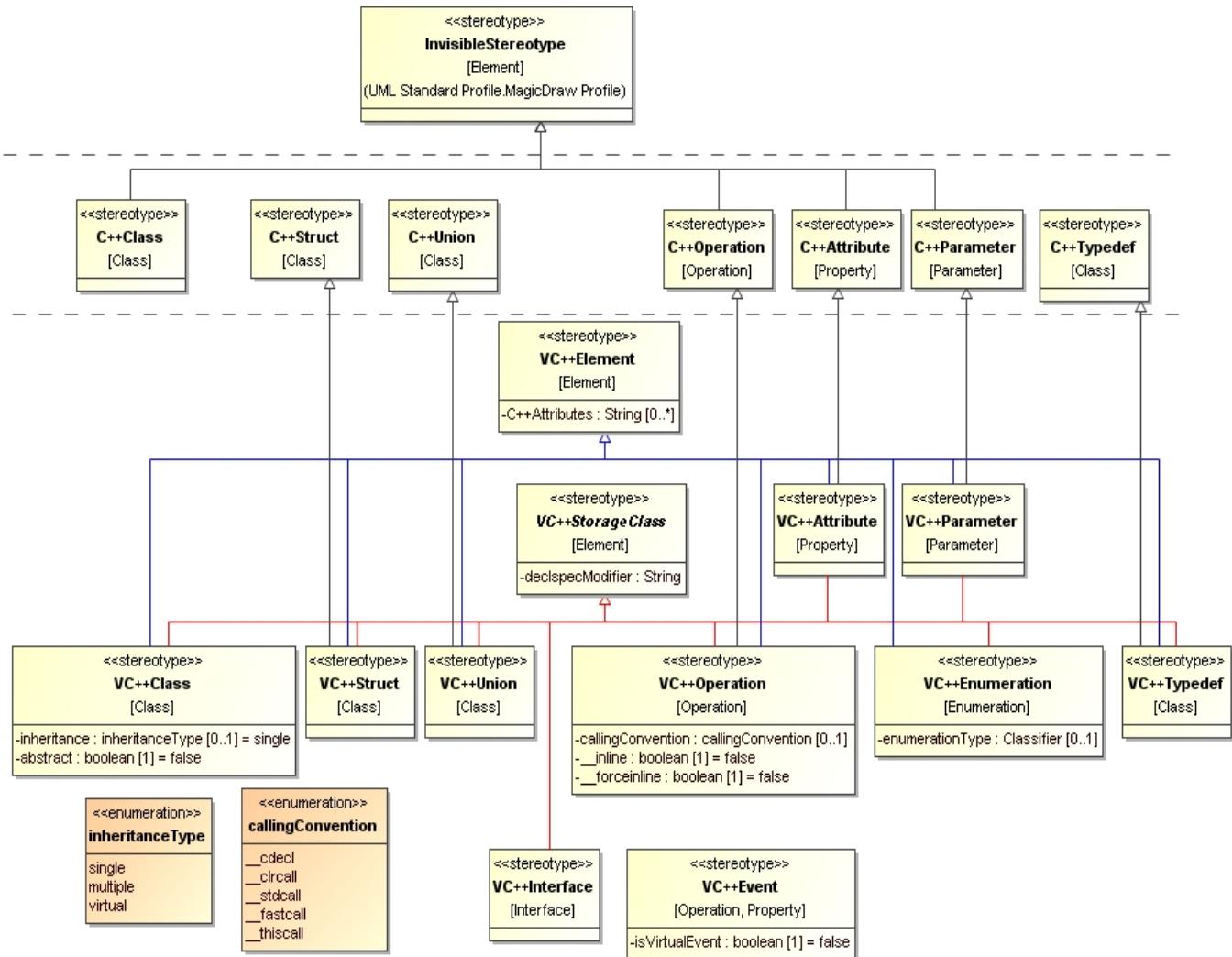


Figure 81 -- Microsoft Visual C++ Stereotypes

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 inheritanceType	Represent the utilization of VC++ keywords, __single_inheritance , __multiple_inheritance and __virtual_inheritance .
abstract	boolean[1]=false	Represent the utilization of keyword __abstract or abstract, depending on C++ dialects.

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

VC++Attribute

Name	Meta class	Constraints
VC++Attribute	Property	

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

VC++Operation

Name	Meta class	Constraints
VC++Operation	Operation	

Tag	Type	Description

callingConvention	callingConvention [0..1]	Represent the utilization of keywords. __cdecl __clrcall __stdcall __fastcall __thiscall
__inline	boolean [1] = false	Represent the utilization of __inline keyword.
__forceinline	boolean [1] = false	Represent the utilization of __forceinline keyword.

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

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 __declspec See Extended storage-class attributes with __declspec Extended storage-class attributes with __declspec

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 __interface keyword.

VC++Event

Name	Meta class	Description
VC++Event	Operation, Property	Represent interface, method or member declaration with __event keyword
Tag	Type	Description
isVirtualEvent	boolean[1]=false	Support virtual event as shown in the example below. // data member as event virtual __event ClickEventHandler* OnClick;

Enumeration

inheritanceType

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

Literal	Description
single	Representation of __single_inheritance keyword
multiple	Representation of __multiple_inheritance keyword
Virtual	Representation of __virtual_inheritance keyword

callingConvention

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

Literal	Description
---------	-------------

<code>__cdecl</code>	Represent the utilization of <code>__cdecl</code> modifier.
<code>__clrcall</code>	Represent the utilization of <code>__clrcall</code> modifier.
<code>__stdcall</code>	Represent the utilization of <code>__stdcall</code> modifier.
<code>__fastcall</code>	Represent the utilization of <code>__fastcall</code> modifier.
<code>__thiscall</code>	Represent the utilization of <code>__thiscall</code> 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

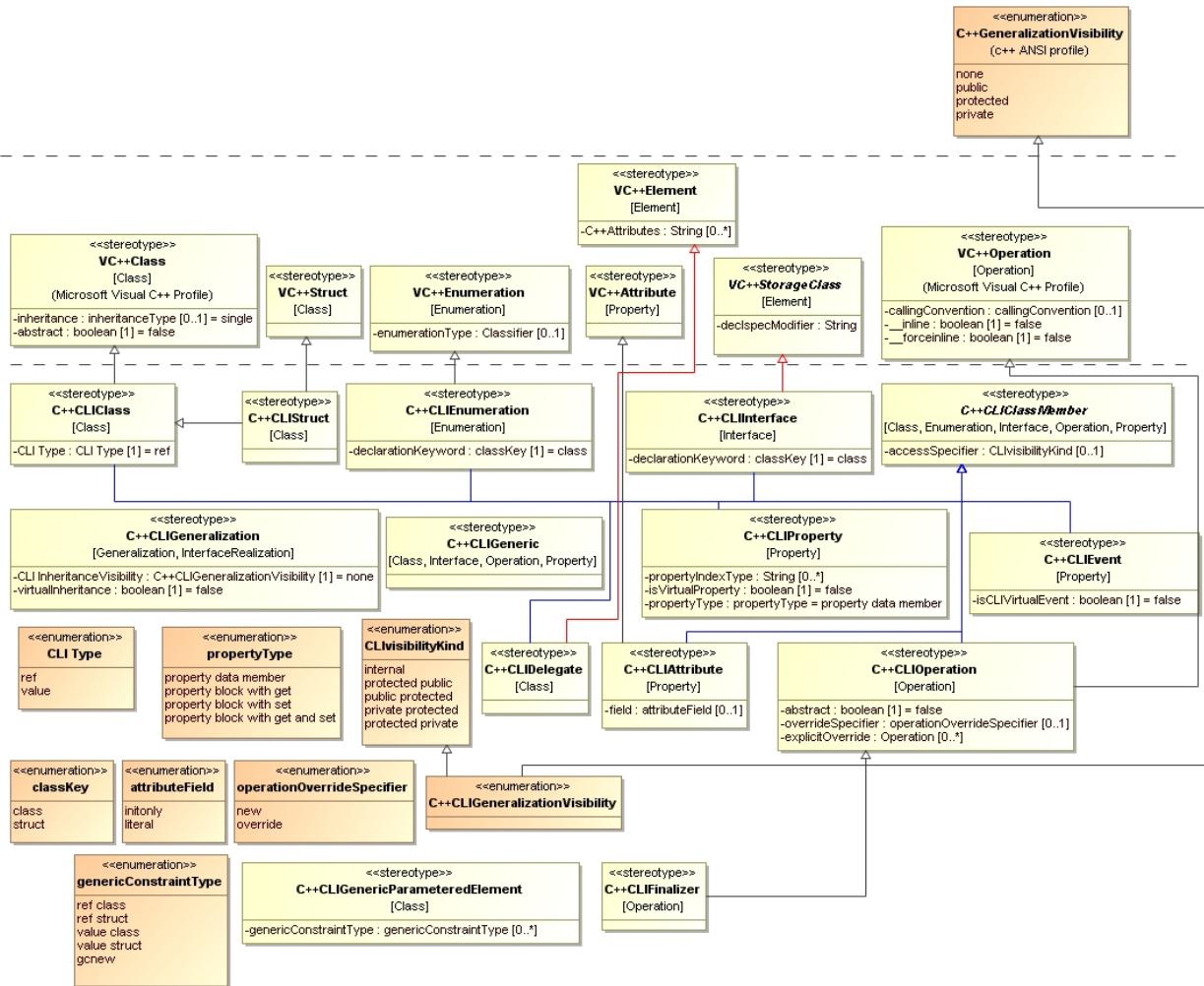


Figure 82 -- C++/CLI Stereotypes

C++CLIClass

Name	Meta class	Constraints
C++CLIClass	Class	

Tag	Type	Description
CLI Type	CLI Type[1]=ref (Enumeration)	Represent the utilization of ref class or value class keywords for defining CLR class.

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

C++CLIStruct

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

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

C++CLInterface

Name	Meta class	Constraints
C++CLI	Interface	

Tag	Type	Description
declarationKeyword	classKey [1] = class (Enumeration) See ref class, ref struct, value class, value struct, interface class, interface structref class, ref struct, value class, value struct, interface class, interface struct	Represent the utilization of interface class or interface struct . keywords.

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

C++CLIAtribute

Name	Meta class	Constraints
C++CLIAtribute	Property	

Tag	Type	Description
field	attributeField [0..1] (Enumeration)	Represent the usage of initonly or literal keywords.

Inherited tag	Type	Description

accessSpecifier <code><< C++CLIClassMember>></code>	CLIVisibilityKind [0..1] (Enumeration)	Represent the usage of visibility kind introduce in C++/CLI including <ul style="list-style-type: none">• internal• protected public• public protected• private protected• protected private
declspecModifier <code><<VC++StorageClass>></code>	String	For keeping extended declaration modifier of <code>__declspec</code> See Extended storage-class attributes with <code>__declspec</code> Extended storage-class attributes with <code>__declspec</code>
C++Attributes <code><<VC++Element>></code>	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 abstract keyword.
overrideSpecifier	operationOverrideSpecifier [0..1] (Enumeration) See Override Specifiers Override Specifiers	Represent the utilization of C++/CLI operation override specifiers, new and override .
explicitOverride	Operation[0..*]	Represent the usage of explicit override feature.

Inherited tag	Type	Description
accessSpecifier <code><< C++CLIClassMember>></code>	CLIVisibilityKind [0..1] (Enumeration)	Represent the usage of visibility kind introduce in C++/CLI including <ul style="list-style-type: none"> • internal • protected public • public protected • private protected • protected private
calling convention <code><<VC++Operation>></code>	callingConvention [0..1]	Represent the utilization of keywords. <code>__cdecl</code> <code>__clrcall</code> <code>__stdcall</code> <code>__fastcall</code> <code>__thiscall</code>
C++Attributes <code><<VC++Element>></code>	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 <code><<C++Attribute>></code> or other stereotypes derived from <code><<C++Attribute>></code>

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"> ● property data member ● property block with get ● property block with set ● property block with get and set

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

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 delegate 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 event keyword.
Tag	Type	Description
isCLIVirtualEvent	boolean[1]=false	Specify whether the event is virtual or not.

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

C++CLIGeneric

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

C++CLIGenericParameteredElement

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

C++CLIGeneralization

Name	Meta class	Constraints
C++CLIGeneralization	Generalization	
Tag	Type	Description
CLIIInheritanceVisibility	C++CLIGeneralizationVisibility [1] = none (Enumeration)	
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 new keyword to qualify override behavior for derivation. In VC++, new 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 override keyword to qualify override behavior for derivation. In VC++, override 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++CLIAtribute>>**. 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 initonly keyword. initonly indicates that variable assignment can only occur as part of the declaration or in a static constructor in the same class.

literal	<p>Represent the utilization of literal keyword.</p> <p>It is the native equivalent of static const variable.</p> <p>Constraint: Is Static and Is Read Only must be set to true.</p>
---------	---

classKey

Literal	Description
class	Represent the keyword that has the word, class .
struct	Represent the keyword that has the word, struct .

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

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

CLIVisibilityKind

Literal	Description
internal	Represent the internal visibility.
protected public	Represent the protected public visibility.
public protected	Represent the public protected visibility.
private protected	Represent the private protected visibility.
protected private	Represent the protected private visibility.

genericConstraintType

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

C++CLIGeneralizationVisibility

C++CLIGeneralizationVisibility is an enumeration inheriting from C++GeneralizationVisibility and CLIVisibilityKind, 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.

Stereotype

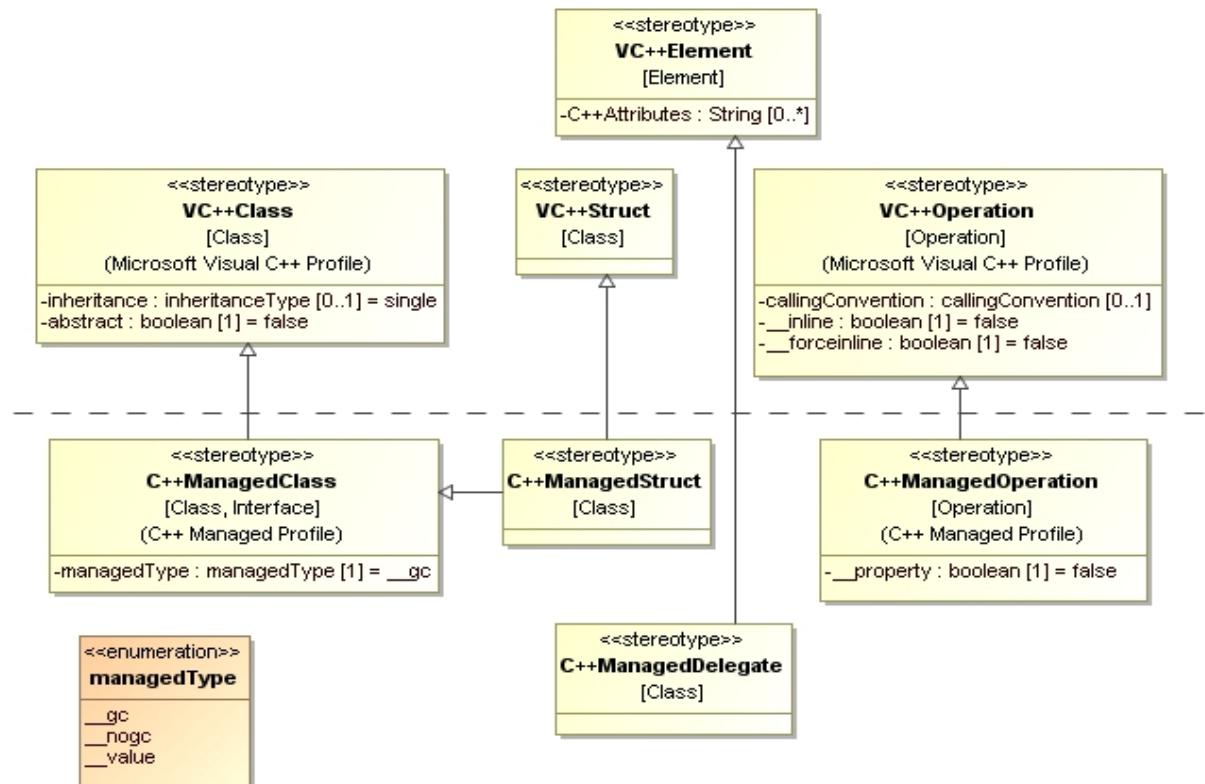


Figure 83 -- C++ Managed Stereotypes

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)	

C++ManagedStruct

Name	Meta class	Constraints
C++ManagedClass	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 __property 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

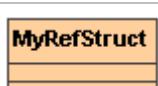
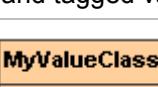
managedType

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

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 {};
```

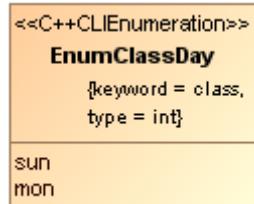
Code	MD-UML
<pre>ref class MyRefClass { };</pre>	 <p>with <<C++CLIClass>> and tagged value managedType = ref</p>
<pre>ref struct MyRefStruct { };</pre>	 <p>with <<C++CLIStruct>> and tagged value managedType = ref</p>
<pre>value class MyValueClass { };</pre>	 <p>with <<C++CLIClass>> and tagged value managedType = value</p>
<pre>value struct MyValueStruct { };</pre>	 <p>with <<C++CLIStruct>> and tagged value managedType = value</p>
<pre>interface class MyInterfaceClass { };</pre>	 <p>with <<C++CLIInterface>> and tagged value keyword = class</p>
<pre>interface struct MyInterfaceStruct { };</pre>	 <p>with <<C++ICLIInterface>> and tagged value keyword = struct</p>

enum class, enum struct

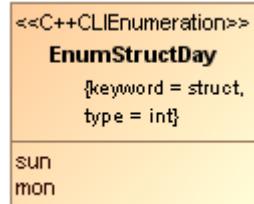
access **enum class** name [: type] { enumerator-list } var;
access **enum struct** name [:type] { enumerator-list } var;

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

```
enum class EnumClassDay
:int
{
    sun,
    mon
};
```



```
enum struct
EnumStructDay :int
{
    sun,
    mon
};
```



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	MD-UML
<pre>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); } };</pre>	<p>MyClass</p> <pre><<C++CLIProperty>>-propertyX : String <<C++CLIProperty>>-propertyY : int <<C++CLIProperty>>-propertyZ : int{index type = int, long}</pre>

deletage

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

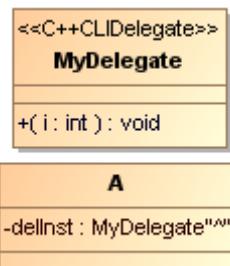
The signature of the function that can be bound to the delegate. The return type of a delegate can be any managed type. For interoperability reasons, it is recommended that the return type of a delegate be a CLS type.

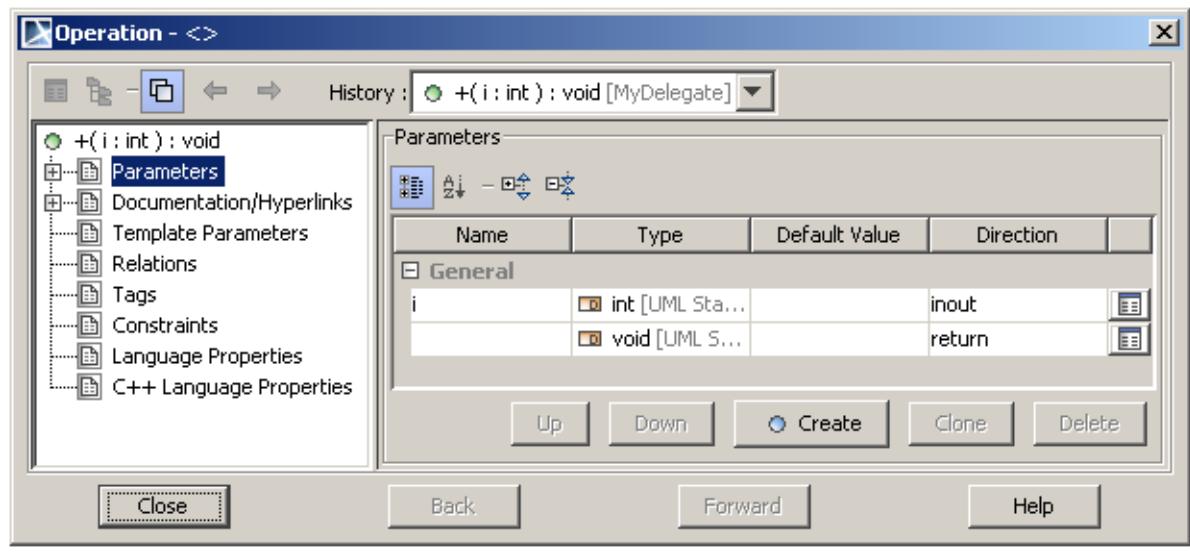
To define an unbound delegate, the first parameter in function_declaration should be the type of the this pointer for the object

Code

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

MD-UML

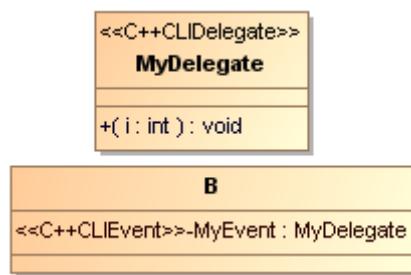


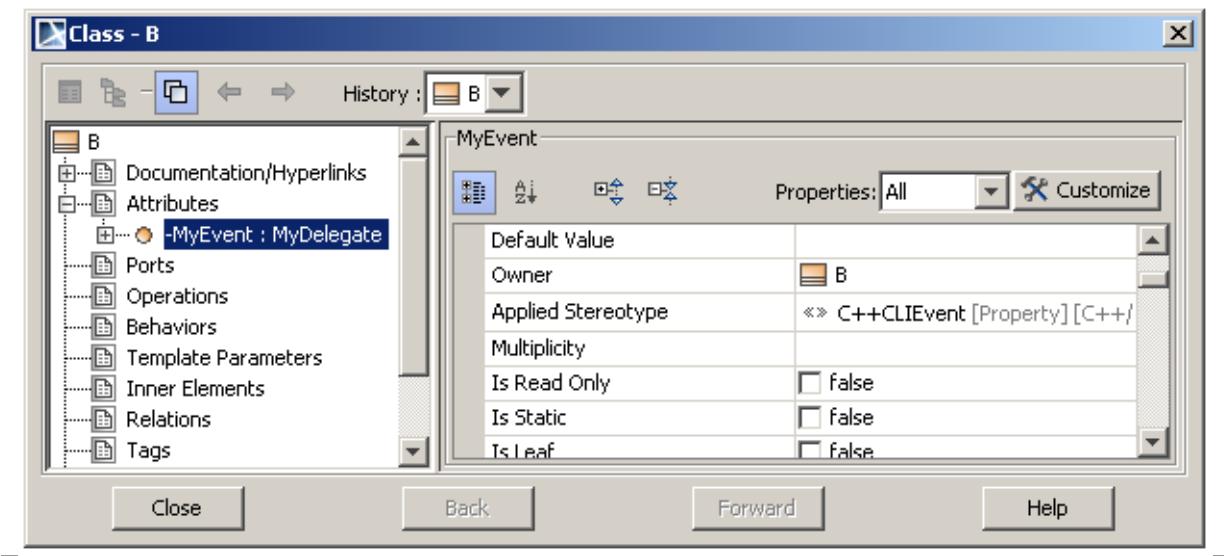


event

```
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);
} // event block
```

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





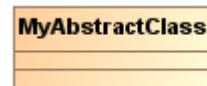
Override Specifiers

abstract

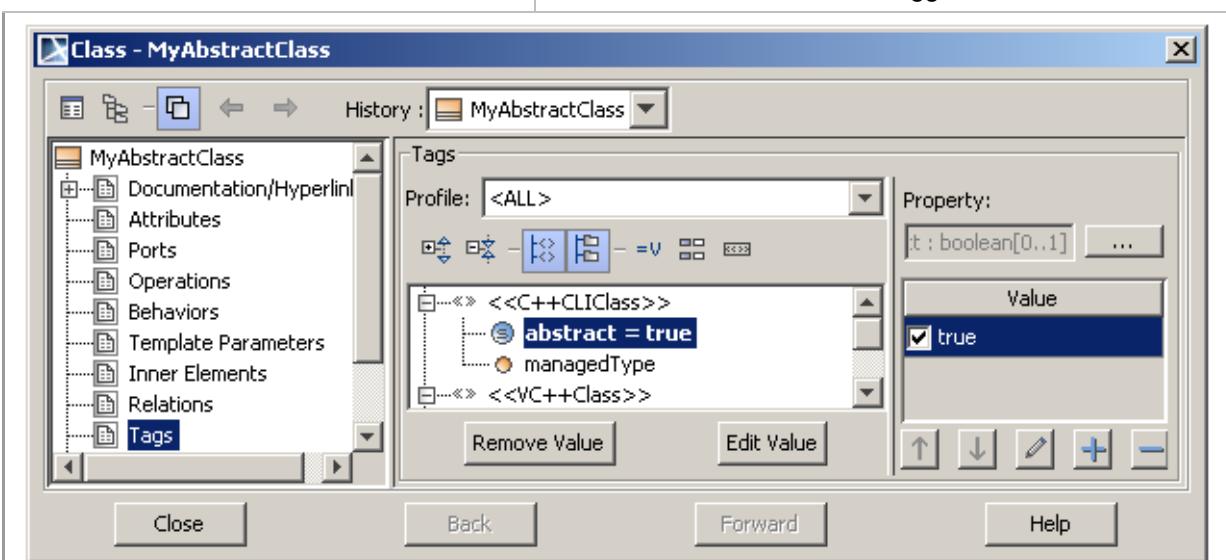
Code

```
class MyAbstractClass abstract
{
};
```

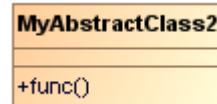
MD-UML



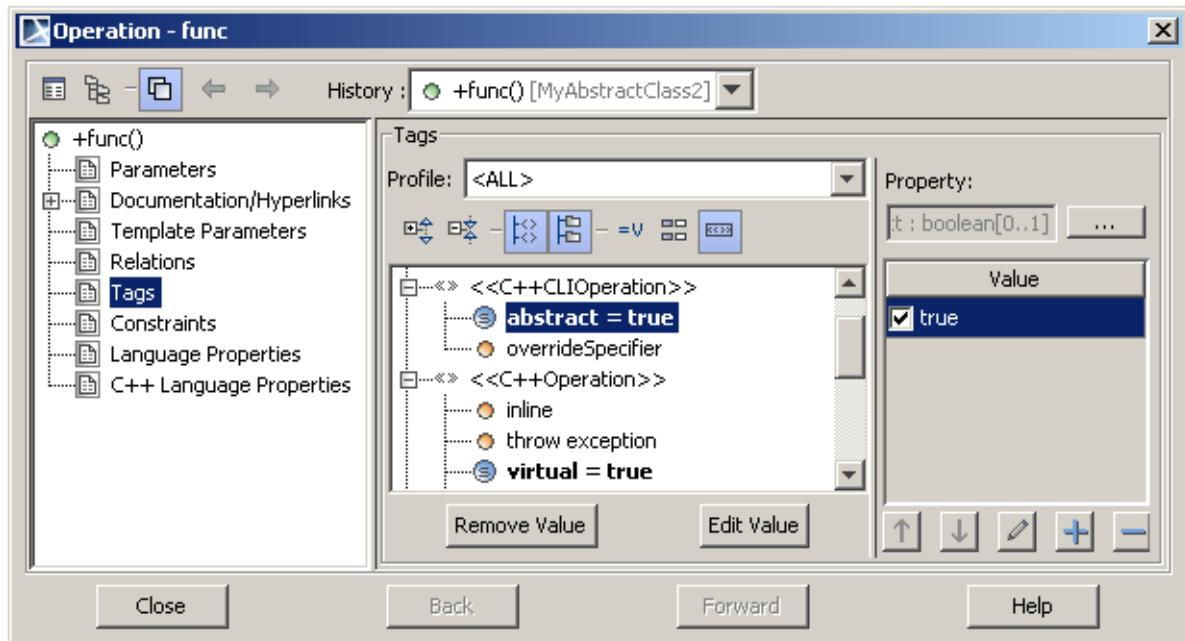
with <<C++CLIClass>> and tagged value **abstract=true**



```
class MyAbstractClass2
{
    public:
    virtual void func() abstract;
};
```



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



new

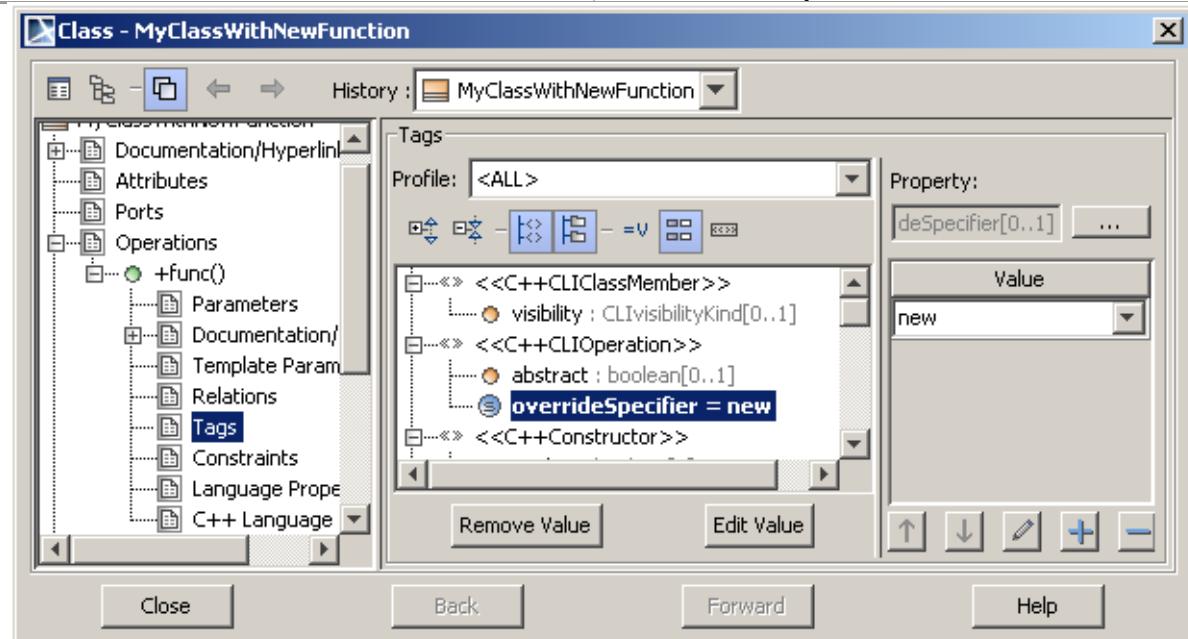
Code

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

MD-UML



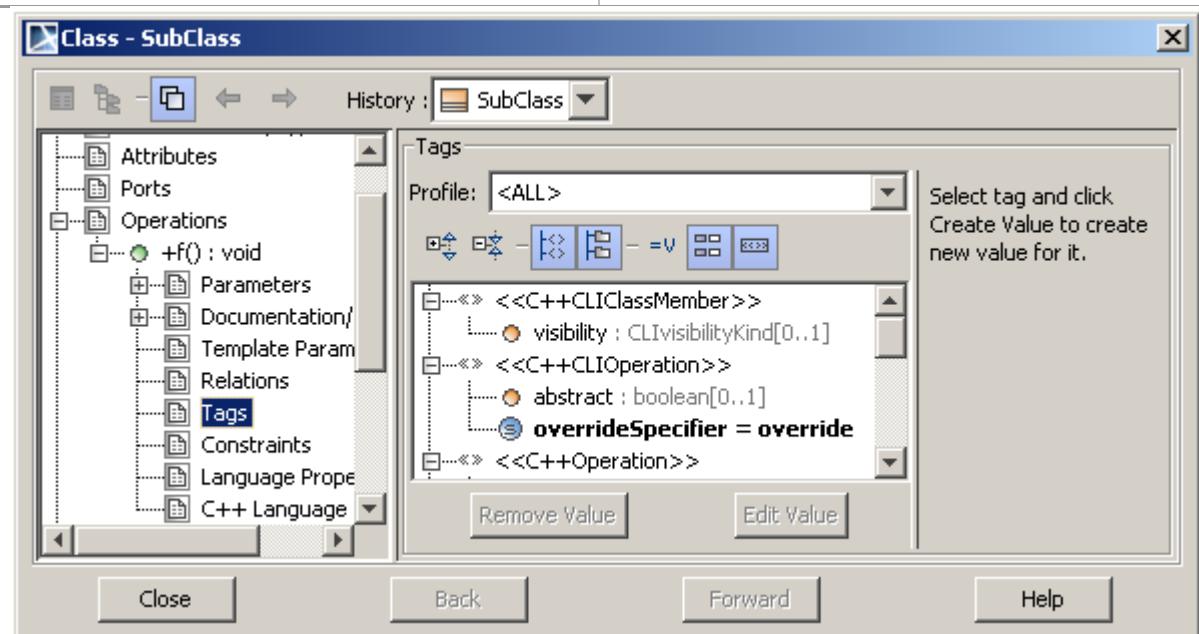
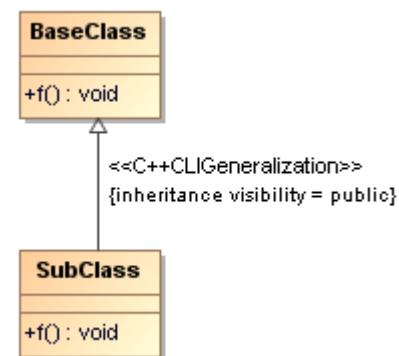
func() with <<C++CLIOperation>> and tagged value **overrideSpecifier = new**



override

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

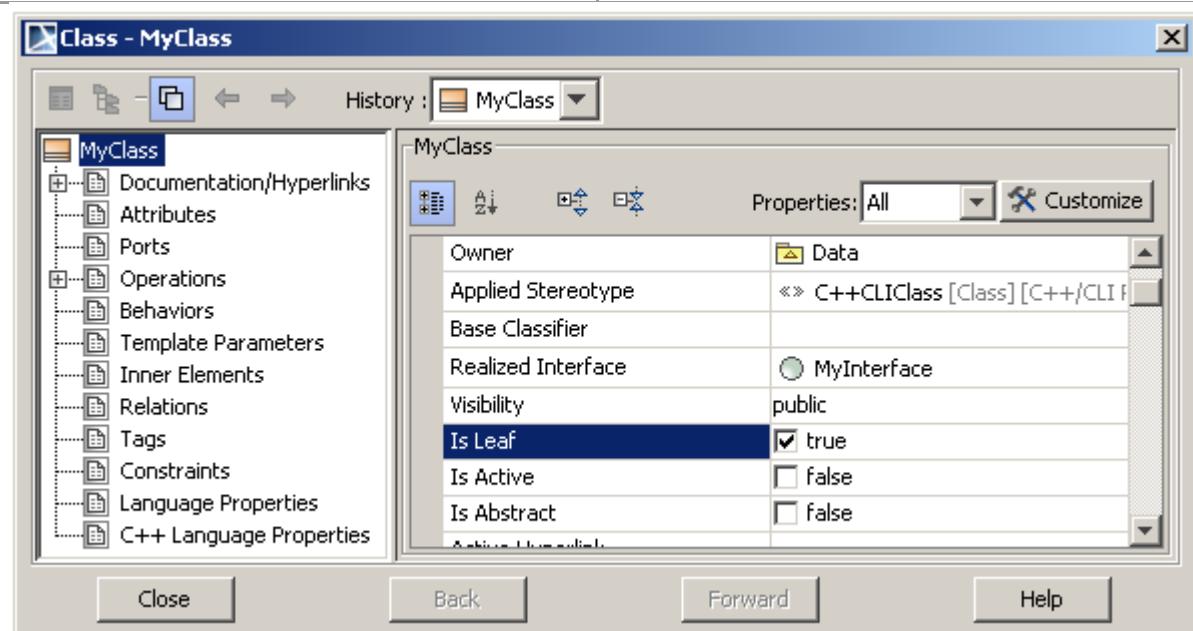
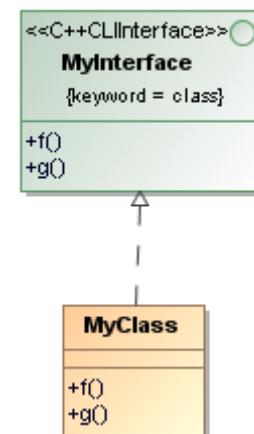
ref class SubClass : public
BaseClass {
    public:
    virtual void f() override {}
};
```



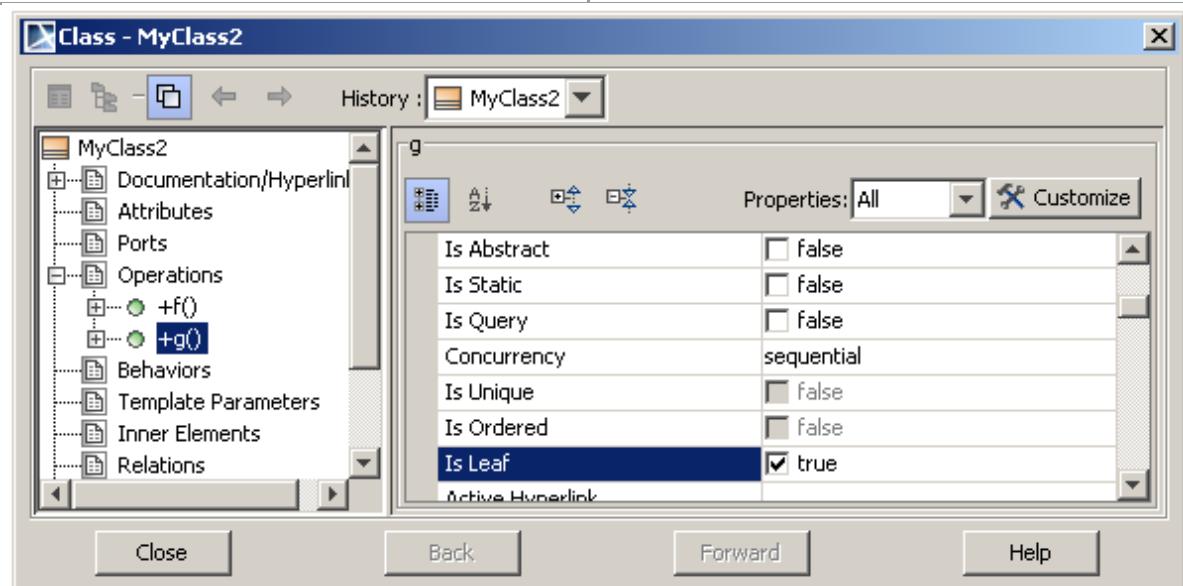
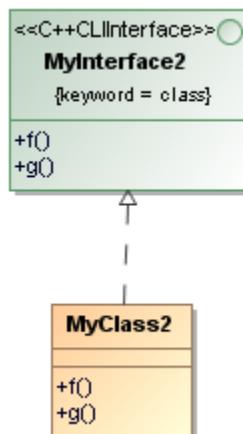
sealed

```
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

```
generic <typename ItemType>
void G(int i) {}
```

MD-UML

```
<<C++Global>>
```

```
<<C++CLIGeneric>>+<ItemType>G( i : int ) : void
```

ItemType with `<<C++TemplateParameter>>` applied and set tagged value **type keyword = typename**.

```
ref struct MyStruct {
    generic <typename Type1>
    void G(Type1 i) {}

    generic <typename Type2>
    static void H(int i) {}
};
```

```
MyStruct
```

```
<<C++CLIGeneric>>+<Type1>G( i : Type1 ) : void
<<C++CLIGeneric>>+<Type2>H( i : int ) : void
```

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

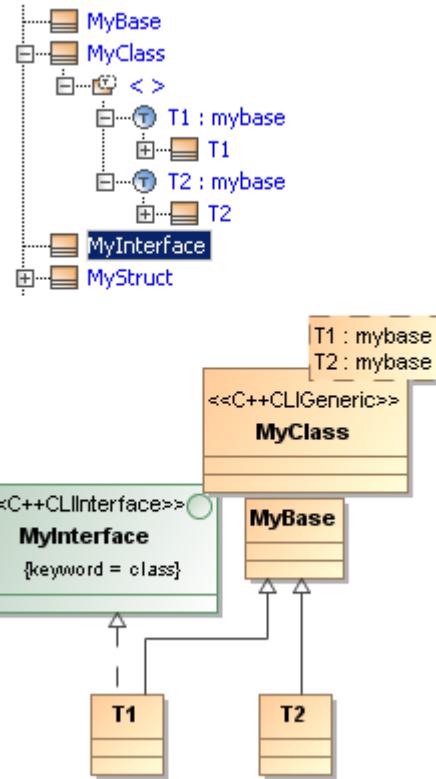
Declarations of any variables of this type. For example: ^identifier[, ...]

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]

[accesibility-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)

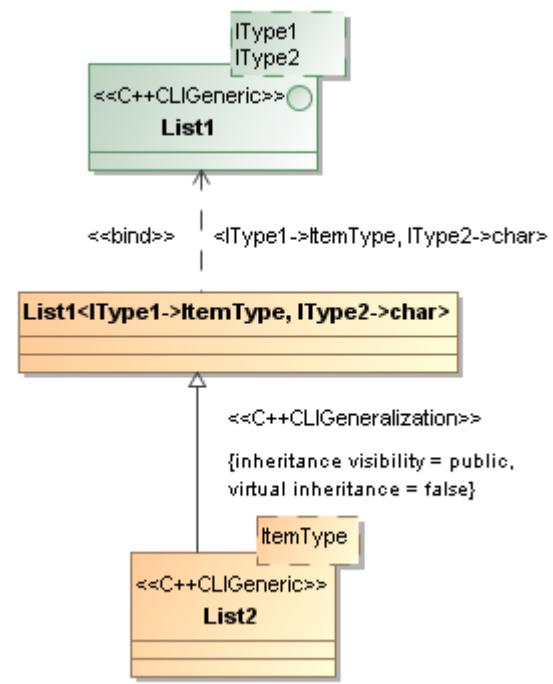
Declarations of variables based on this type.

Code

```
generic <typename IType1, typename
IType2>
public interface class List {
    IType1 x;
};

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

MD-UML



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.

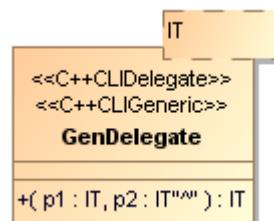
formal-parameters (Optional)

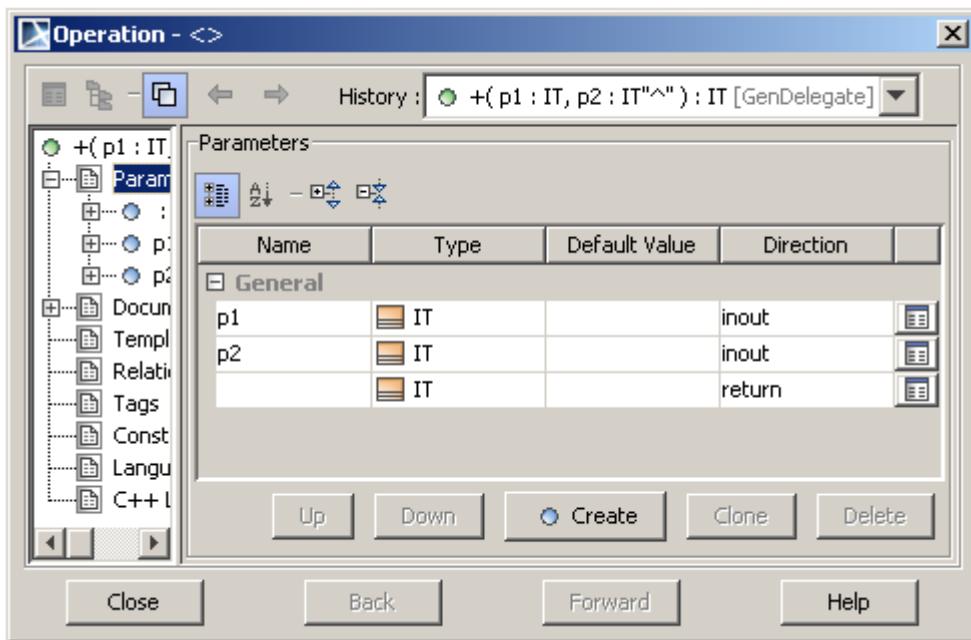
The parameter list of the delegate.

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.

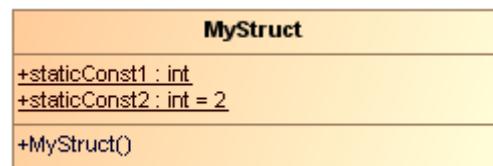
Code

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

    initonly
    static int staticConst2 = 2;

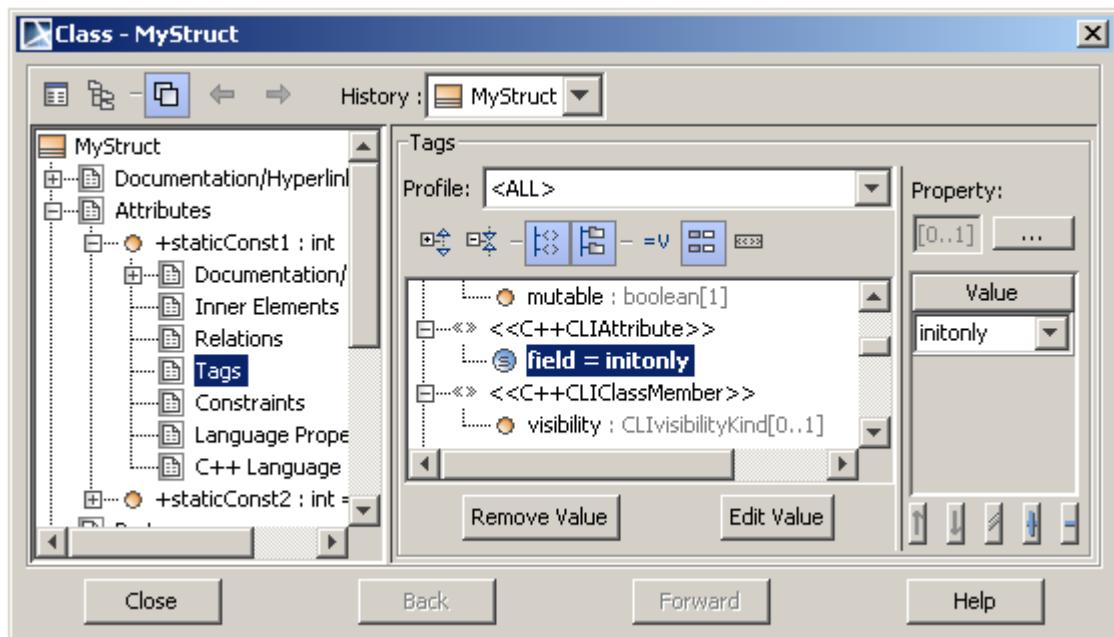
    static MyStruct() {
        staticConst1 = 1;
    }
};
```

MD-UML



MyStruct with <<C++CLIStruct>>

staticConst1 and **staticConst2** have **<<C++CLIAtribute>>** 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
<pre>ref class MyClassWithLiteral { literal int i = 1; };</pre>	
	<p>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</p>

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.

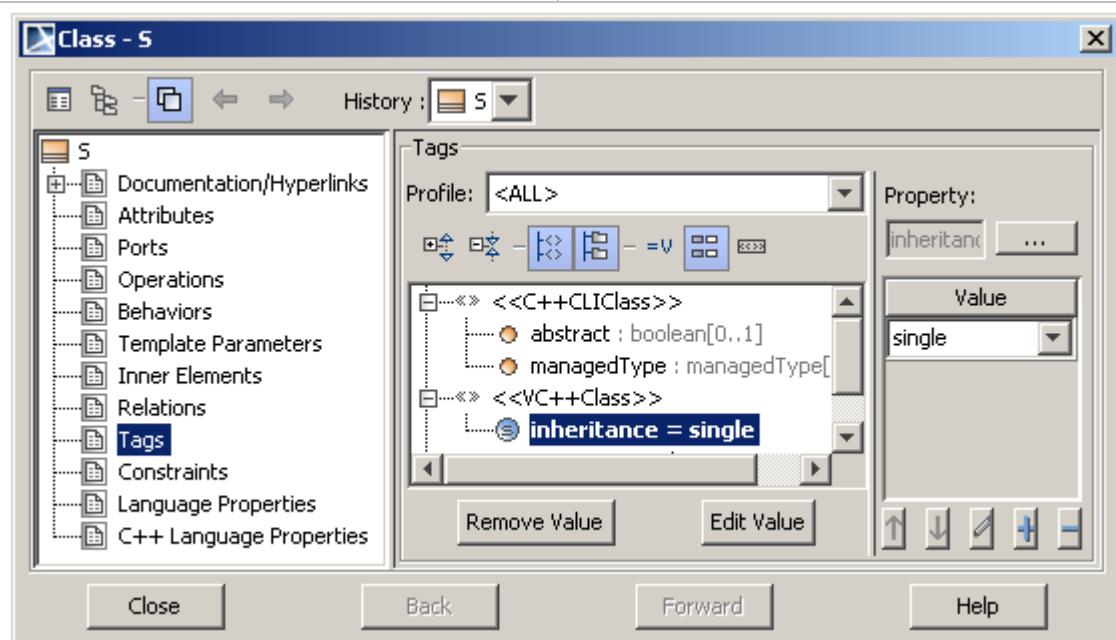
Code

```
class __single_inheritance S;
```

MD-UML



Class with <<VC++Class>>



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	MD-UML
<pre>__interface MyInterface {};</pre>	

__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:

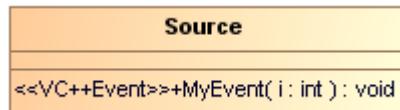
```
__event method-declarator;
__event __interface interface-specifier;
__event member-declarator;
```

Native Events

Code

```
class Source {
public:
    __event void MyEvent(int i);
};
```

MD-UML



Com Events

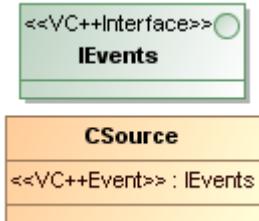
The __interface keyword is always required after __event for a COM event source.

Code

```
__interface IEvents {
};

class CSource {
public:
    __event __interface IEvents;
};
```

MD-UML



__event __interface IEvents;
is mapped to attribute without name

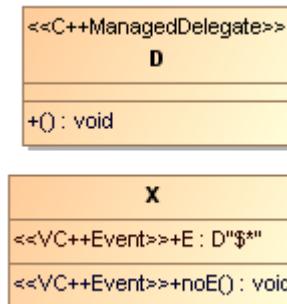
NOTE This mapping produces the exceptional case for syntax checker. The syntax checker has to allow attribute without name for this case.

Managed Events

Code

```
public __delegate void D();
public __gc class X {
public:
    __event D* E;
    __event void noE();
};
```

MD-UML



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

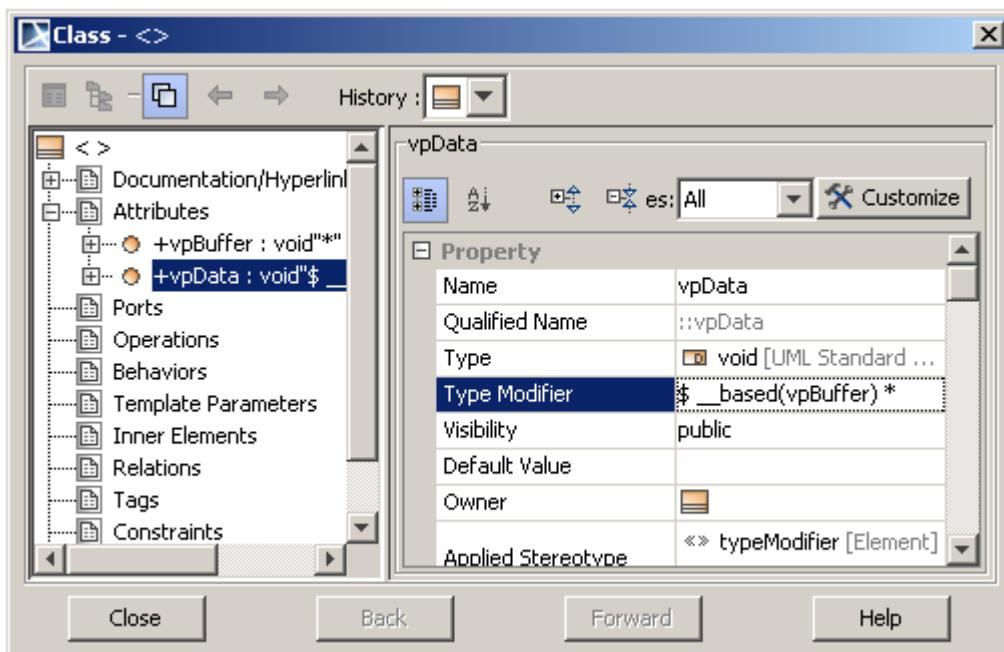
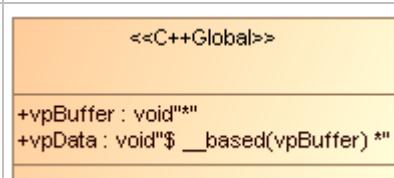
Example

```
// based_pointers1.cpp
// compile with: /c
void *vpBuffer;
struct llist_t {
    void __based( vpBuffer ) *vpData;
    struct llist_t __based( vpBuffer ) *llNext;
};
```

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)]
```

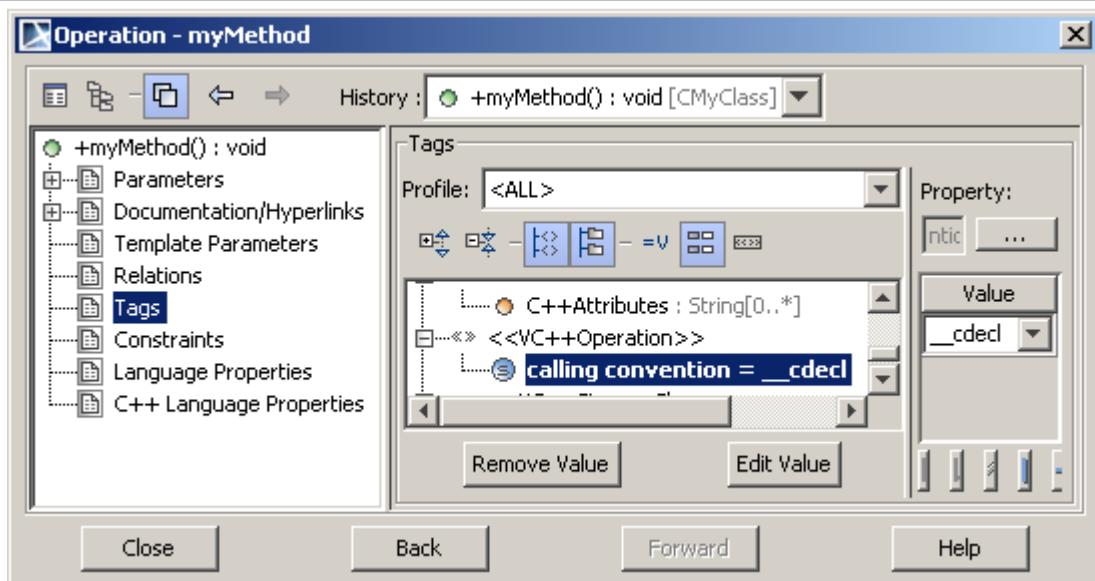
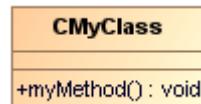
This is the default calling convention for C and C++ programs. Because the stack is cleaned up by the caller, it can do **vararg** functions. The **__cdecl** calling convention creates larger executables than **stdcall**, because it requires each function call to include stack cleanup code. The following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

implementation of this calling convention.

Code

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

MD-UML



__clrcall

return-type __clrcall function-name[(argument-list)]

Specifies that a function can only be called from managed code. Use __clrcall 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

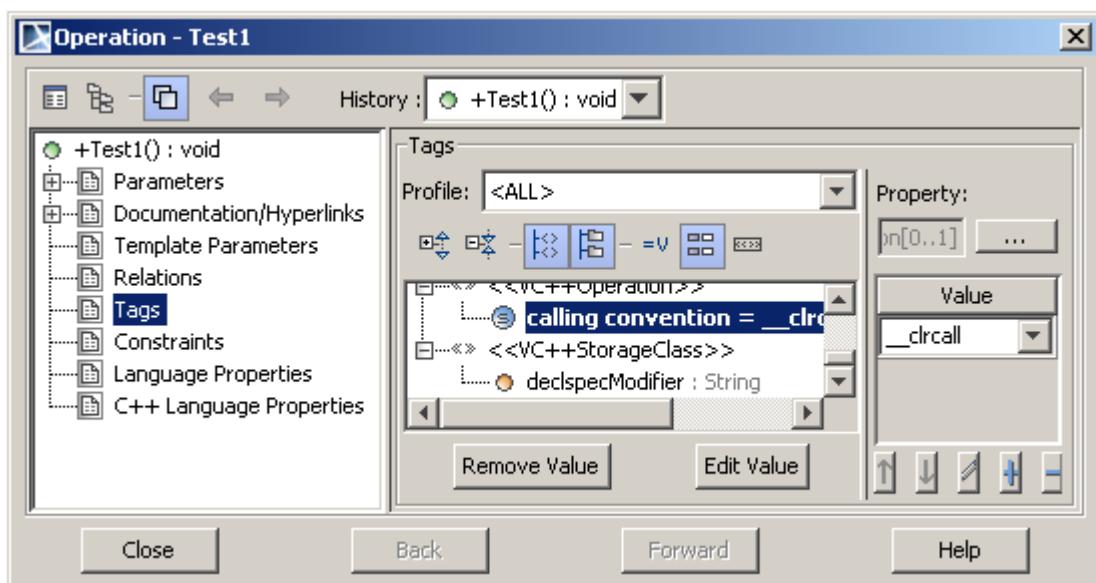
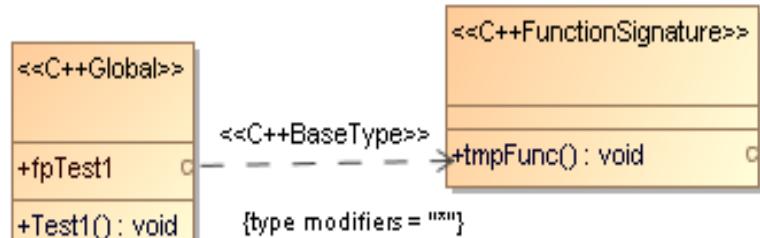
```
// compile with: /clr:oldSyntax /LD  
void __clrcall Test1( ) {}
```

```
void (__clrcall *fpTest1)() = &Test1;
```

Code

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

MD-UML



__stdcall

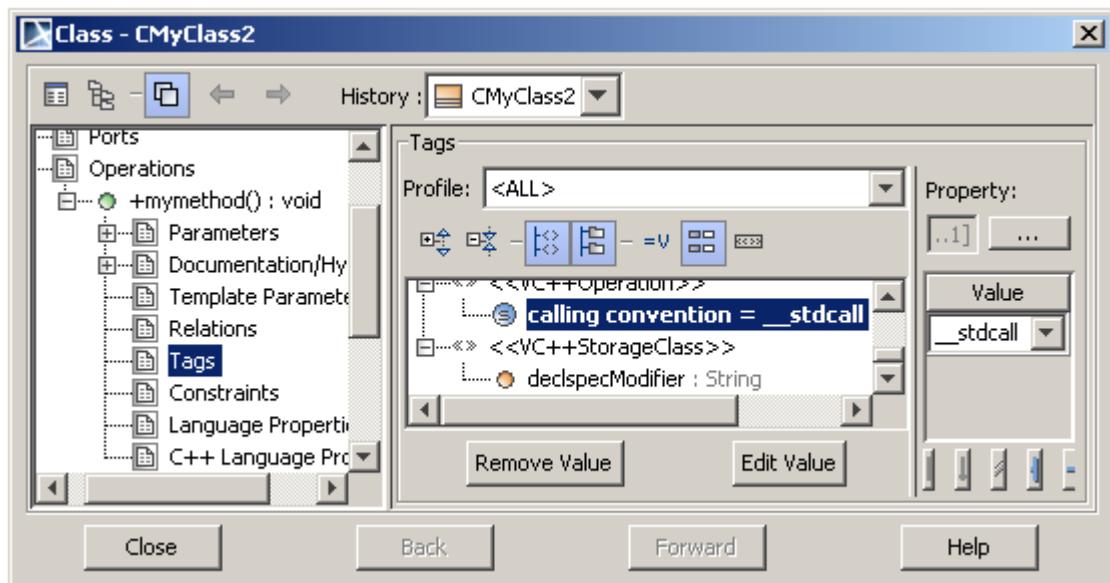
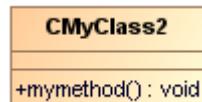
return-type __stdcall function-name[(argument-list)]

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

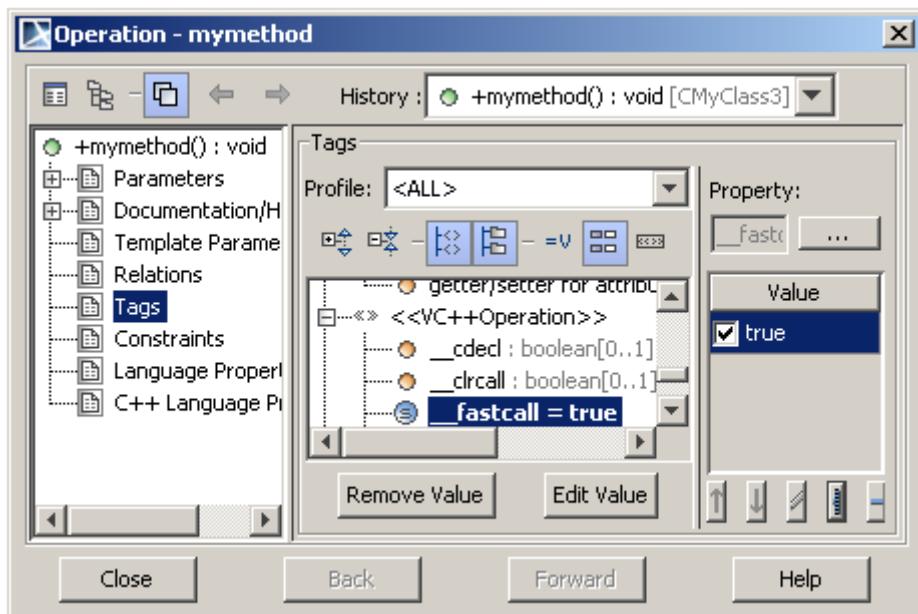
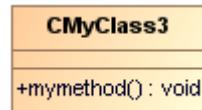
```
class CmyClass3 {  
    void __fastcall mymethod();
```

};

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
```

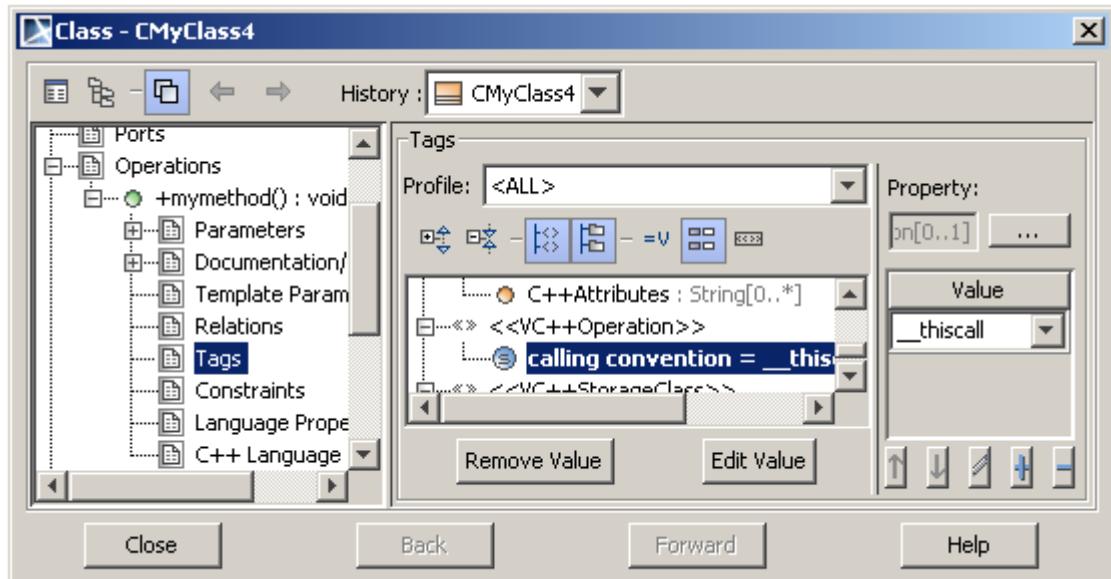
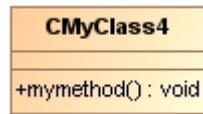
```
class CmyClass4 {
```

```
void __thiscall mymethod();  
  
void __clrcall 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

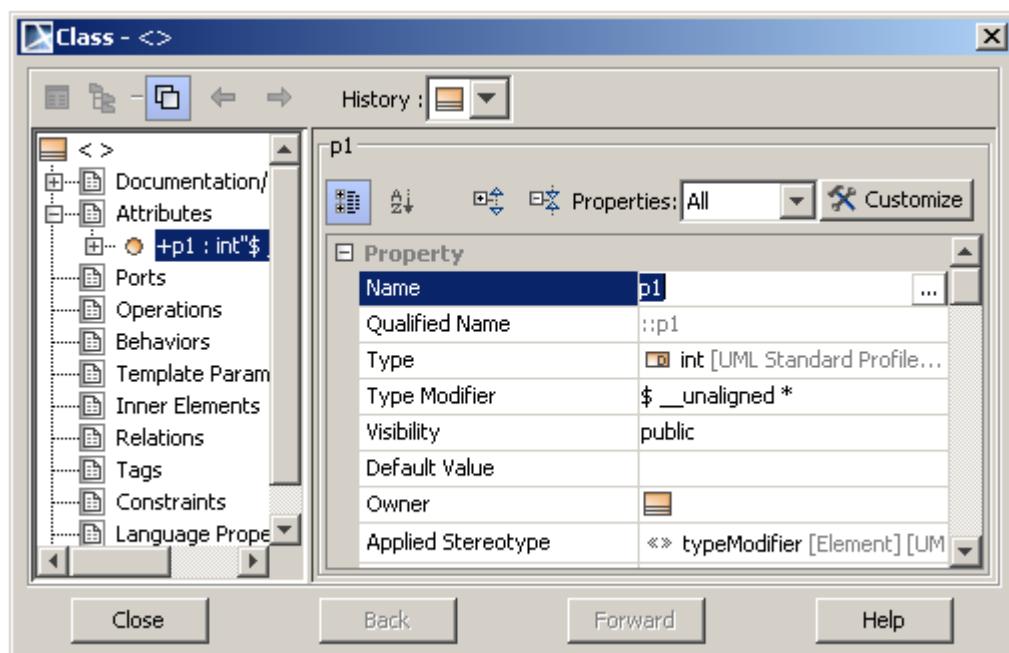
```
// compile with: /c  
// processor: x64 IPF
```

```
#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

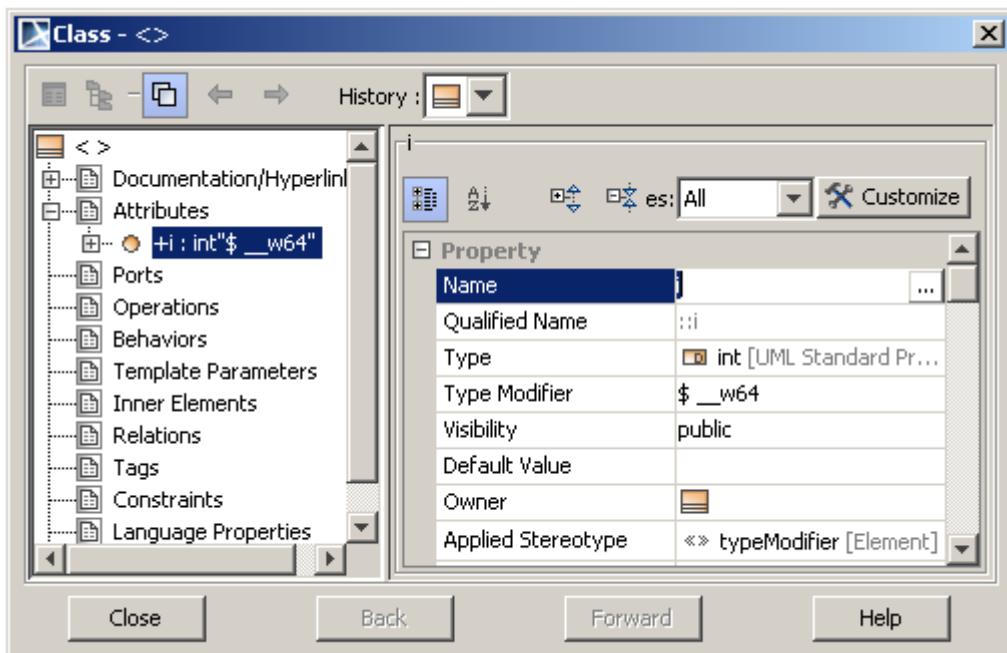
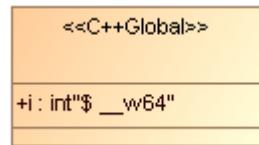
```
// compile with: /W3 /Wp64
typedef int Int_32;
#ifndef _WIN64
typedef __int64 Int_Native;
#else
typedef int __w64 Int_Native;
#endif
int main() {
    Int_32 i0 = 5;
    Int_Native i1 = 10;
    i0 = i1; // C4244 64-bit int assigned to 32-bit int

    // 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
dllexport
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**, **dllexport**, **dllimport**, **jitintrinsic**, **naked**, **noalias**, **noinline**, **noreturn**, **nothrow**, **novtable**, **process**, **restrict**, **selectany**, and **thread**. It also supports these COM-object attributes: **property** and **uuid**.

The **dllexport**, **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 **dllexport** 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

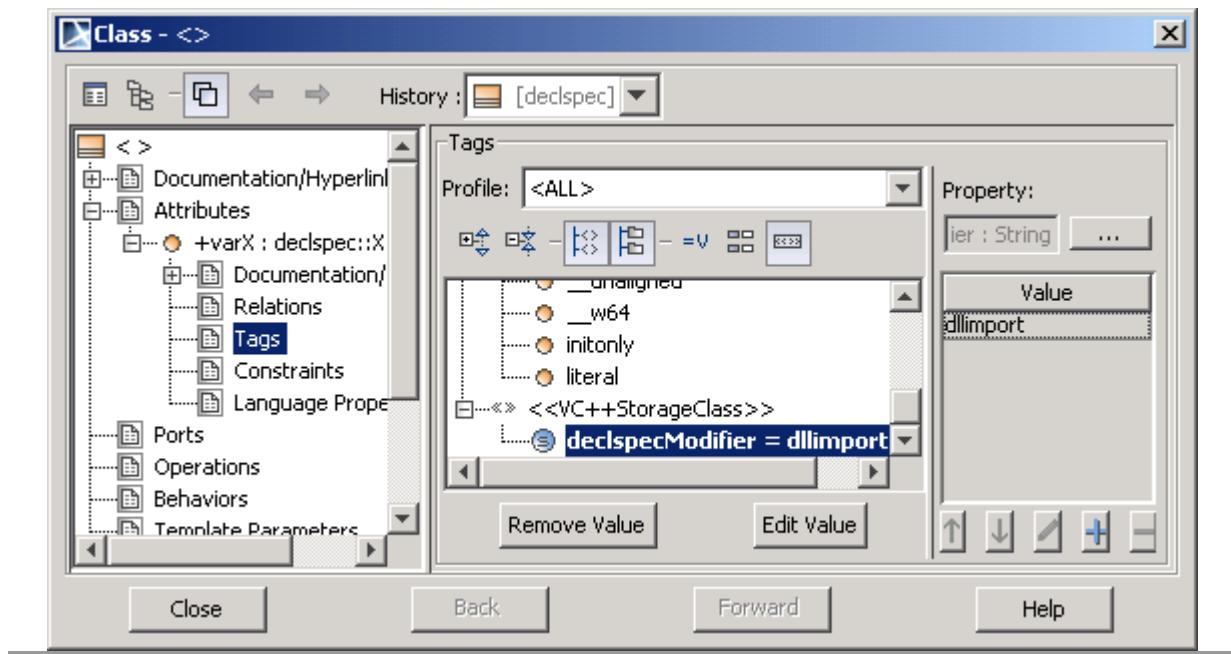
declaration.

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(dllexport) X {};</pre>	<p>UML Class Diagram showing class X with a note: <code><<VC++StorageClass>></code> and <code>[declspecModifier = "dllexport"]</code>.</p>

Code	MD-UML
<pre>__declspec(dllexport) class X {} varX;</pre>	<p>UML Class Diagram showing a global variable varX of type X.</p>



__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;  
};
```

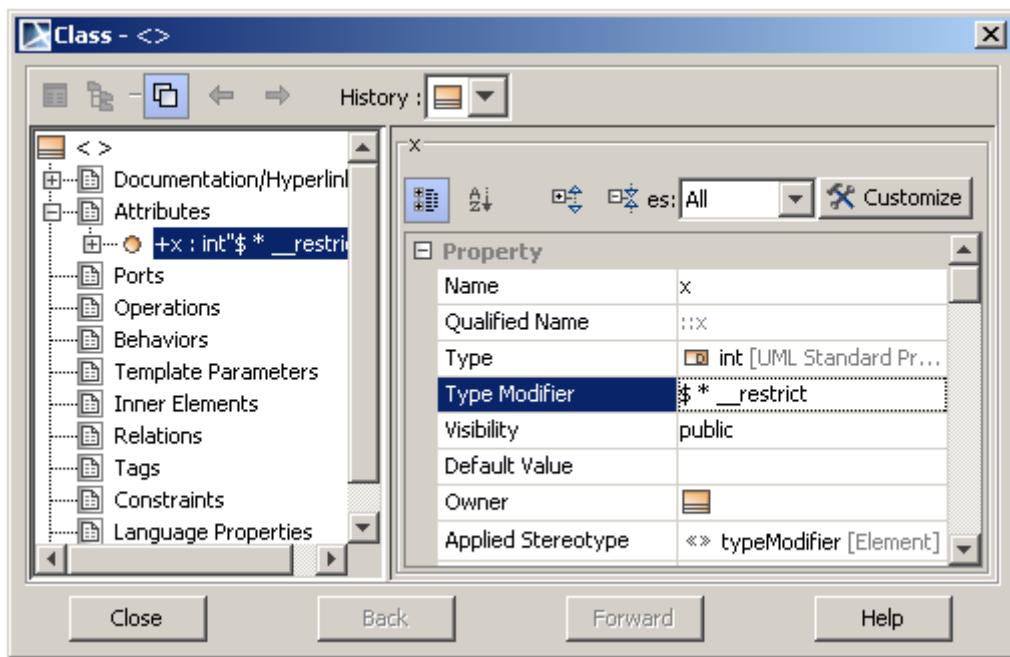
Code

```
int * __restrict x;
```

MD-UML

<<C++Global>>

+x : int"\$ * __restrict"



__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**.

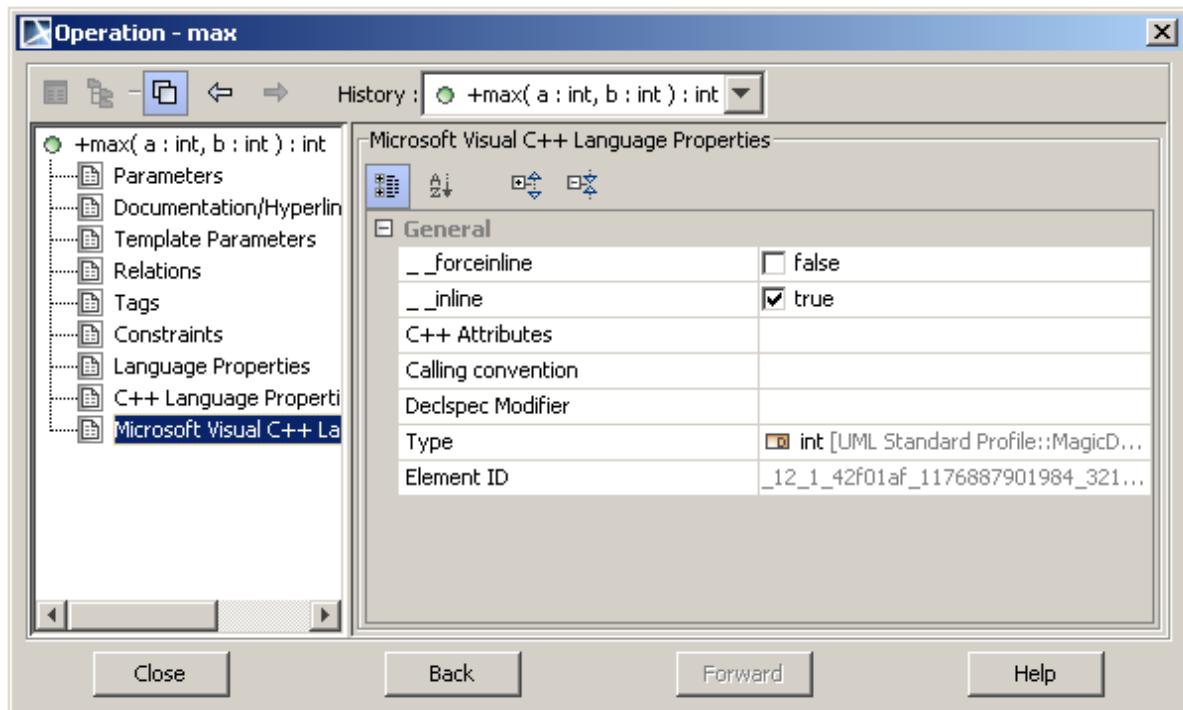
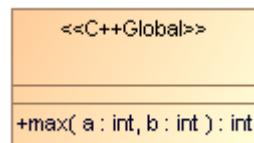
Grammar:

`_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

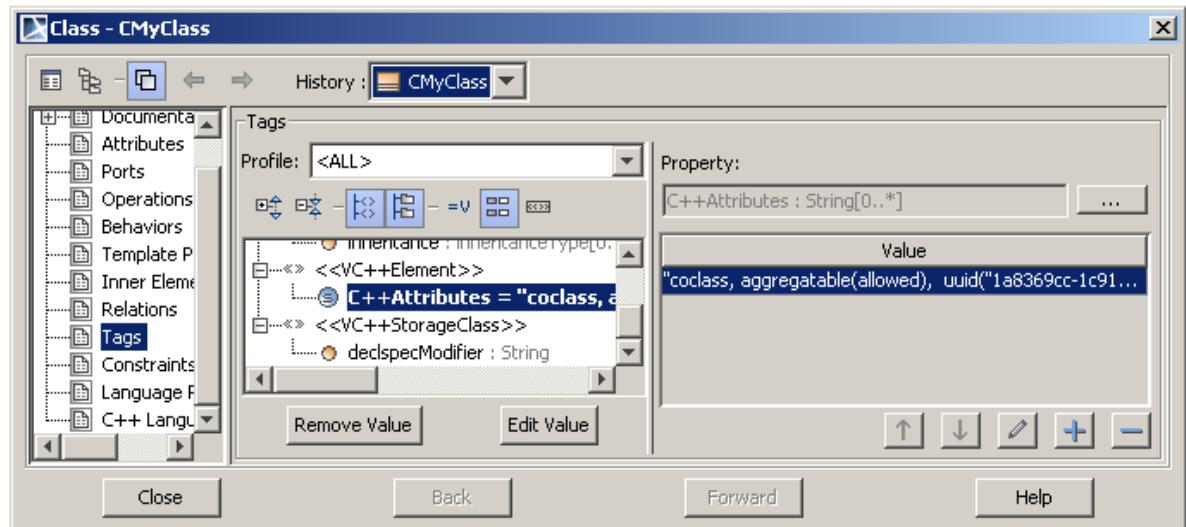
Attributes are designed to simplify COM programming and .NET Framework common language runtime development. When you include attributes in your source files, the compiler works with provider DLLs to insert code or modify the code in the generated object files.

Code

```
[ coclass,  
  aggregatable(allowed),  
  uuid("1a8369cc-1c91-  
  42c4-befa-5a5d8c9d2529") ]  
class CMyClass {};
```

MD-UML

```
<<VC++Element>>  
CMyClass  
[C++Attributes = "coclass, aggregatable(allowed), uuid("1a8369cc-1c91-42c4-befa-5a5d8c9d2529")"]
```



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

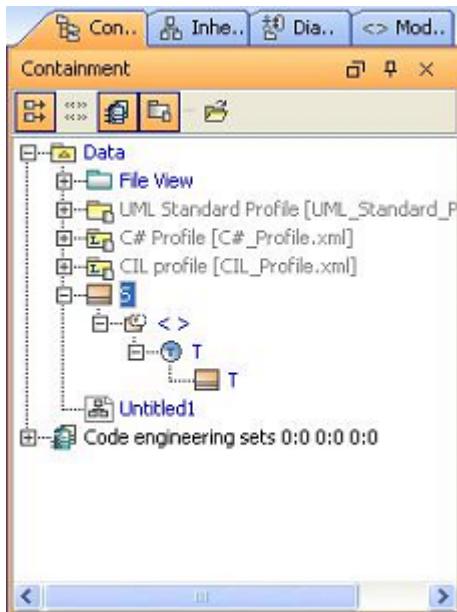
Generics permit classes, structs, interfaces, delegates, and methods to be parameterized by the types of data they store and manipulate. Generic class declaration should be mapped to the UML classifier (class or interface) with a template parameter. (See the detail of Generics in C# Specification chapter 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.

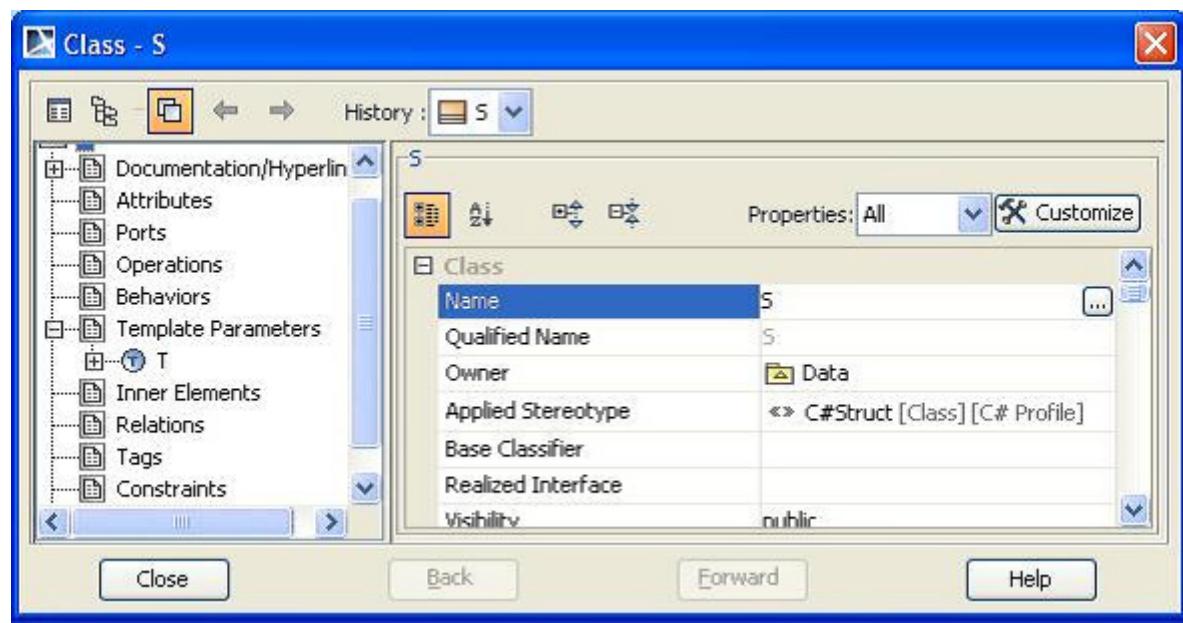
Code	MD-UML
<pre>public class S<T> { }</pre>	



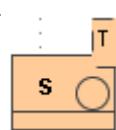
Generic Struct

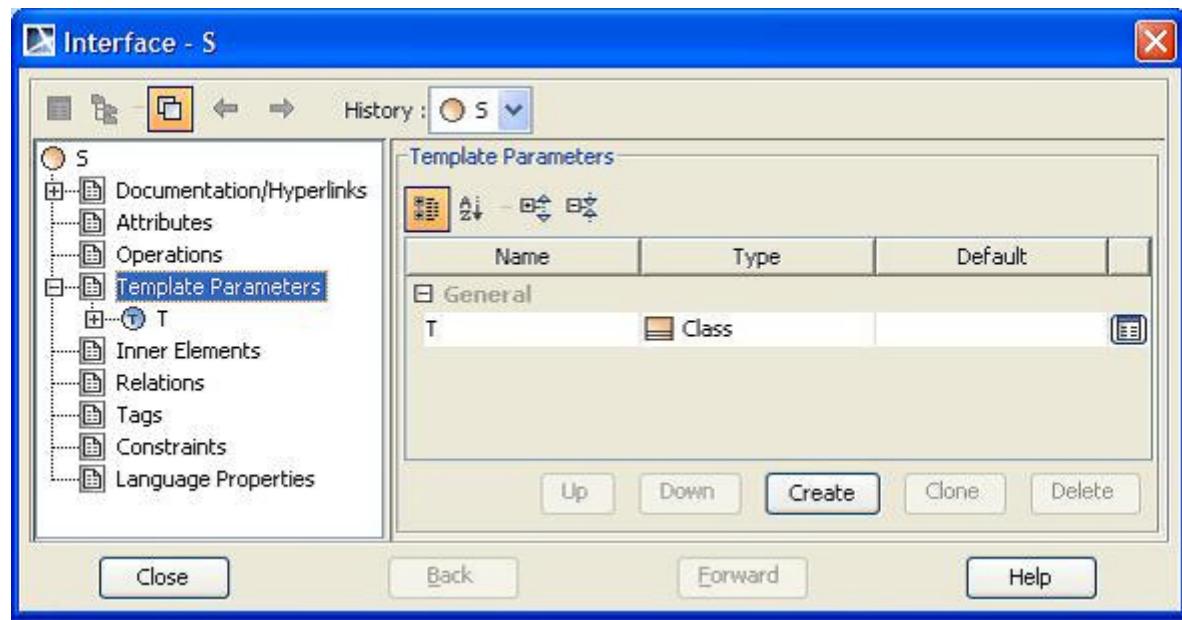
The type parameter of generic struct is created the same as generic class, but we apply <<C#Struct>> stereotype to the model.

Code	MD-UML
<pre>struct S<T> { }</pre>	



Generic Interface

Code	MD-UML
interface S<T> { }	

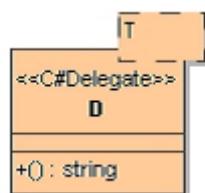


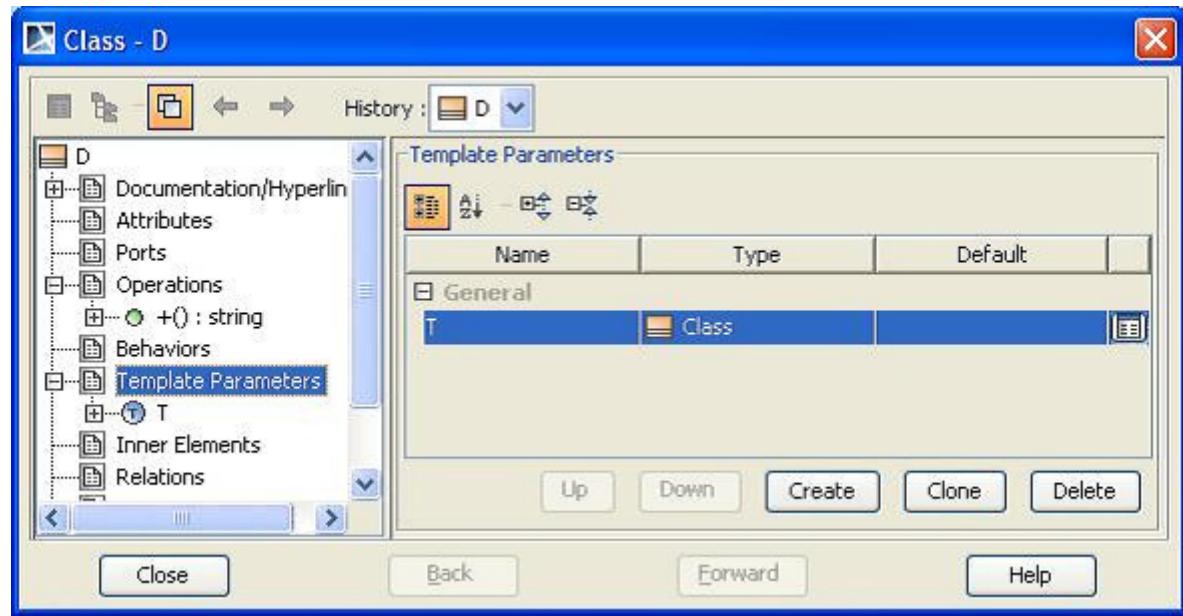
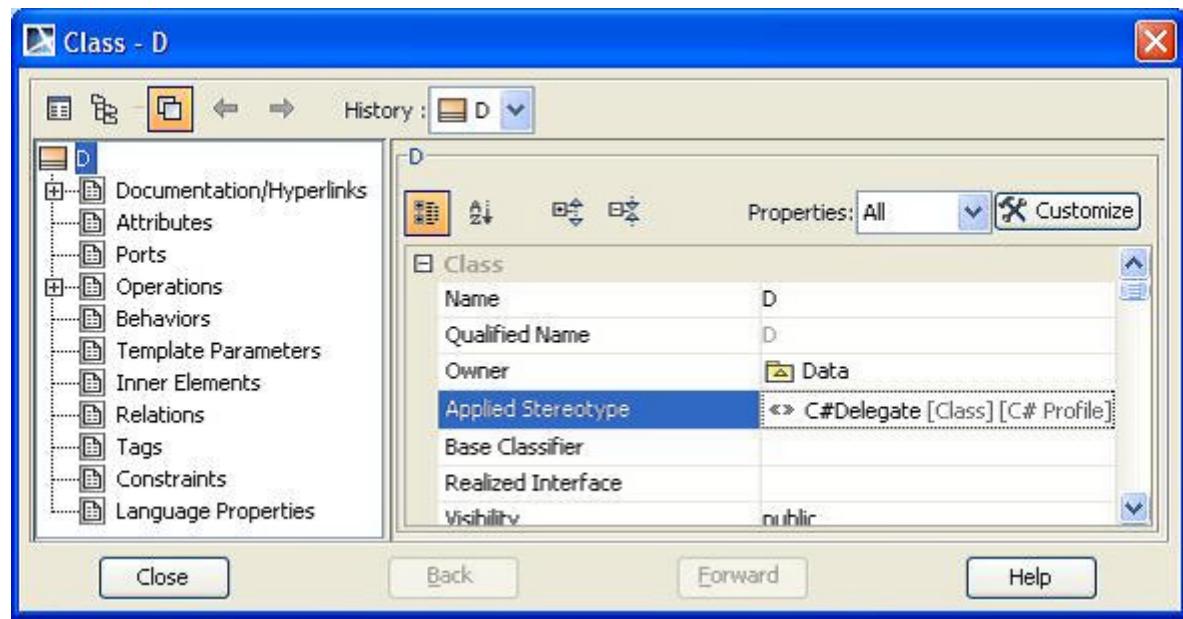
Generic Delegate

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

```
delegate string D<T>();
```



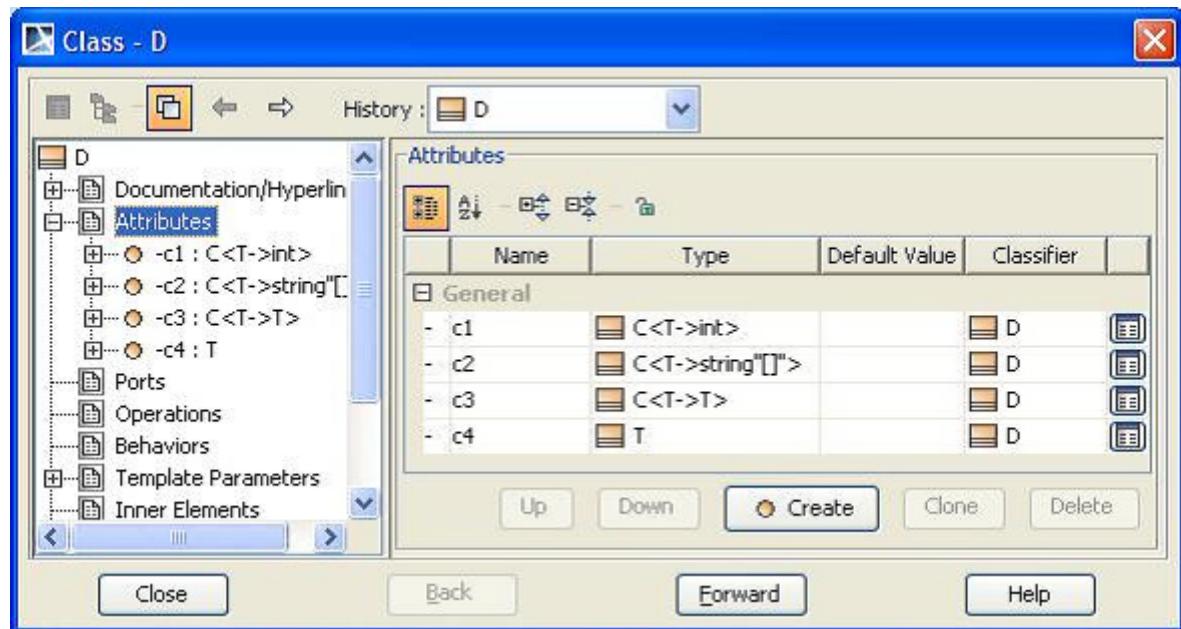


Generic Attribute

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.

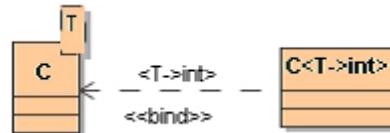
Code	MD-UML
<pre>class C<T> { } class D<T> { private C<int> c1; private C<string[]> c2; private C<T> c3; private T c4; }</pre>	<p>The MD-UML diagram illustrates the generic binding and template parameters for the classes and their attributes:</p> <ul style="list-style-type: none"> Class C<T>: Represented by a box labeled "C". It has three outgoing associations: <ul style="list-style-type: none"> An association to a box labeled "C<T>int" with multiplicity <T->int> and binding <<bind>>. An association to a box labeled "C<T>string[]>" with multiplicity <T->string["[]"]> and binding <<bind>>. An association to a box labeled "C<T>T" with multiplicity <T->T> and binding <<bind>>. Class D<T>: Represented by a box labeled "D". It has one outgoing association to the "C" box, indicating it uses the generic binding for its attributes. Attributes: <ul style="list-style-type: none"> c1: Associated with the "C<T>int" box. c2: Associated with the "C<T>string[]>" box. c3: Associated with the "C<T>T" box. c4: Directly associated with the "T" box.



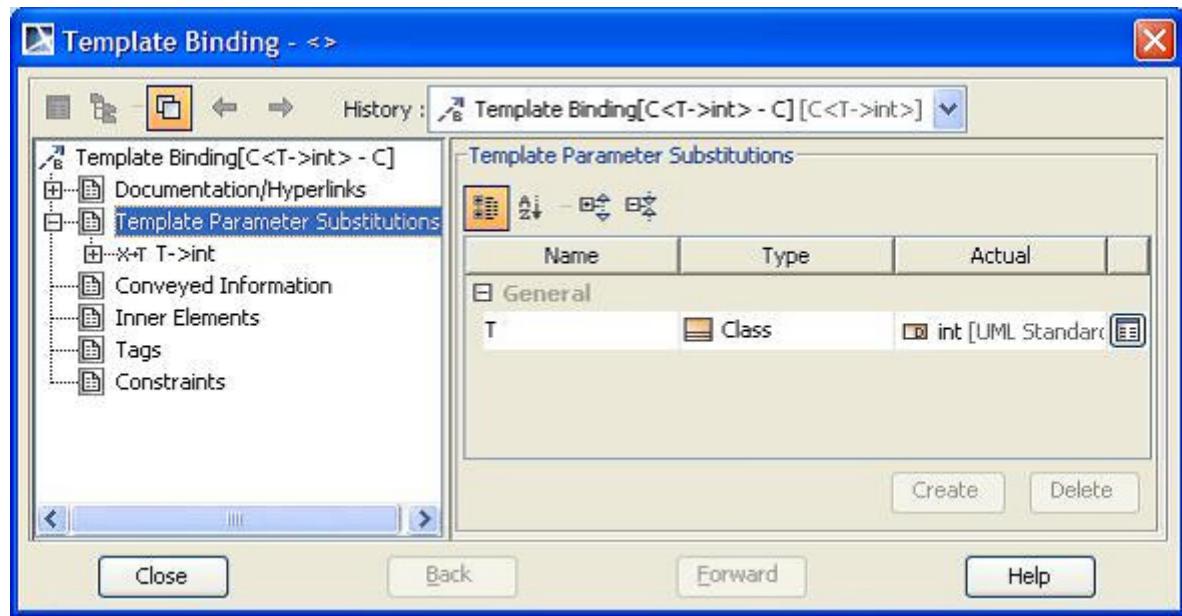
The following shows how to create each attribute.

```
..  
Private C<int> c1;
```

template binding for C<int>

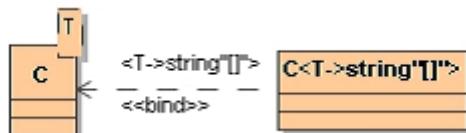


Open the specification of template binding link and add the binding type in Template Parameter Substitutions section. For binding with int datatype, select data type int in the Actual.

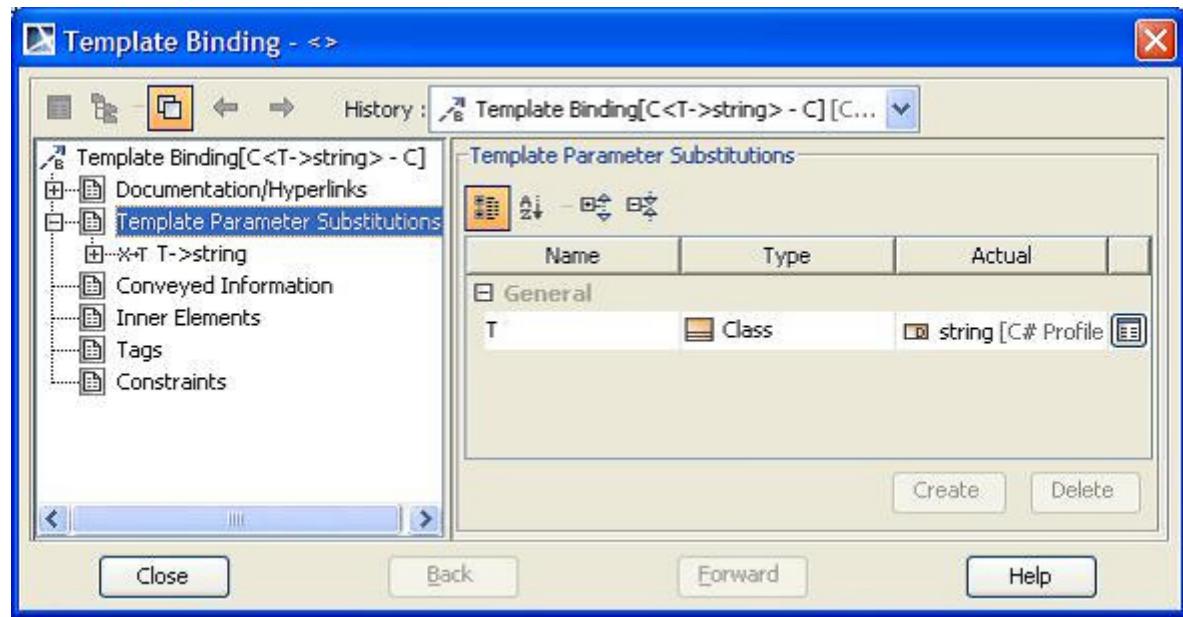


```
..  
private C<string[]> c2;
```

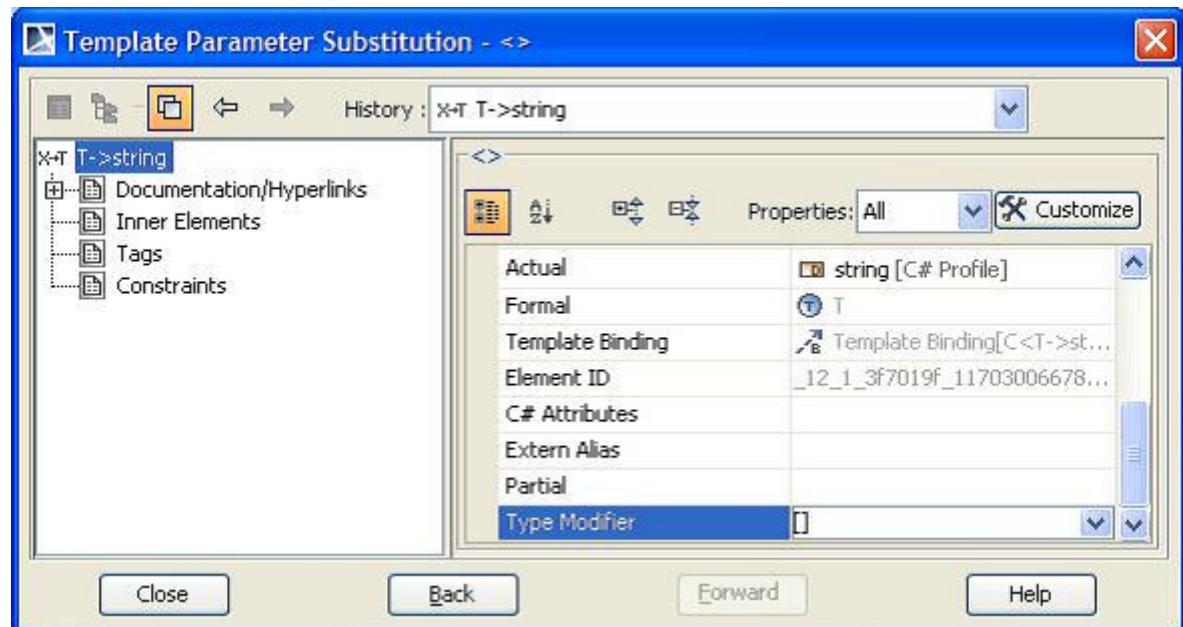
Template binding for $C<\text{string}[]>$



Create binding type as usual.

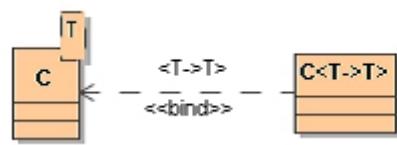


To add type modifier [] to string, open the specification of template parameter substitution, and then add [] to Type Modifier property.

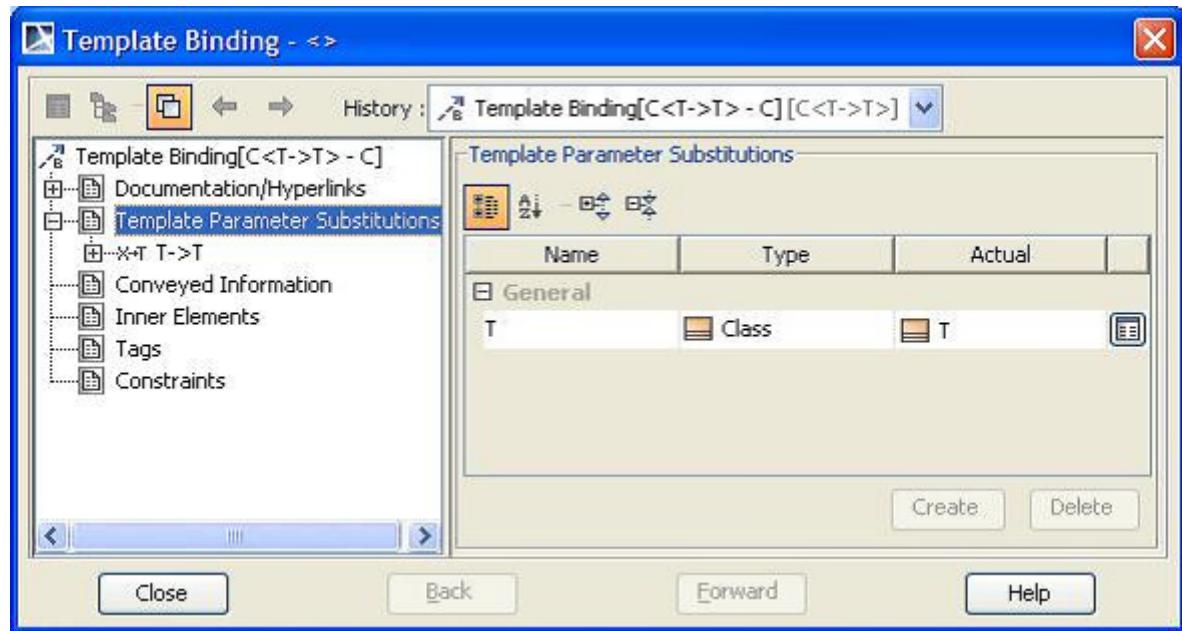


```
..  
private C<T> c3;
```

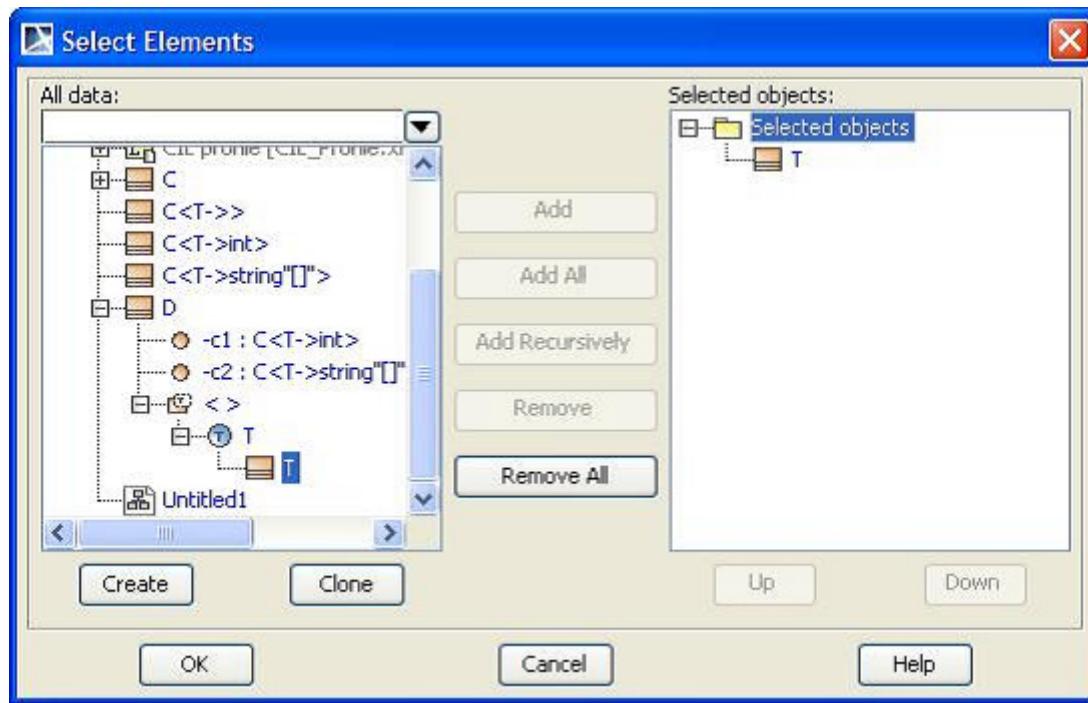
Template binding for C<T>



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 $D<T>$.



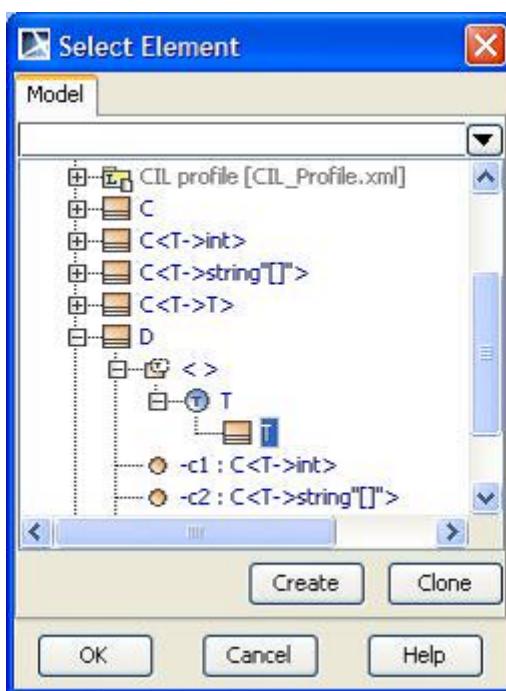
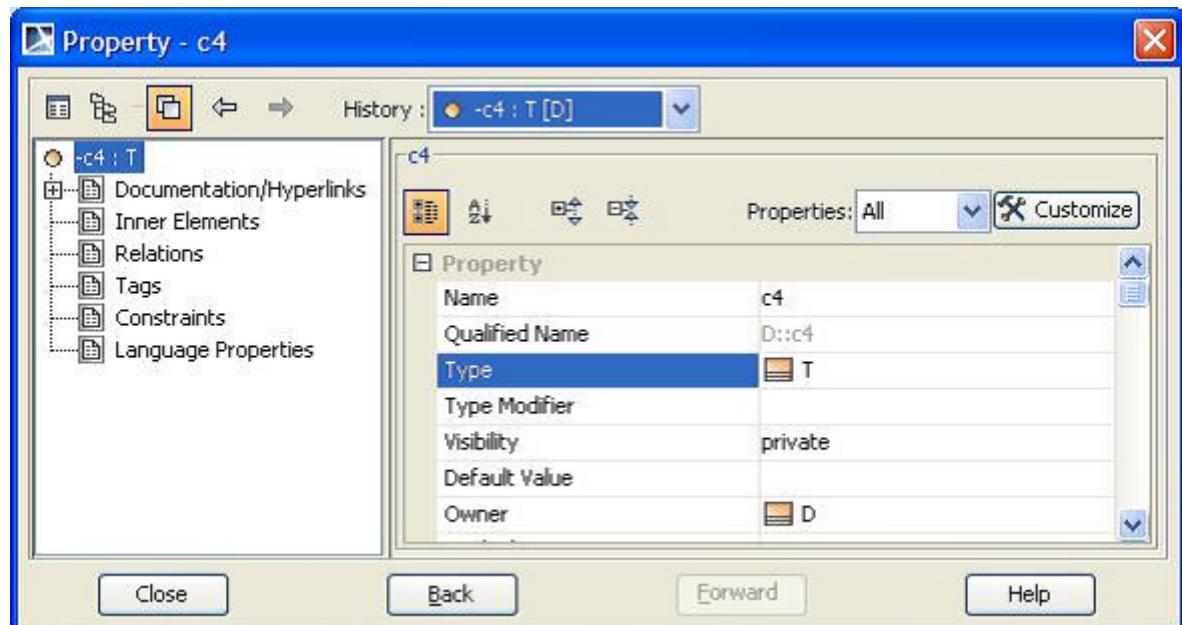
The following panel shows the correct type for creating template binding type T .

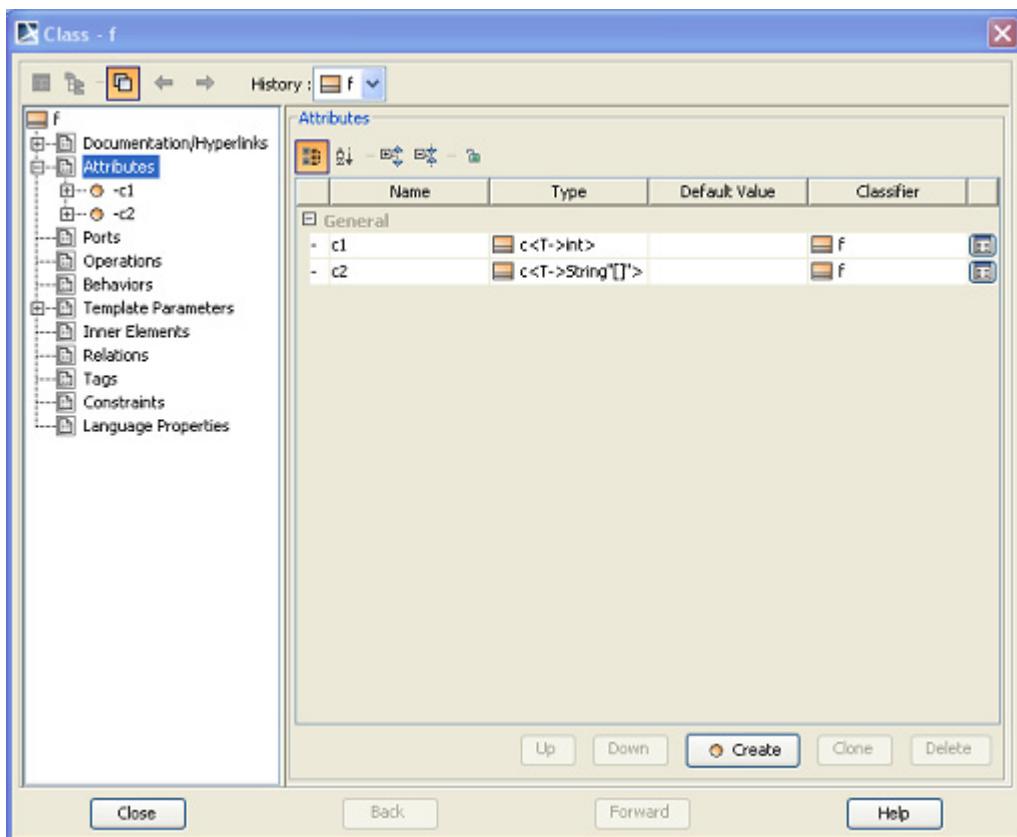


```
..  
private T c4;
```

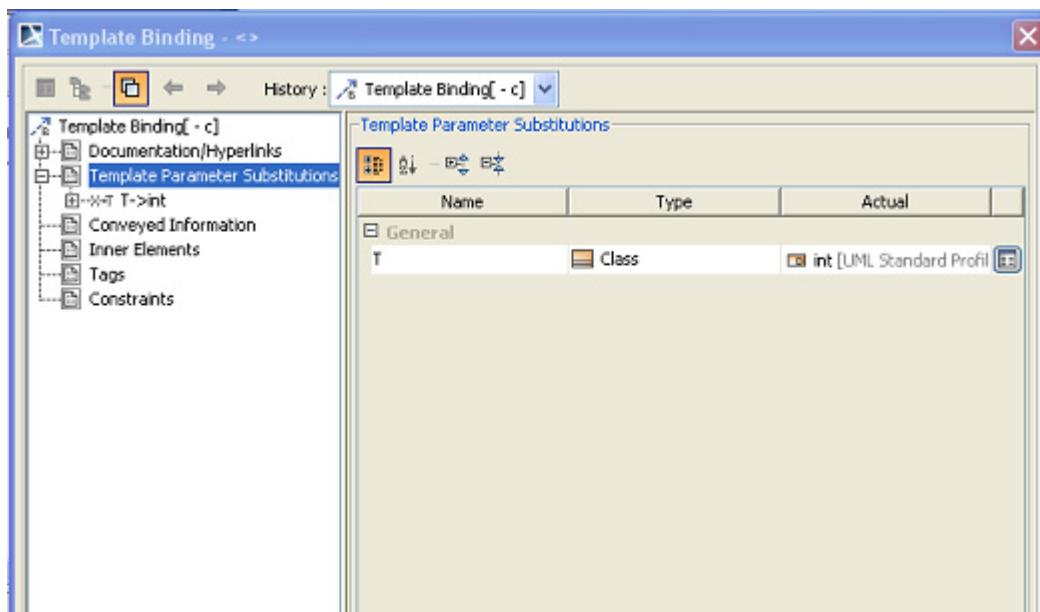
The type of attribute, c4 is not a template binding class. Its type is a template parameter of the owner class D<T>

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

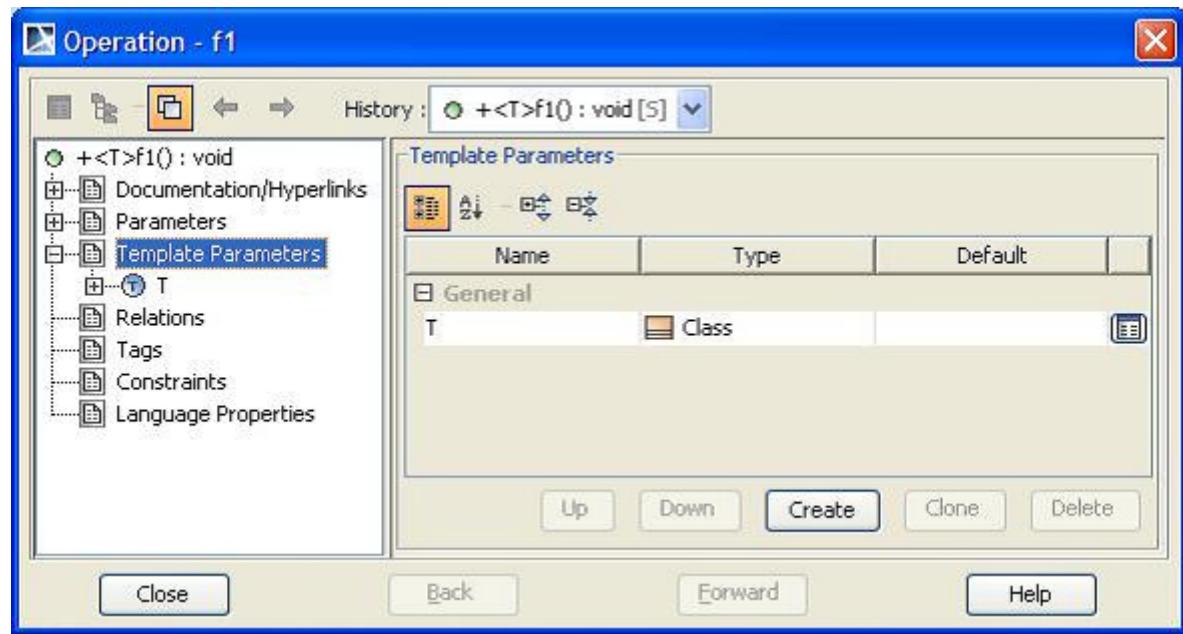


Generic Operation

Generic can be applied to operation such as operation name, return type and operation parameter.

Code	MD-UML
<pre>public class S<T> { public void f1<T>() { }</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.

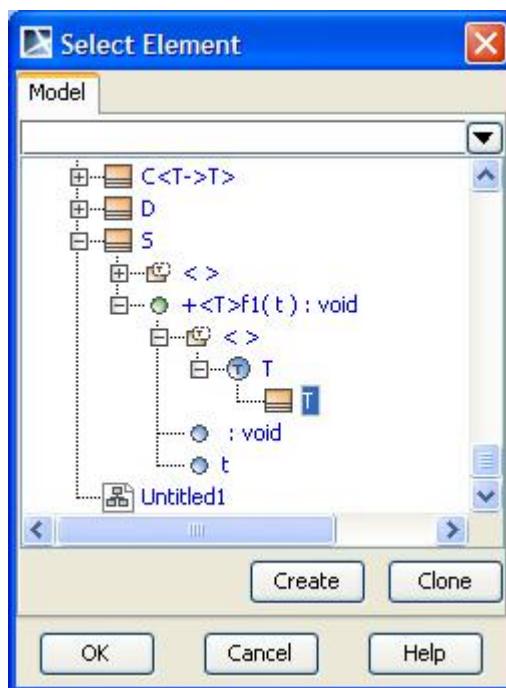
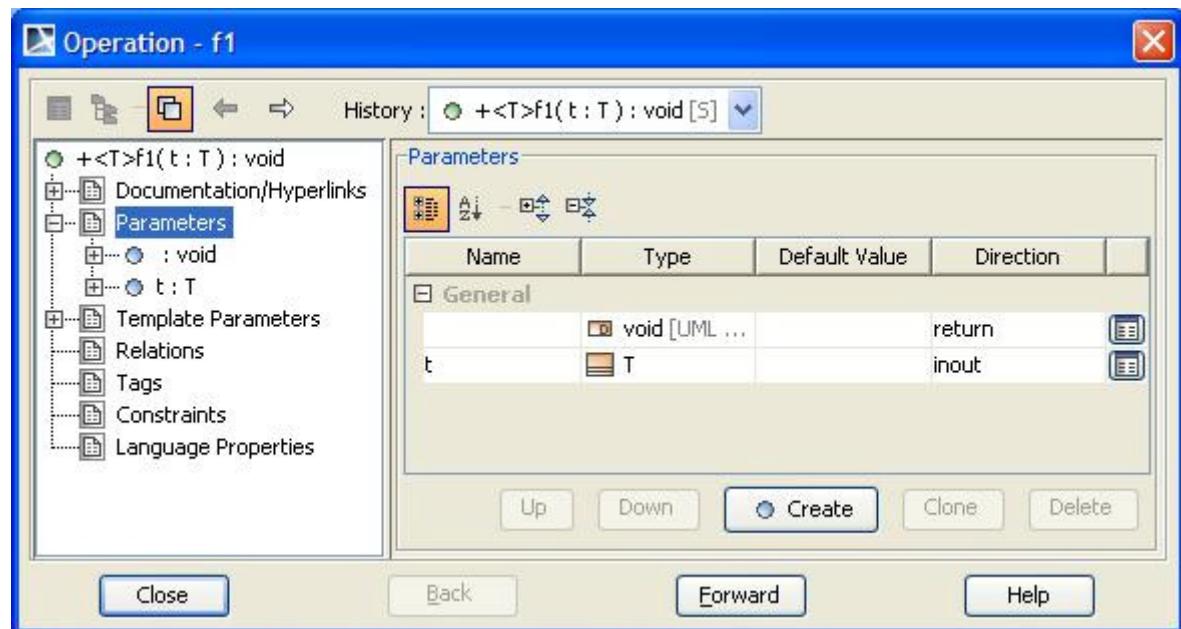
Important: We have to create or select the correct binding types.

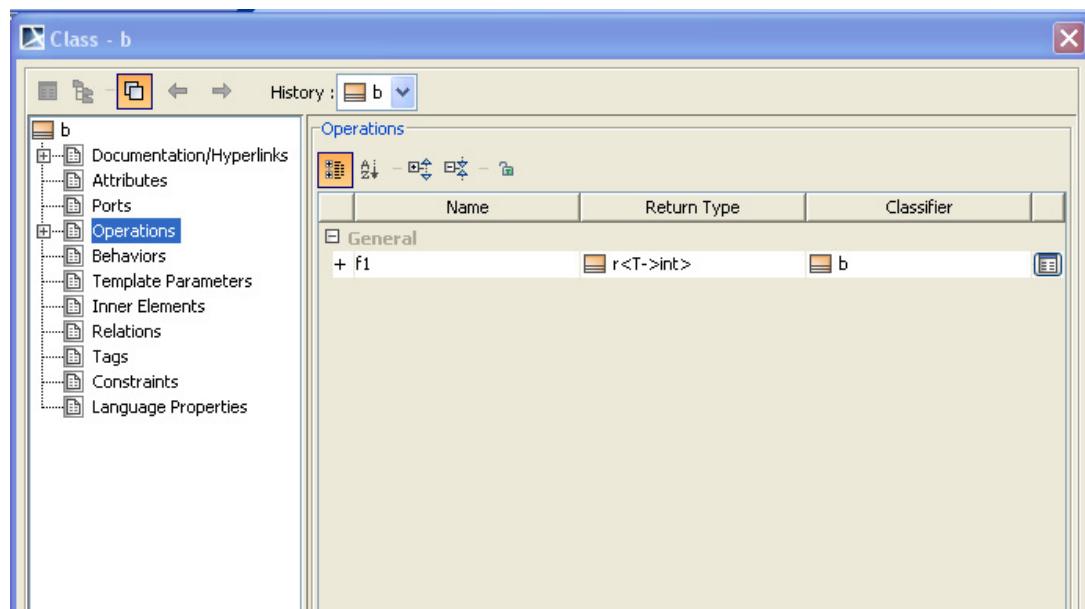
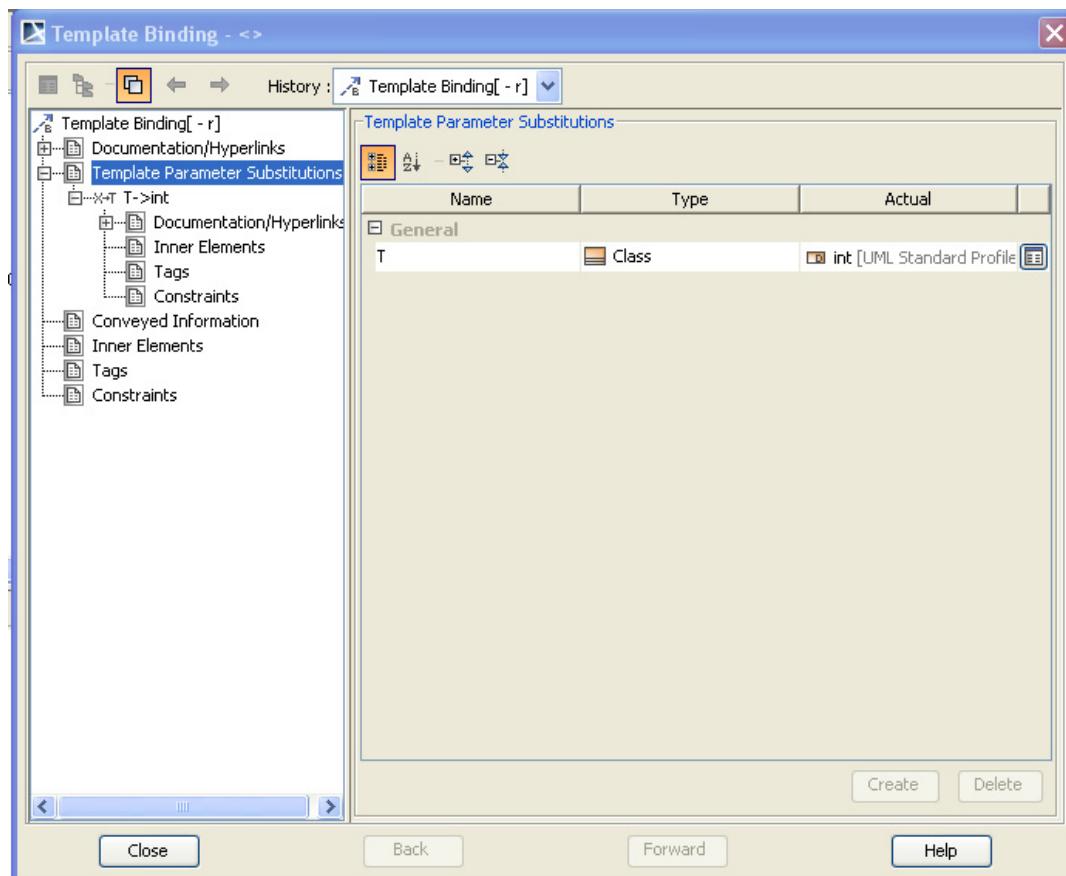
In the example code, the type of method parameter t in the first method, public void f1<T> (T t), must be the template parameter of the owner method, f1<T>, not the template parameter of the owner class, S<T>.

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>

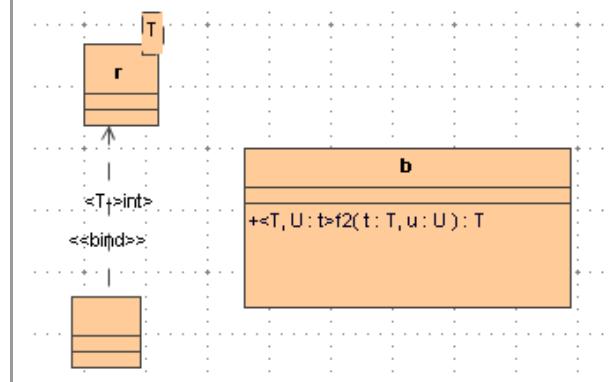
Code	MD-UML
<pre>public class S<T> { public void f1<T> (T t) { } public void f1<U, V> (T t, U u) { } }</pre>	<p>The MD-UML diagram shows a class named S with a dashed dependency arrow pointing to a type T. The class has two methods: <code>+<T>f1(t:T): void</code> and <code>+<U,V>f1(t:T,u:U): void</code>.</p>

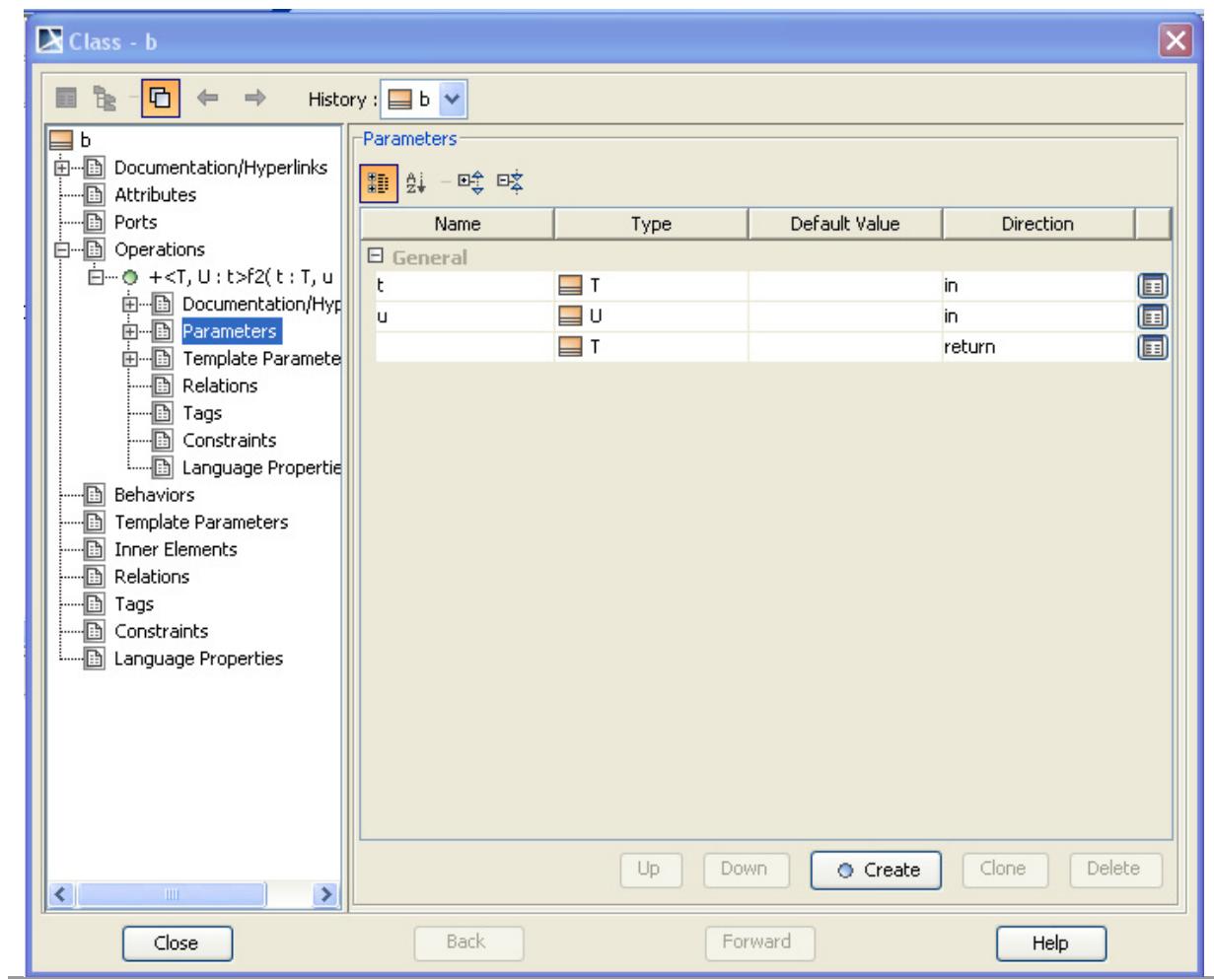
The following panels show the selection of method parameter t of the first method.





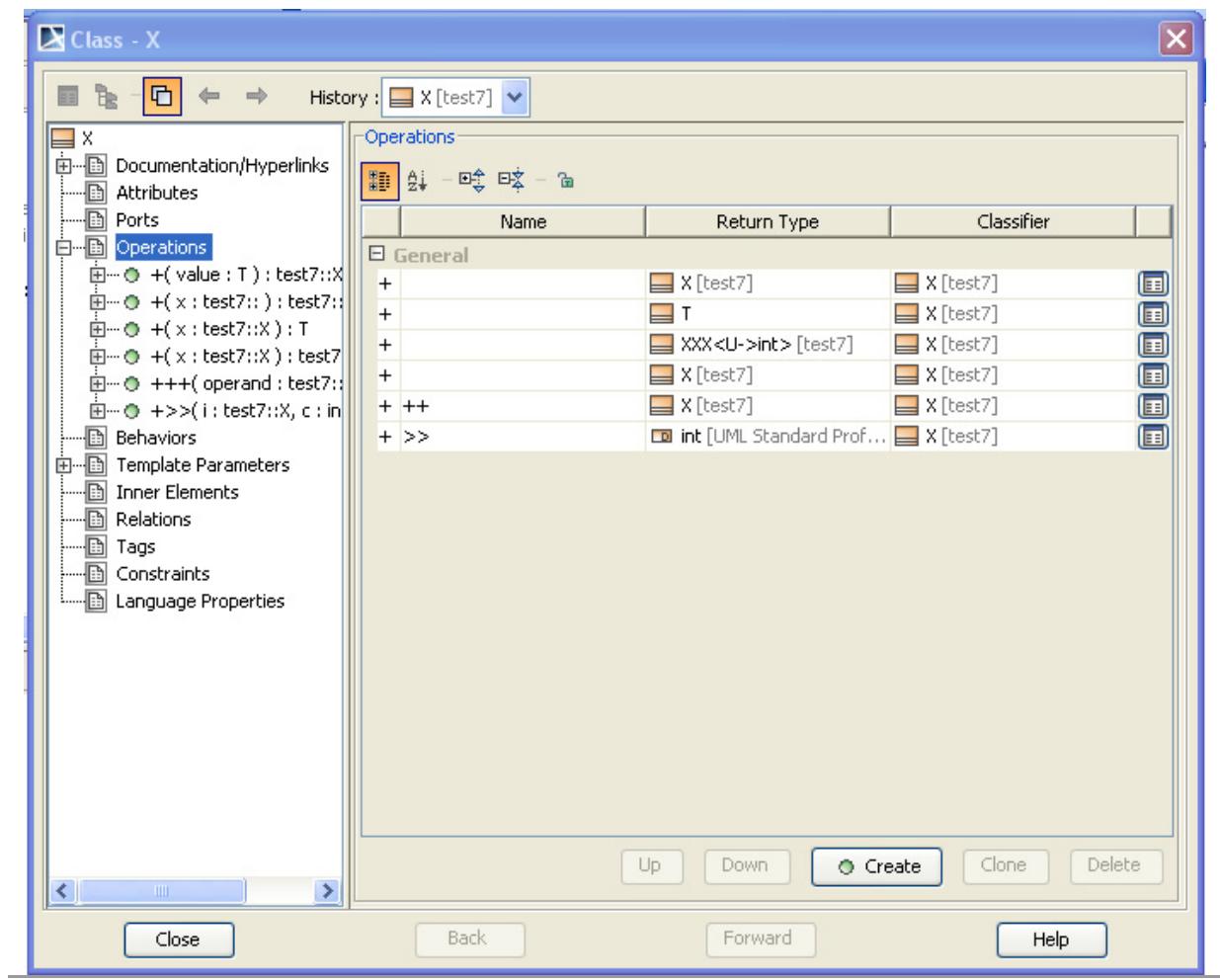
```
class b
{
    public T f2<T, U>(T t, U u)
        where U : T { return t; }
}
```





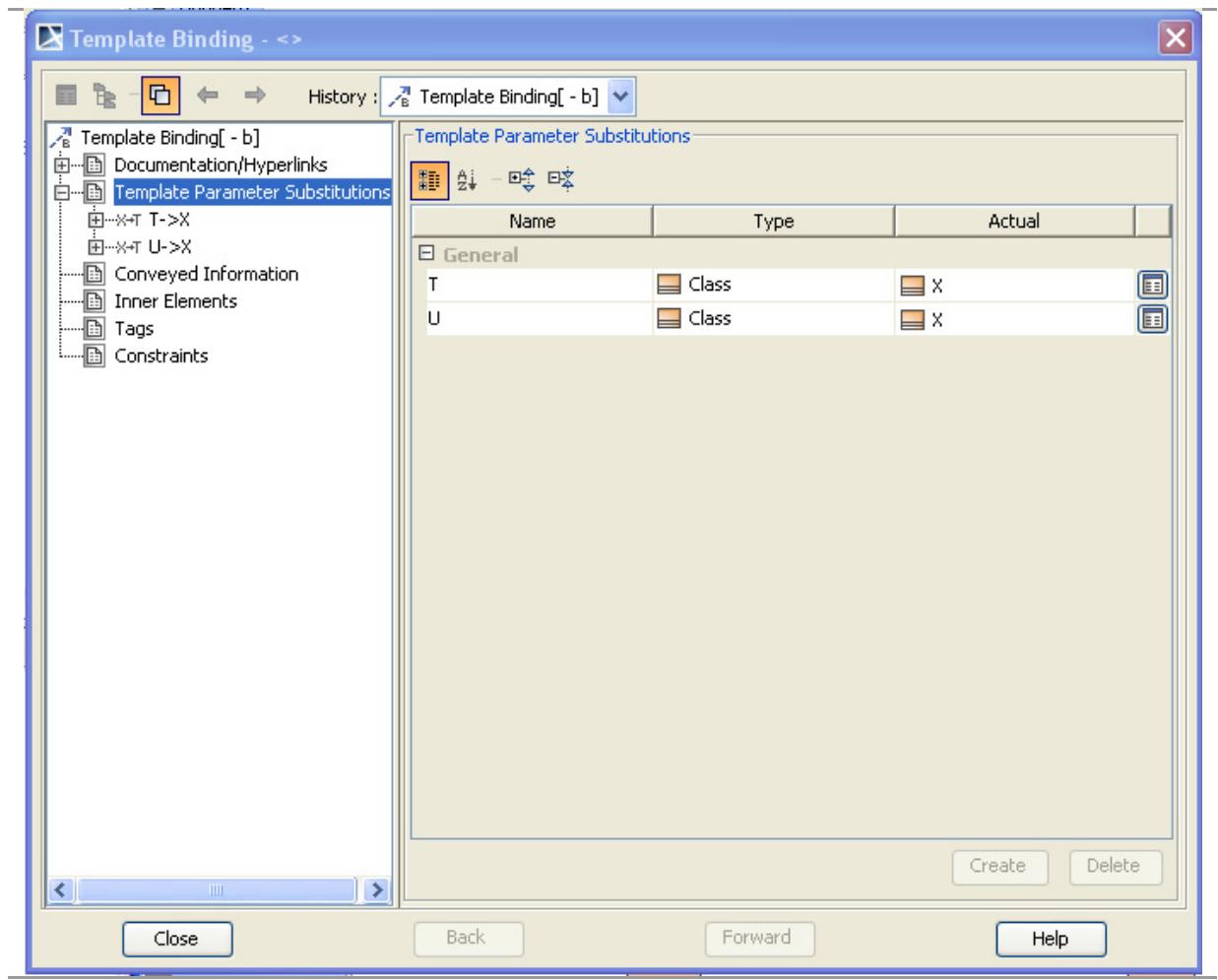
Generic Overloading

Code	MD-UML
<pre>class X<T> { public static explicit operator X<T>(T value) { return null; } public static implicit operator T(X<T> x) { return x.item; } public static explicit operator XXX<int>(X<T> x) { return null; } public static explicit operator X<T>(XXX<int> x) { return null; } public static X<T> operator ++(X<T> operand) { return null; } public static int operator >>(X<T> i, int c) { return c; } }</pre>	<p>The MD-UML diagram illustrates the generic class X<T>. It contains several operator overloads:</p> <ul style="list-style-type: none"> <code><<C#Operator>>+(value : T) : X</code> <code><<C#Operator>>+(x : X) : T</code> <code><<C#Operator>>+(x : X) :</code> <code><<C#Operator>>+(x :) : X</code> <code><<C#Operator>>+++(operand : X) : X</code> <code><<C#Operator>>+>>(i : X, c : int) : int</code>



Generic Parent Class

Code	MD-UML
<pre>class b<T, U> {} class b1<X> : b<X[], X[,]> {}</pre>	<p>The MD-UML diagram illustrates the generic parent class 'b' and its derived class 'b1'. The class 'b' is represented by a rectangle divided into three horizontal sections. The top section contains two orange boxes labeled 'T' and 'U'. The middle section contains the letter 'b'. The bottom section is empty. An association arrow points from 'b' to 'b1'. The association is labeled with the note '<T->X[], U->X[,]>' and the binding indicator '<<bind>>'. The class 'b1' is also divided into three horizontal sections. The top section contains an orange box labeled 'X'. The middle section contains the letter 'b1'. The bottom section is empty.</p>



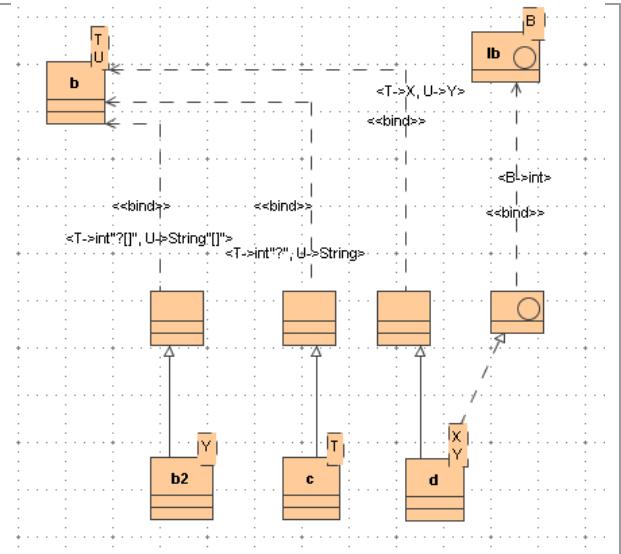
Code

MD-UML

```

class b<T, U>
{}
interface Ib<B>
{}
class b2<Y> : b<int?[], string[]>
{}
class c<T> : b<int?, string>
{}
class d<X, Y> : b<X, Y>, Ib<int>
{}

```



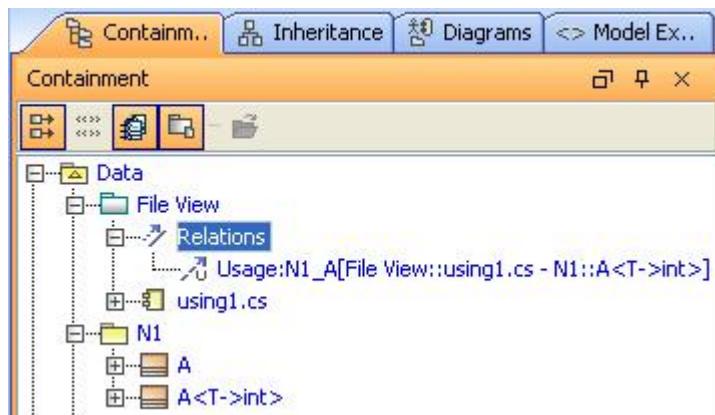
Code	MD-UML
<pre> class b<T, U> {} interface Ib {} class g<U> : b<b<int, object>, U>, Ib<string> {} </pre>	<p>This UML Class Diagram shows the modified code where b is now a base class for g and b2. g has a generic parameter U. b2 remains a derived class of b with a single generic parameter Y. The interface Ib is still present with its generic parameter B. Annotations like <<bind>> and <B->String> are included.</p>

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<int>; namespace N1 { public class A<T> {}} class A { N1_A a; }</pre>	<p>The MD-UML diagram illustrates the mapping of the C# code. It shows a component box labeled "using1.cs" with a "using" directive "N1_A" pointing to a template binding "A<T>int". Three dashed arrows originate from the component and point to three separate instances of class "A". The first instance has a type constraint "T" above it. The second instance is a simple "A". The third instance is an "A" with a dependency "-a : A<T>int" below it.</p>



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*

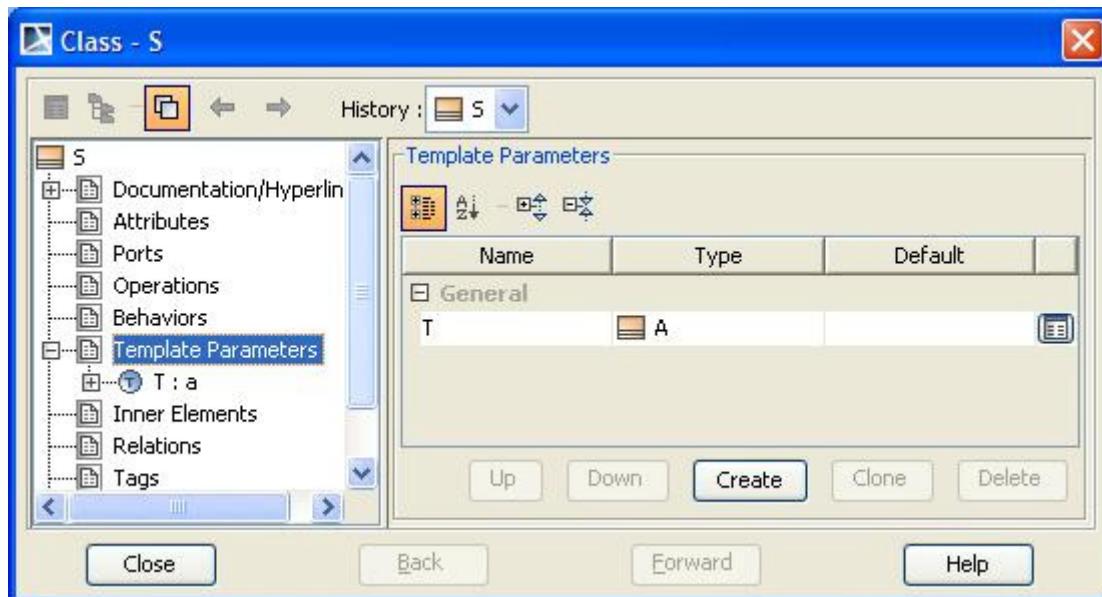
constructor-constraint:

 new ()

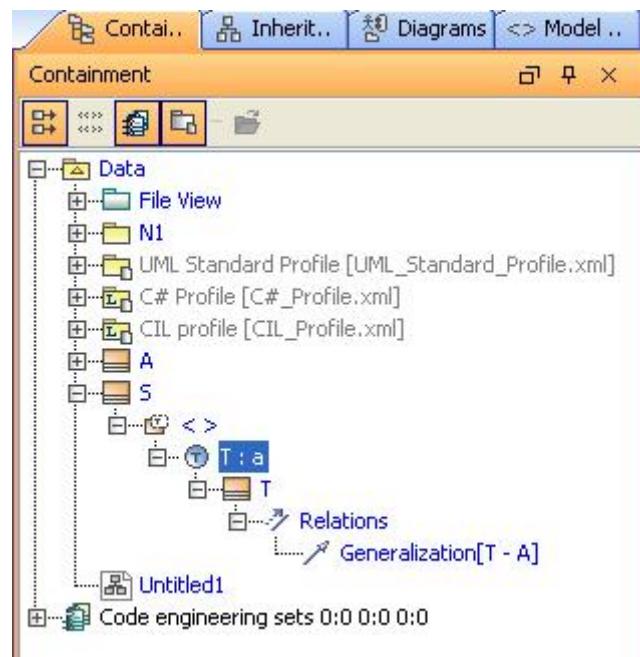
Code	MD-UML
<pre>public class A { } interface IA {} public class S<T> where T: A, IA{}</pre>	

Code	MD-UML

To create the first template constraint for template parameter T, we select the type of template parameter T as class A.

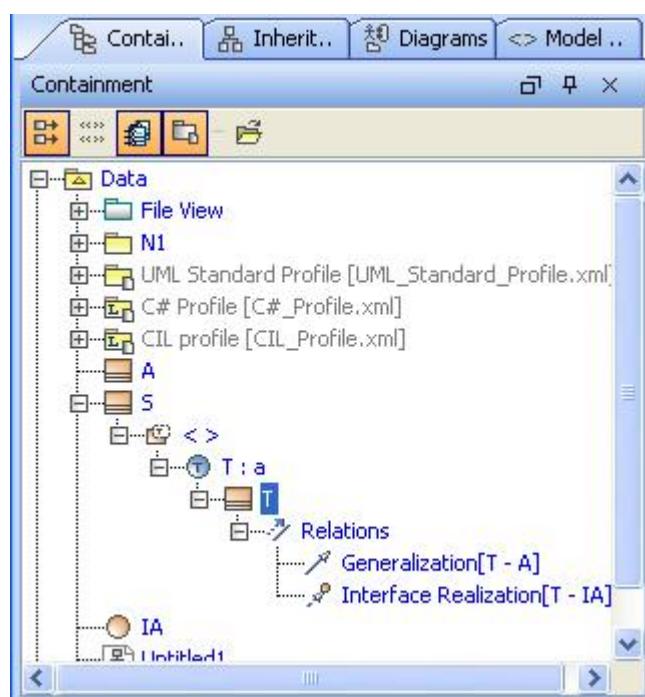
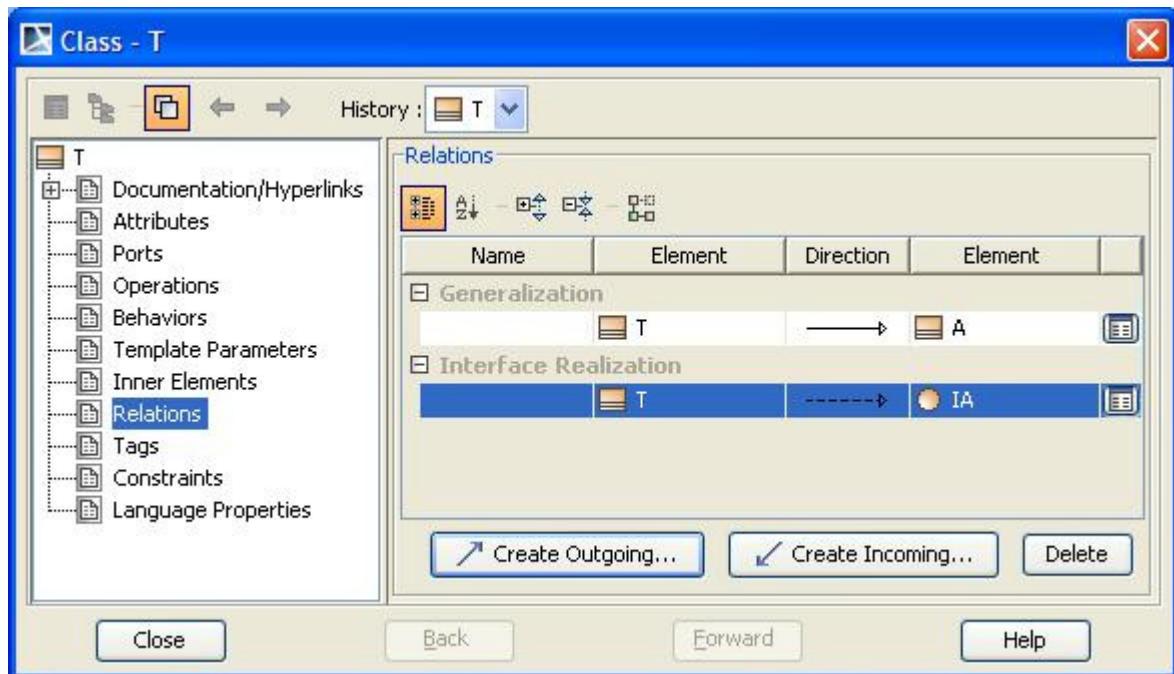


The constraint A is shown as generalization relation of T.

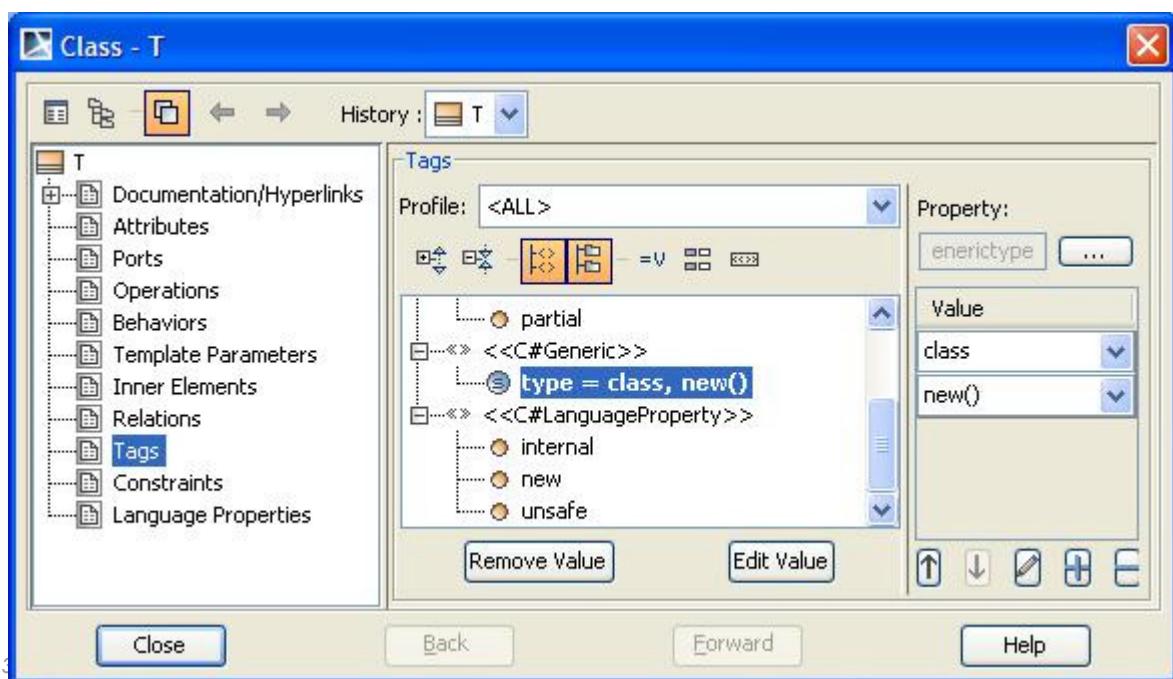
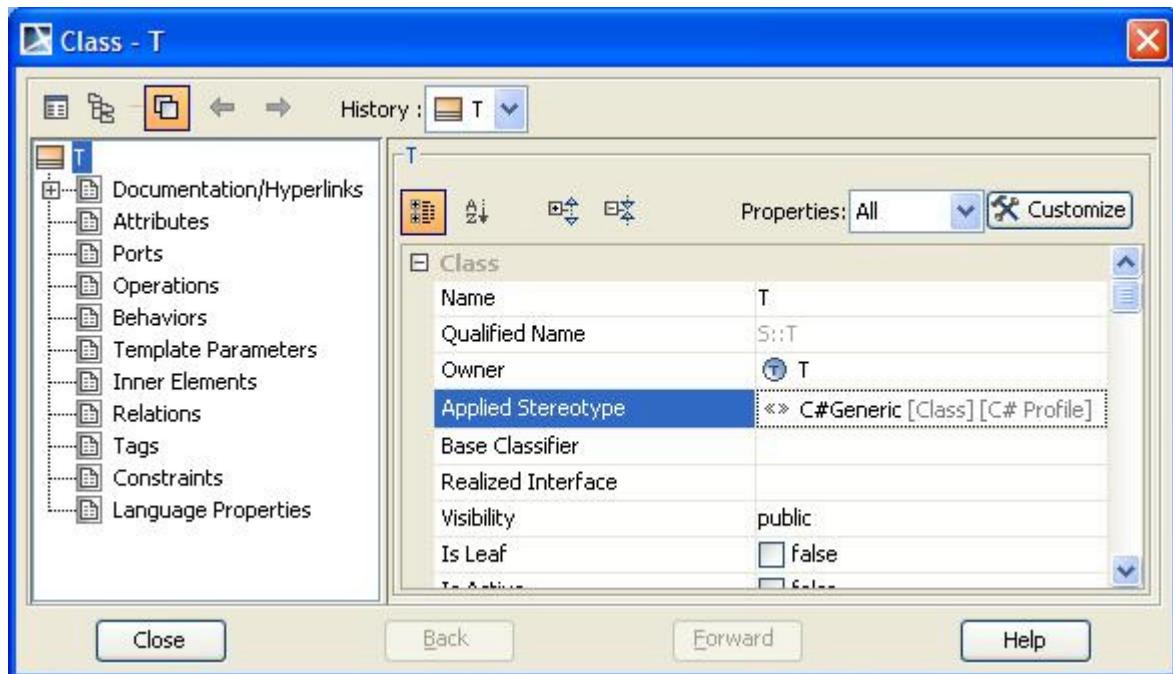


Code	MD-UML
------	--------

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
```

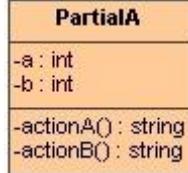
Each part of a partial type declaration must include a `partial` modifier and must be declared in the same namespace as the other parts. The `partial` modifier indicates that additional parts of the type declaration

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.

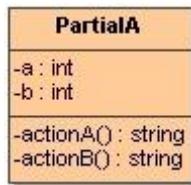
Nested types may be declared in multiple parts by using the `partial` modifier. Typically, the containing type is declared using `partial` as well, and each part of the nested type is declared in a different part of the containing type.

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>	 <pre>PartialA -a : int -b : int -actionA() : string -actionB() : string</pre> <p>All child elements will be merged into one Class element.</p>

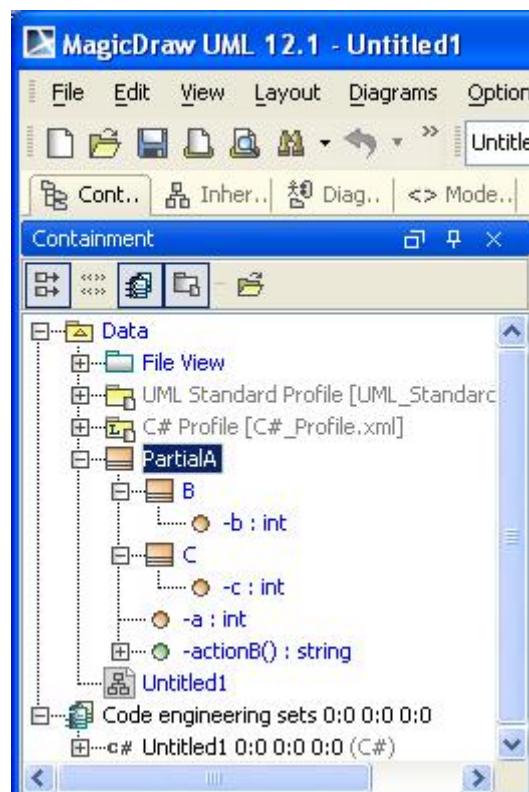
```
//Case #2
//The partial class is written into
//separate class file.
//PartialA1.cs
public partial class PartialA{
    int a;
    string actionA()
{
}
}

//PartialA2.cs
public partial class PartialA{
    int b;
    string actionB()
{
}
}
```

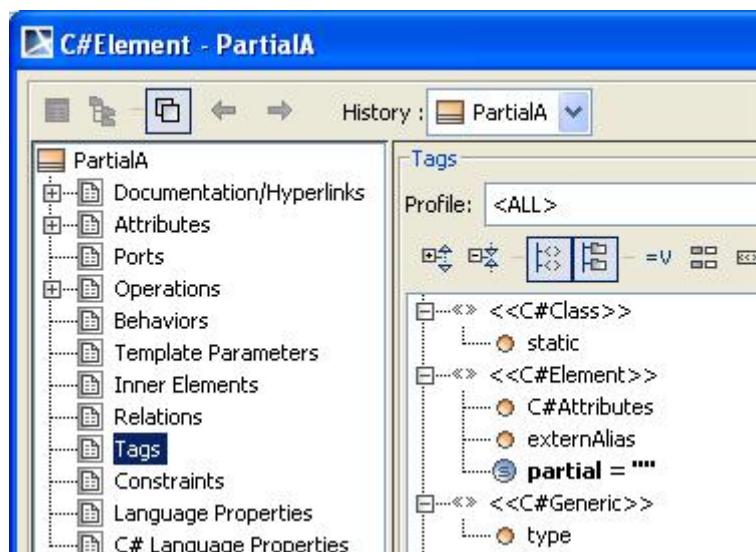


All child elements will be merged into one Class element, the same as both classes are written into one class file.

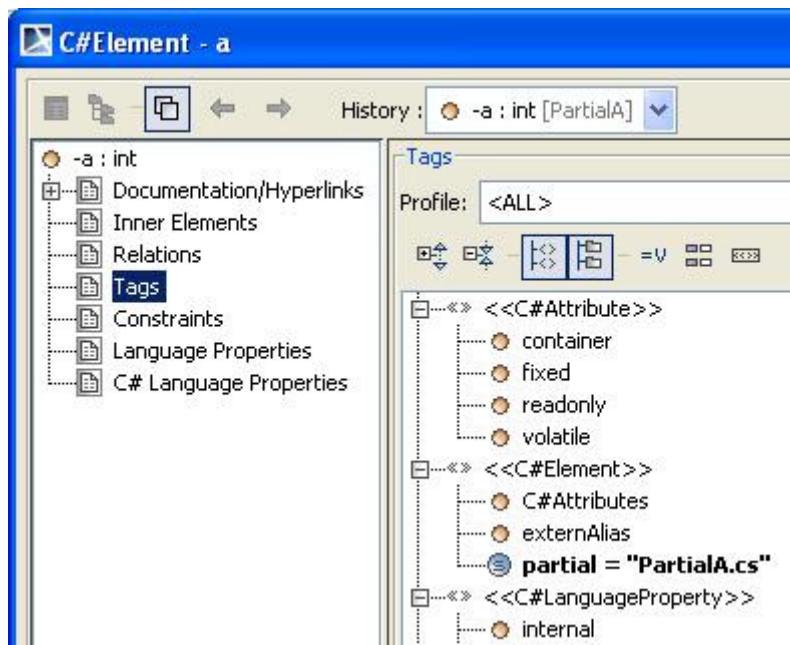
```
//Case #3
//The partial class with inner class
public partial class PartialA{
    public class B
    {
        int b;
    }
}
public partial class PartialA{
    int a;
    string actionB()
    {
    }
    public class C
    {
        int c;
    }
}
```



All inner classes have to be located as inner class of the parent class (partial class).



The "partial" tag will have a blank tagged value for Partial Class Element.



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 ?

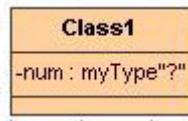
non-nullable-value-type:

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>	

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_{opt} *accessor-modifier_{opt}* **get** *accessor-body*

set-accessor-declaration:

attributes_{opt} *accessor-modifier_{opt}* **set** *accessor-body*

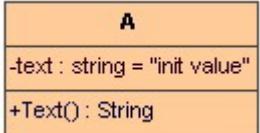
accessor-modifier:

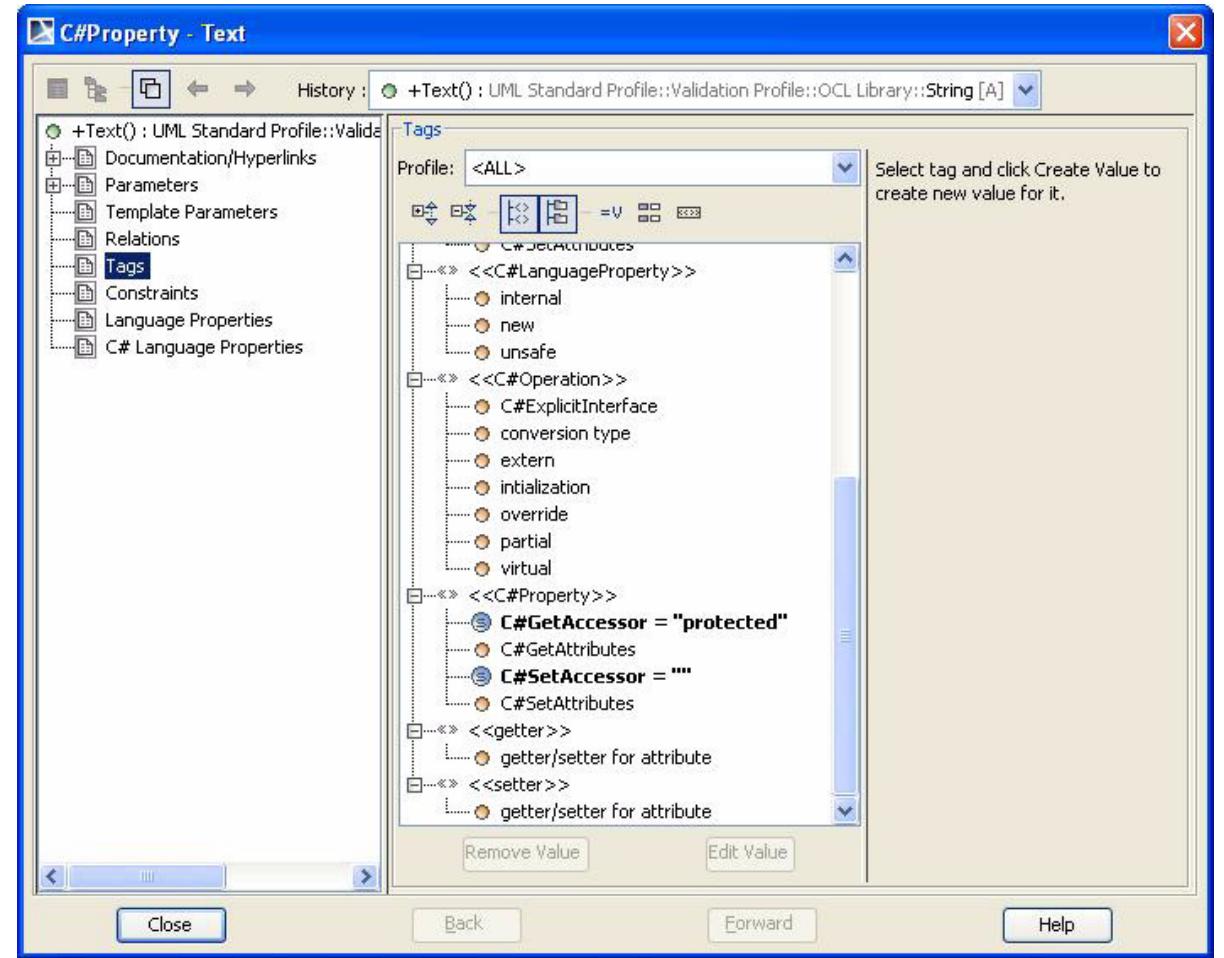
protected
internal
private
protected internal
internal protected

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>	 <pre> classDiagram class A { -text : string = "init value" +Text() : String } </pre> <p>The UML class diagram shows a class named 'A'. It has a private attribute '-text : string' initialized to 'init value'. It also has a public method '+Text() : String'.</p>



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_{opt} *class-modifiers_{opt}* *partial_{opt}* *class identifier type-parameter-list_{opt}*
class-base_{opt} *type-parameter-constraints-clauses_{opt}* *class-body ;_{opt}*

class-modifiers:

class-modifier

class-modifiers class-modifier

class-modifier:

new

public

protected

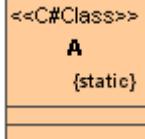
internal

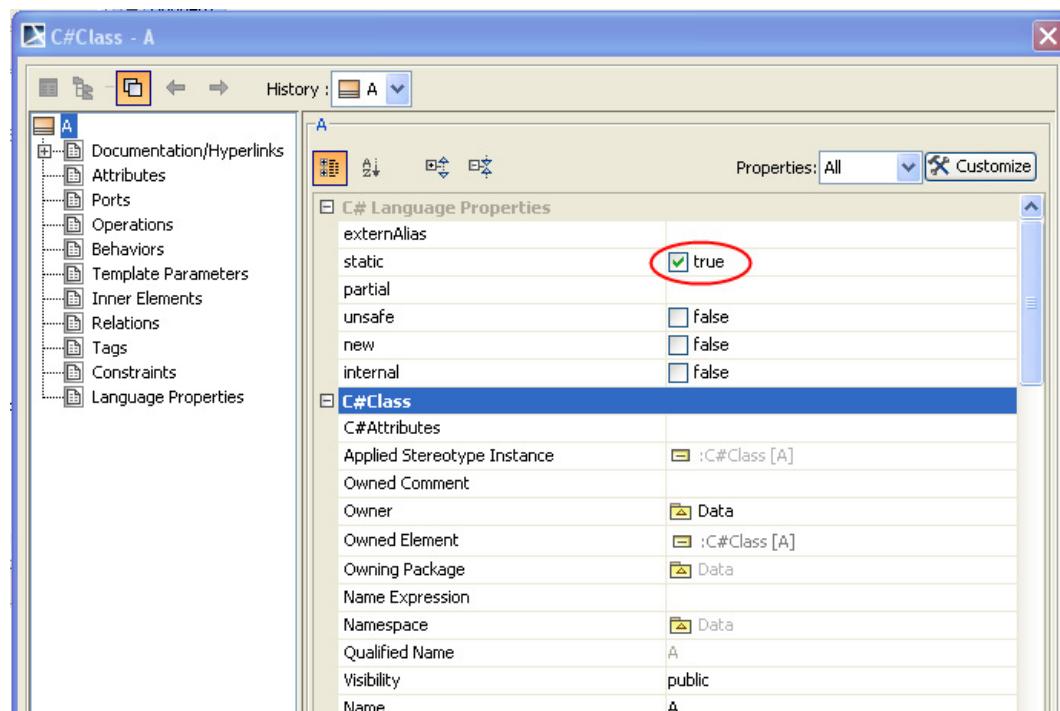
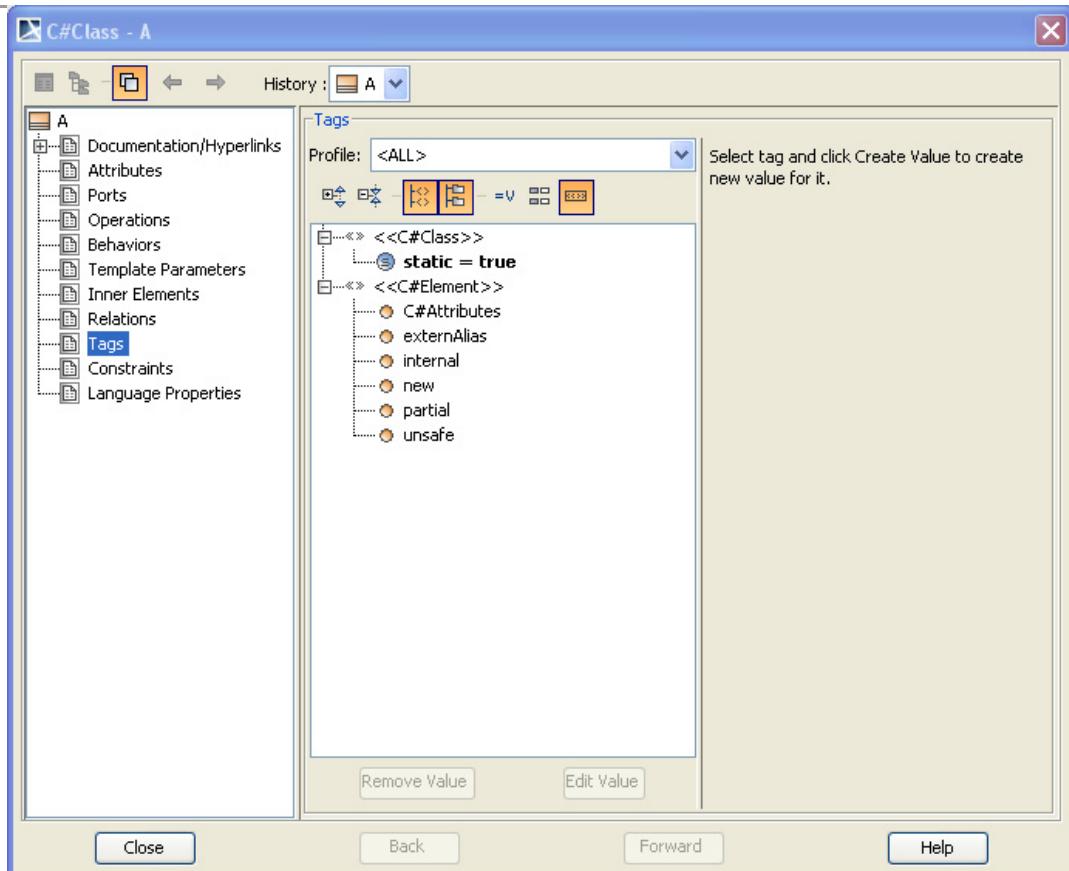
private

abstract

sealed

static

Code	MD-UML
<pre>static class A { ... }</pre>	 <p>The diagram shows a UML class named 'A' with a double-lined border, indicating it is a static class. The class has a compartment labeled '{static}' below the name.</p>



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-directives_{opt} *using-directives_{opt}* *global-attributes_{opt}*
namespace-member-declarations_{opt}

namespace-body:

{ *extern-alias-directives_{opt}* *using-directives_{opt}* *namespace-member-declarations_{opt}* }

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
```

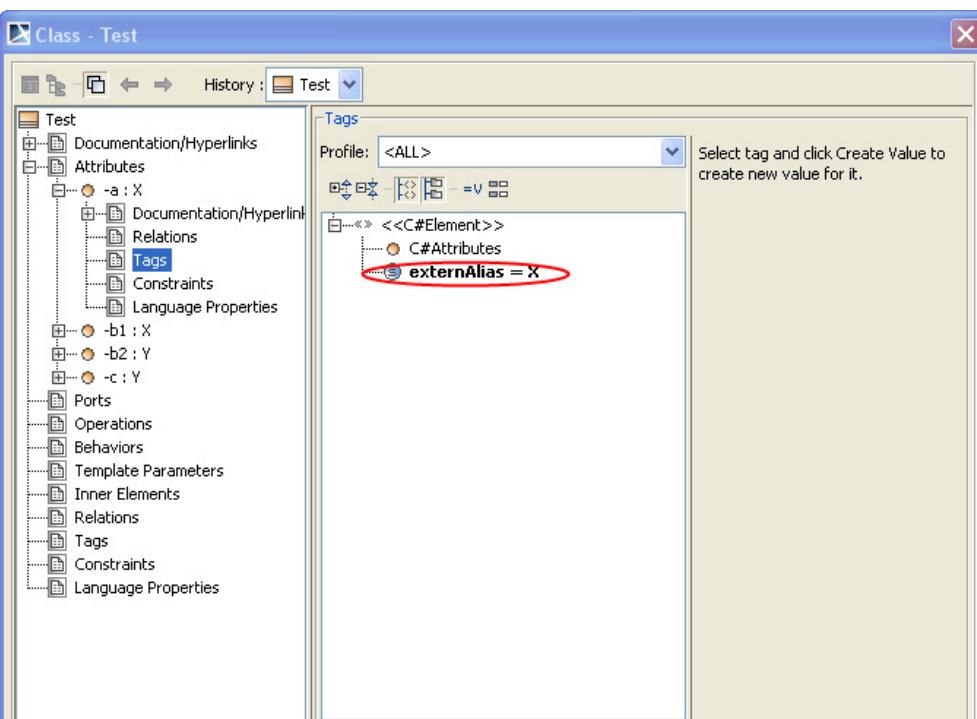
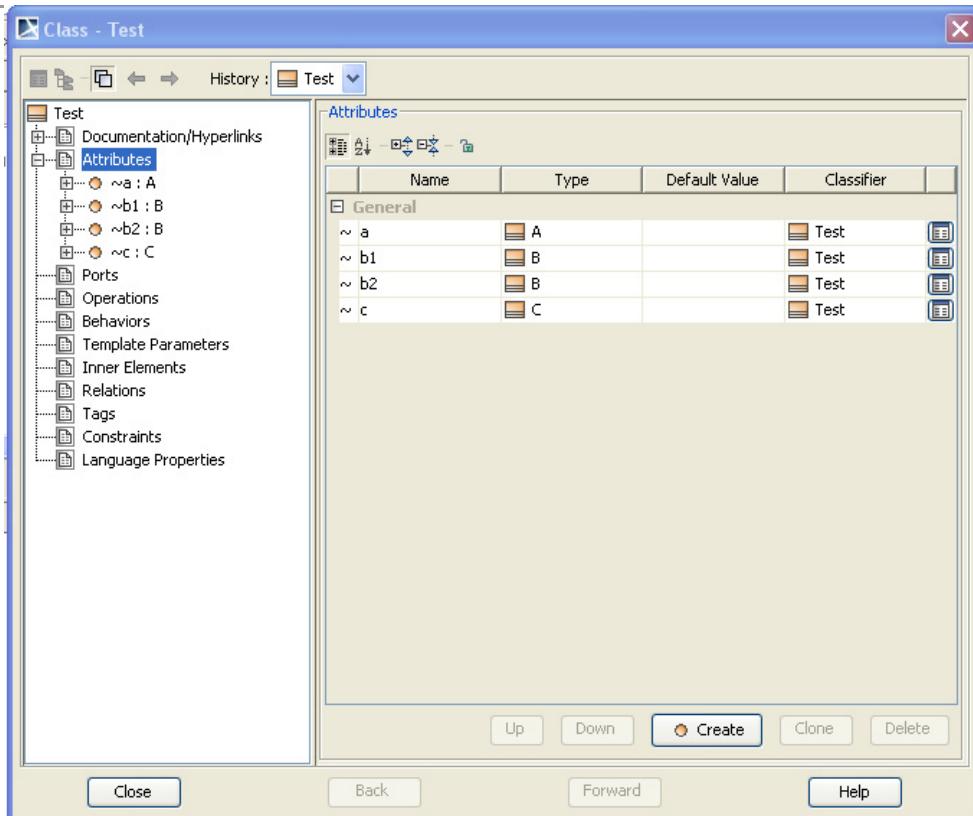
```
{
    N.A a;
    N.C c;
```

```
}
```

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: 10px;"> Test <pre><<C#Element>>-a : X{externAlias = X} <<C#Element>>-b1 : X{externAlias = X} <<C#Element>>-b2 : Y{externAlias = Y} <<C#Element>>-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:

whitespace_{opt} # *whitespace_{opt}* **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 *whitespace_{opt}* , *whitespace_{opt}* *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:

```
attributes opt fixed-size-buffer-modifiers opt 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

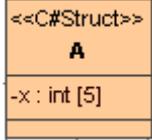
fixed-sized-buffer-declarators:

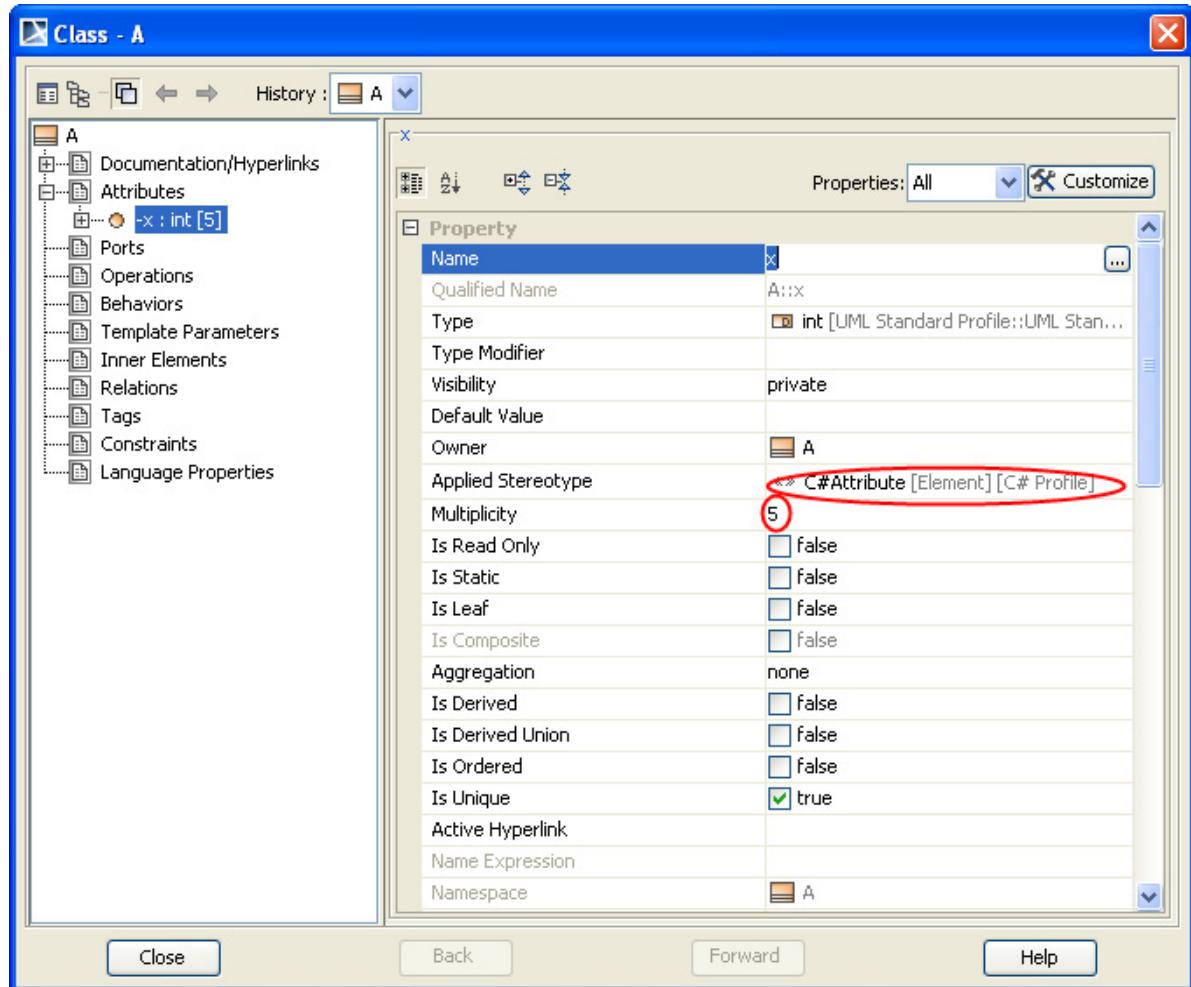
fixed-sized-buffer-declarator

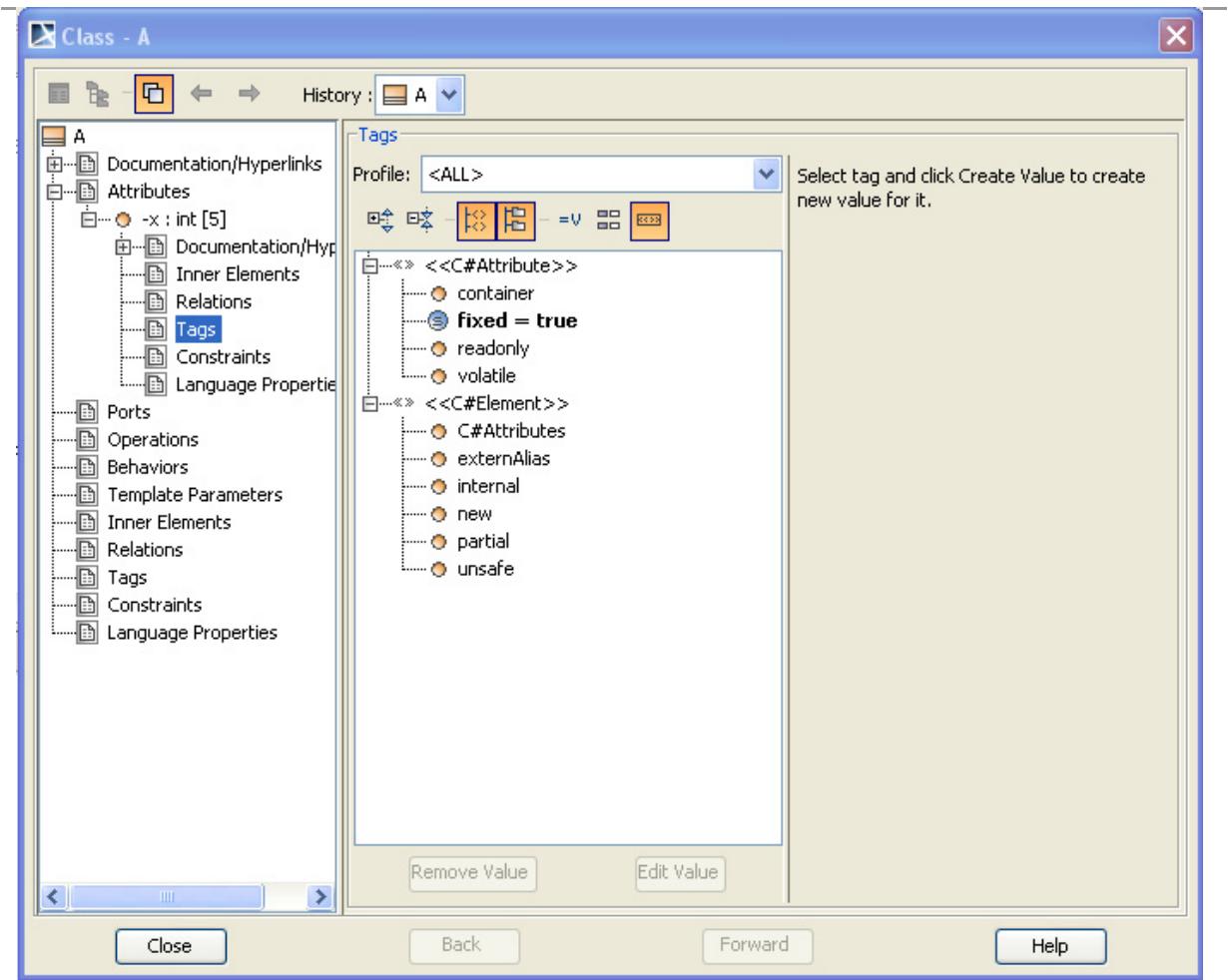
fixed-sized-buffer-declarator *fixed-sized-buffer-declarators*

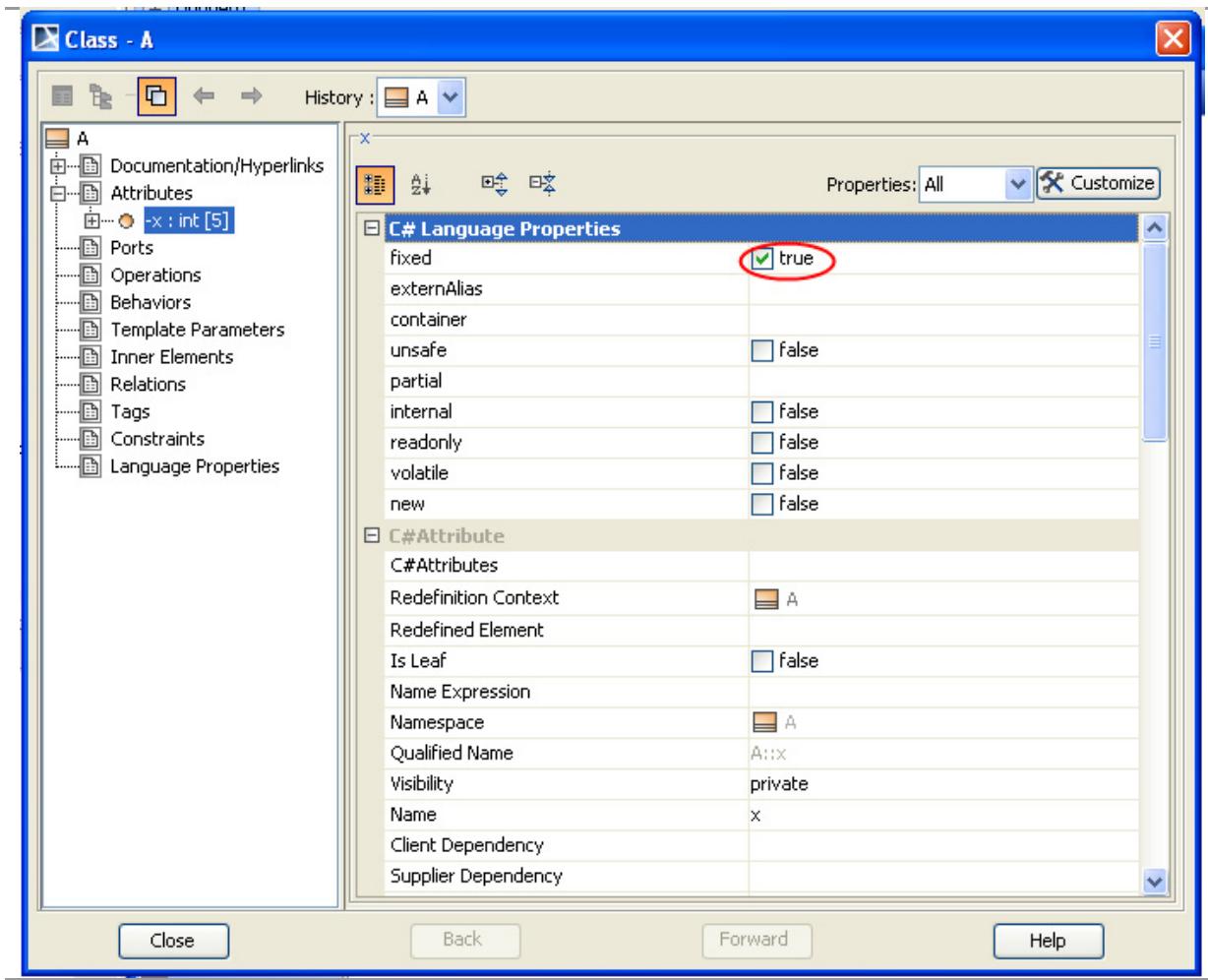
fixed-sized-buffer-declarator:

identifier [*const-expression*]

Code	MD-UML
<pre>unsafe struct A { public fixed int x[5];</pre>	 <p>The diagram shows a UML class named 'A' with a note '<<c#struct>>' '-x="" :="" [5]',="" a="" above="" attribute,="" below="" class="" compartment="" has="" in="" int="" it="" it.="" name.<="" one="" p="" shown="" the=""></c#struct>>'></p>







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 :

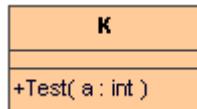
```
public static class Extensions
{
    public static intToInt32(this string s) {
        return Int32.Parse(s);
    }

    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;
    }
}
```

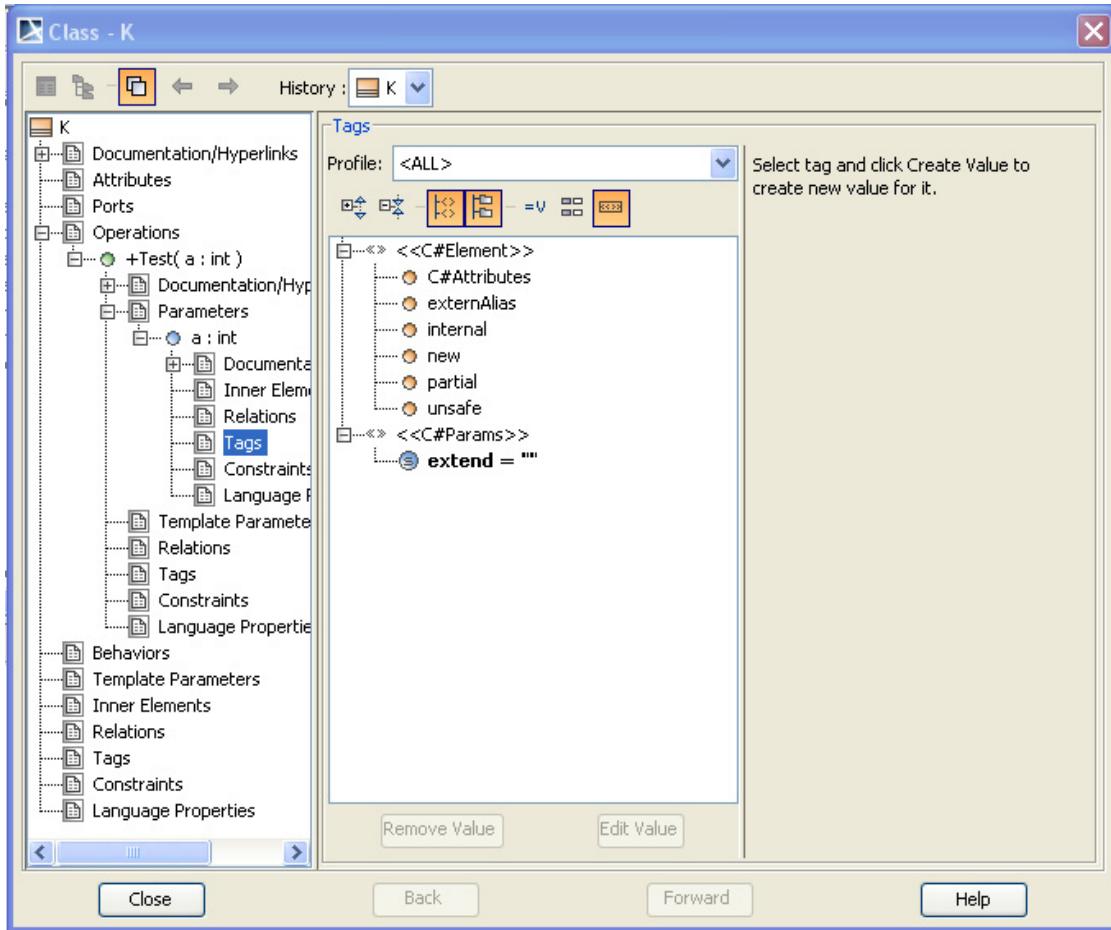
It becomes possible to invoke the extension methods in the static class `Extensions` using instance method syntax:

```
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>	 <pre>K +Test(a:int)</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

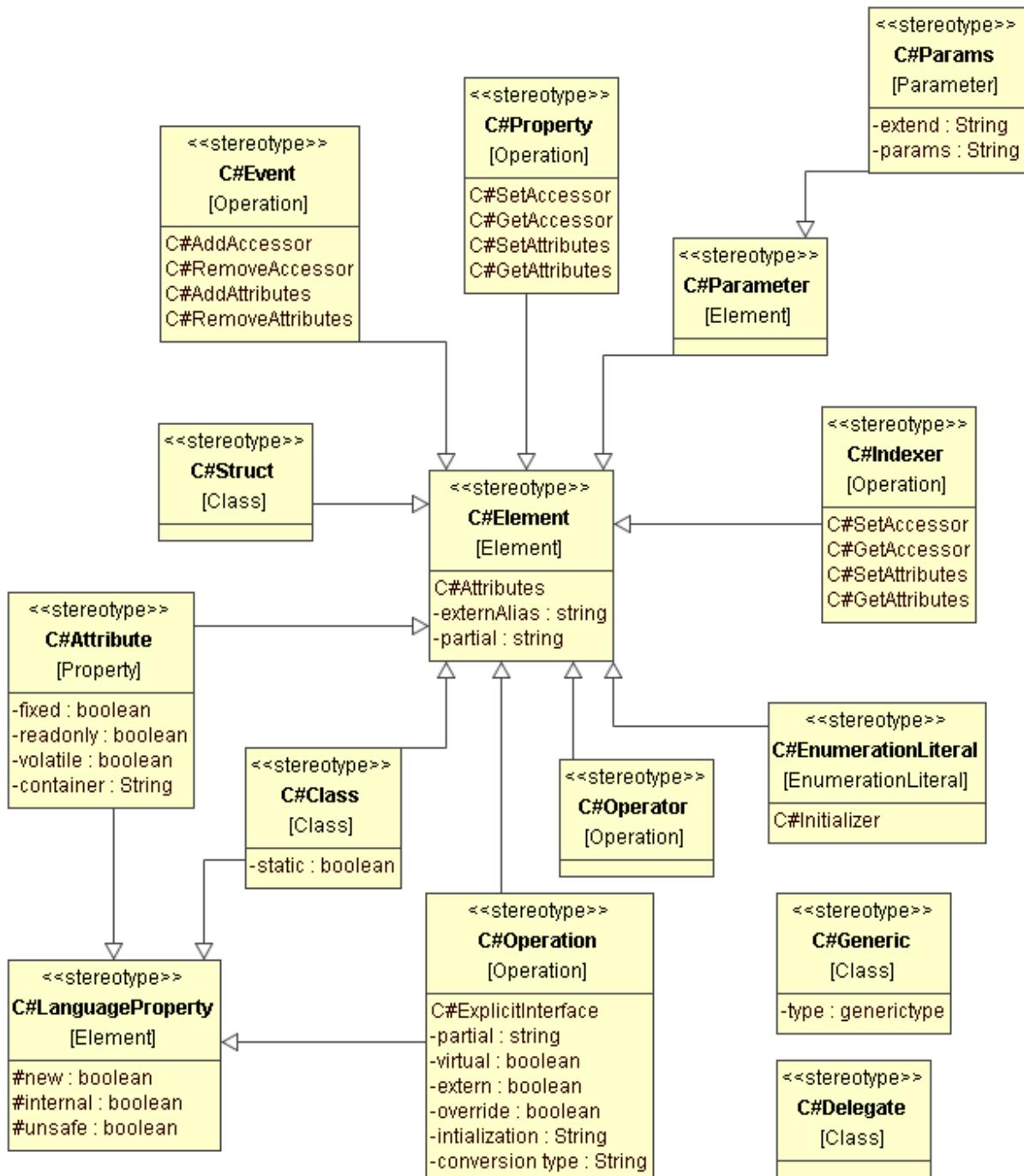


Figure 1 -- C# Profile

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	<p>Represents the usage of Fixed Size Buffer</p> <pre>unsafe struct A { public fixed int x[5]; }</pre>

readonly	Boolean	Represents the usage of read-only attribute Class A { public readonly int x; }
volatile	Boolean	Represents the usage of volatile attribute Class A { public volatile int x; }

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 static Class A { public int x; }

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 extern alias X; class Test { X::N.A a; }
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	
unsafe	boolean	Represents the usage of unsafe element unsafe struct A { public fixed int x[5]; }

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 { public extern int x(); }
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.

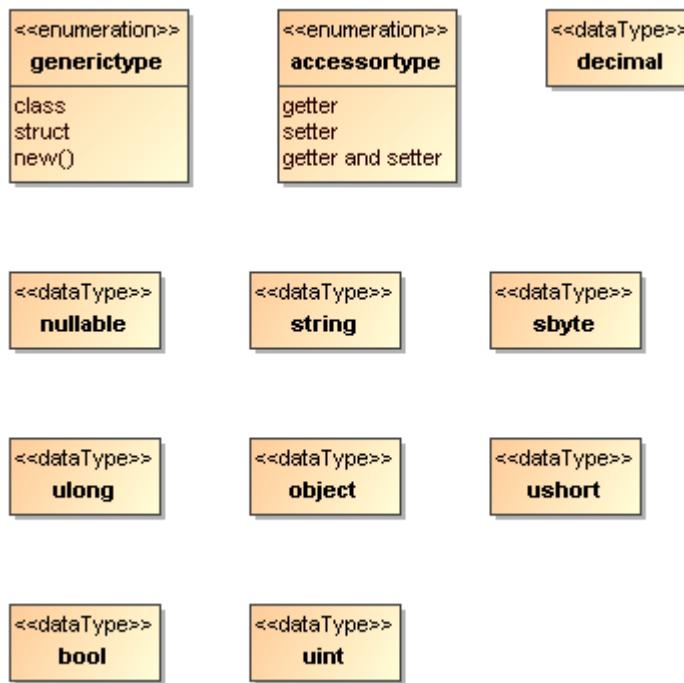


Figure 2 -- Data Type of C# Profile

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

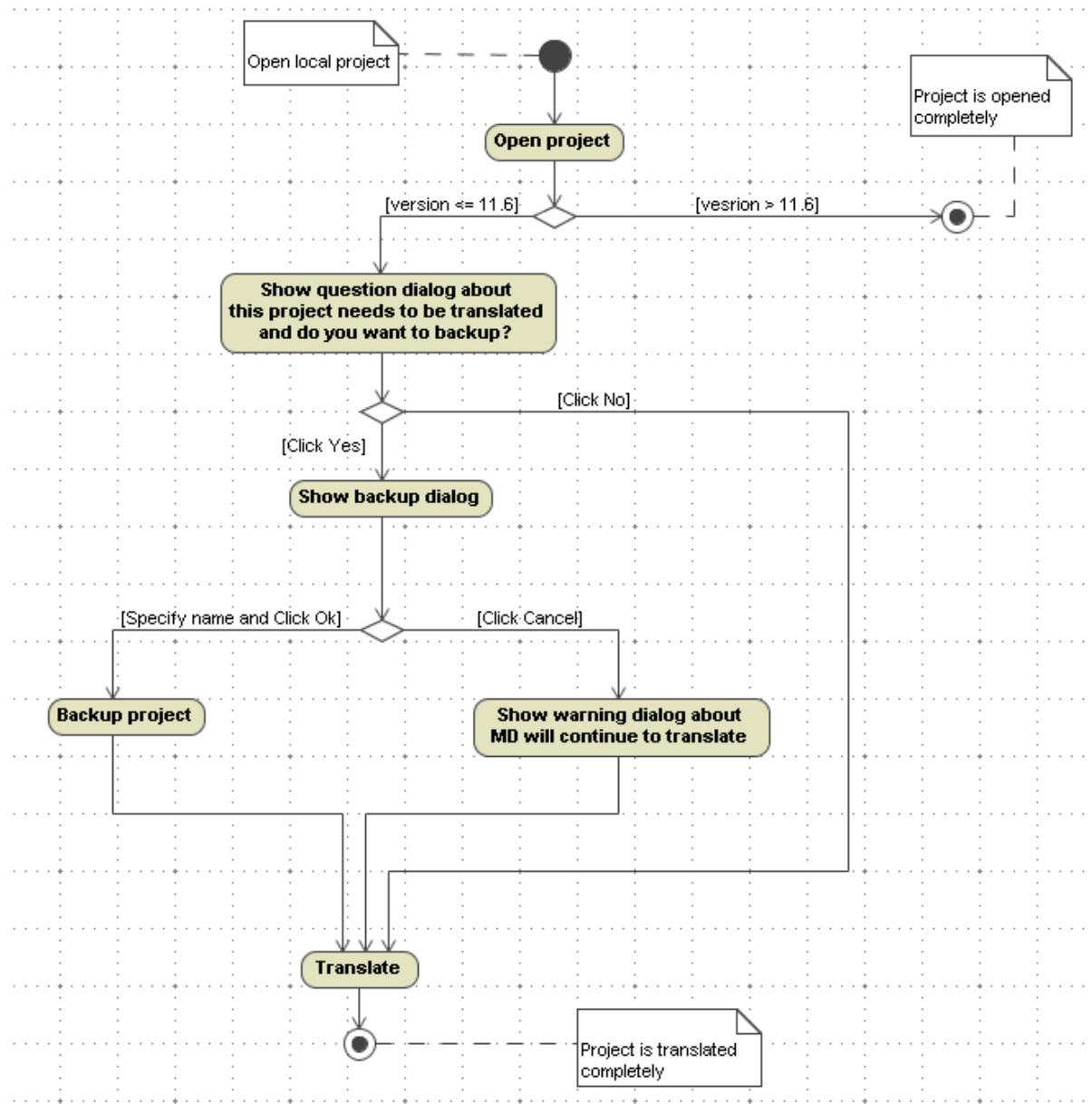


Figure 3 -- Open local project Activity Diagram

Open teamwork project

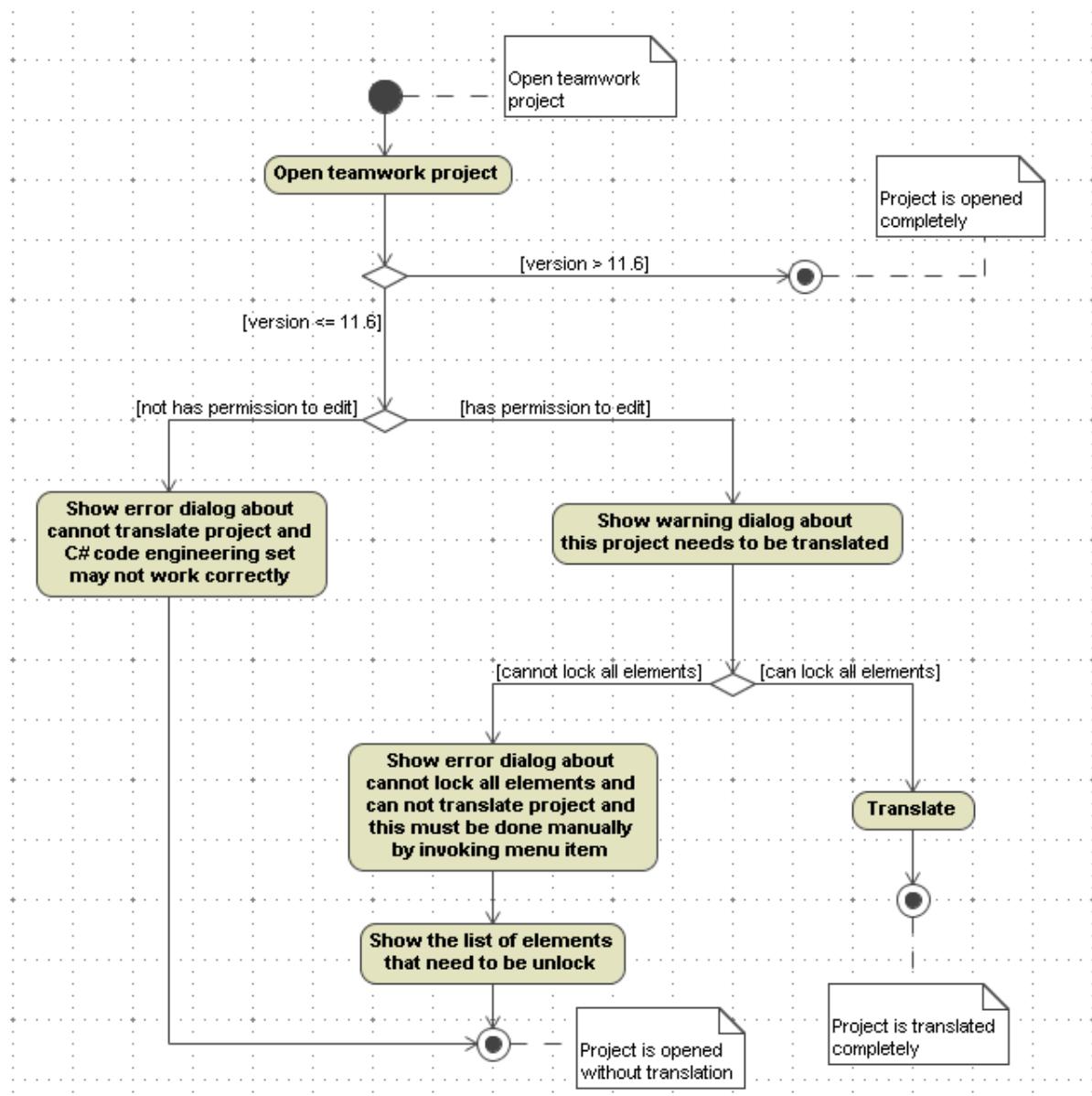


Figure 4 -- Open teamwork project Activity Diagram

Import MagicDraw project

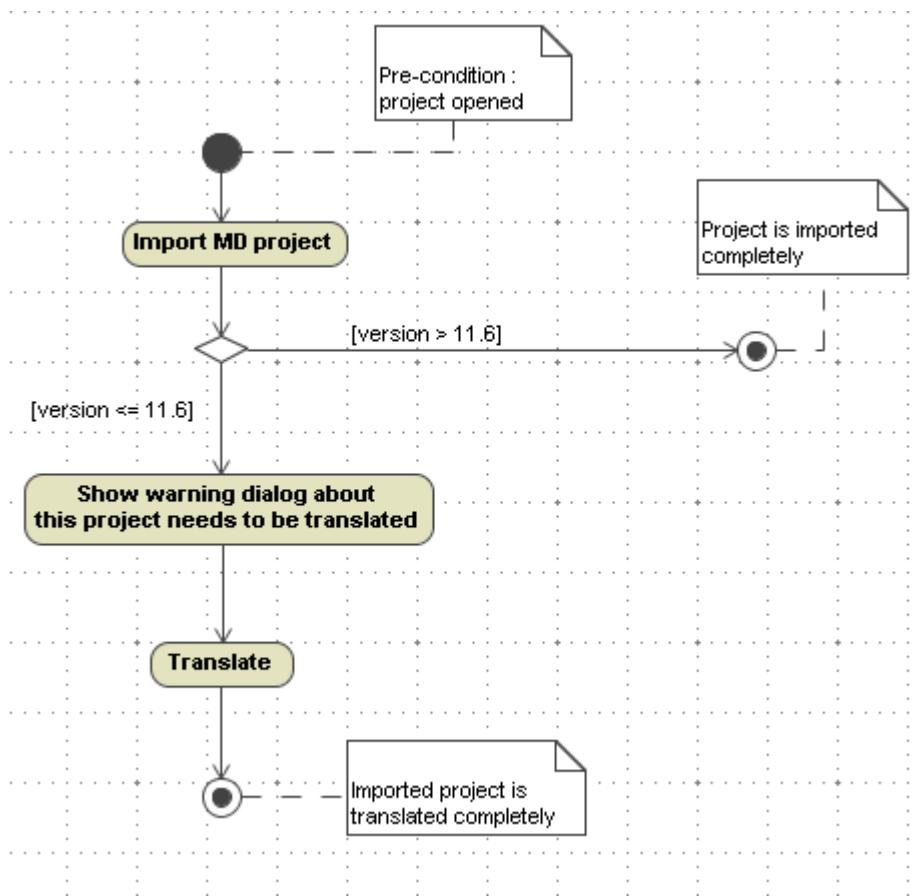


Figure 5 -- Import MagicDraw project Activity Diagram

Use module

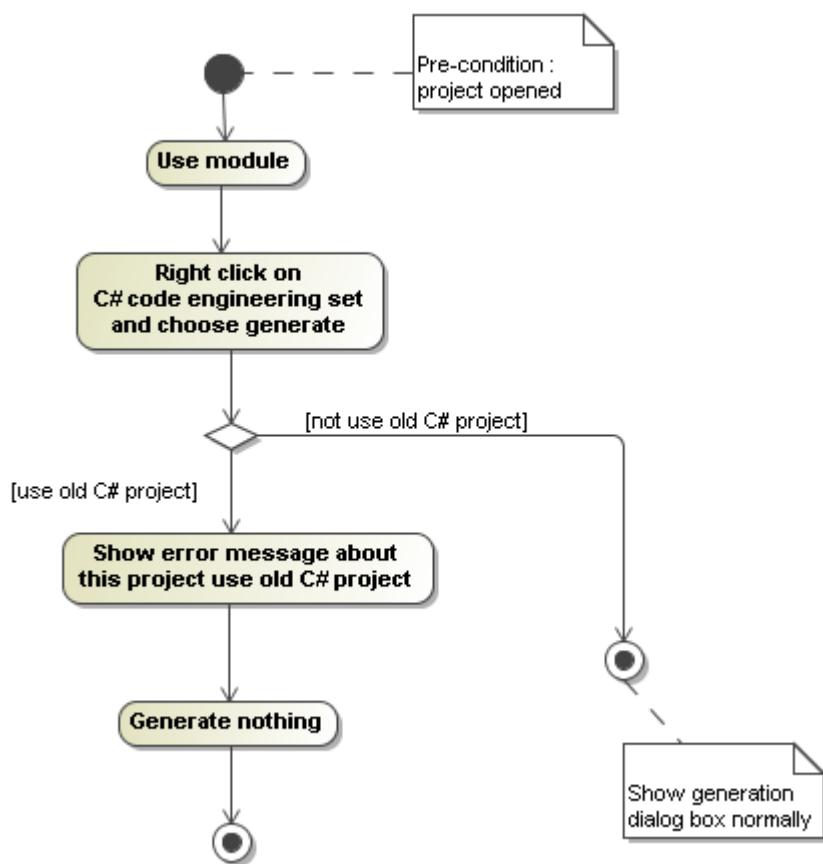


Figure 6 -- Use module Activity Diagram

Update C++ Language Properties and Profiles

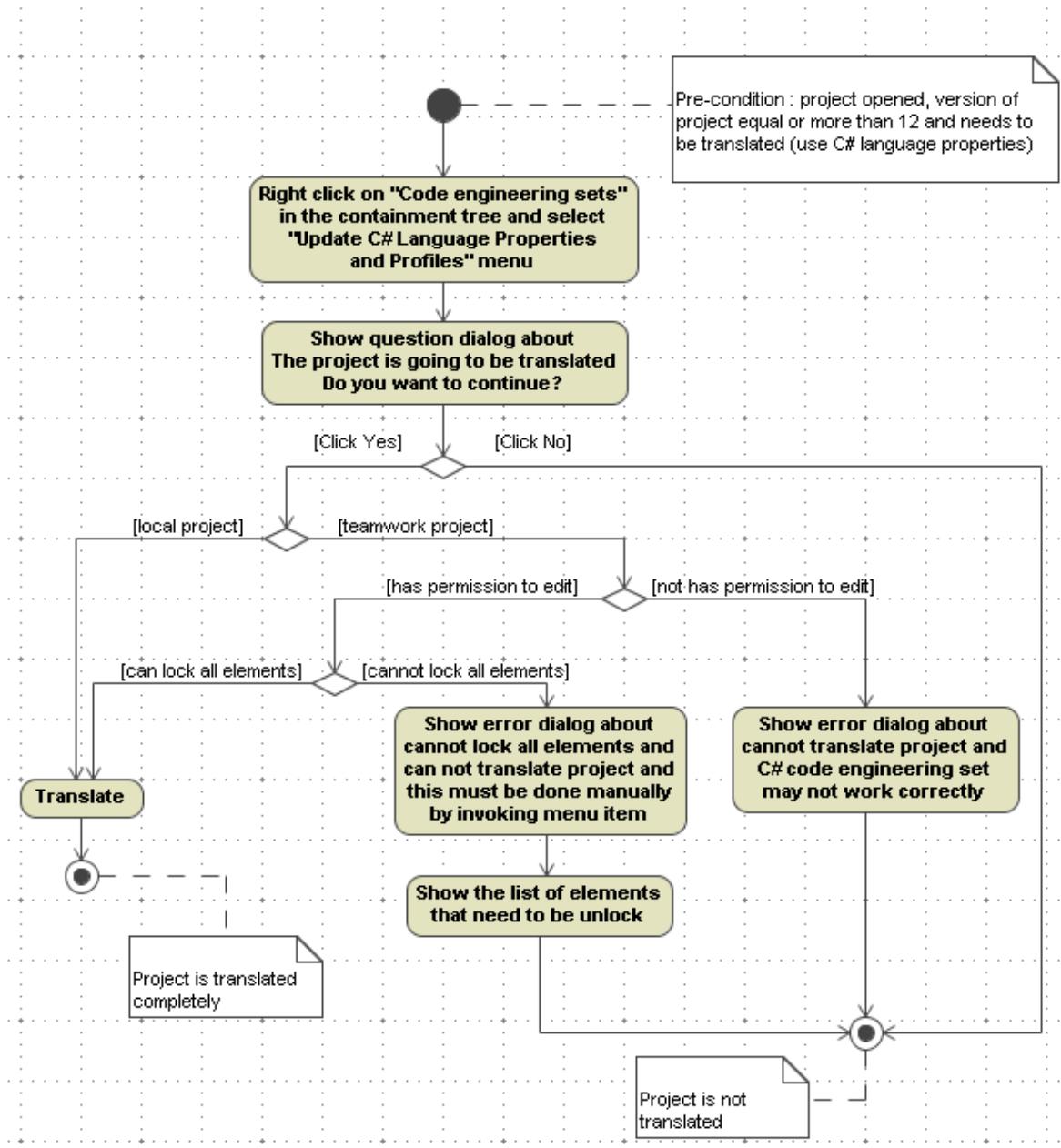


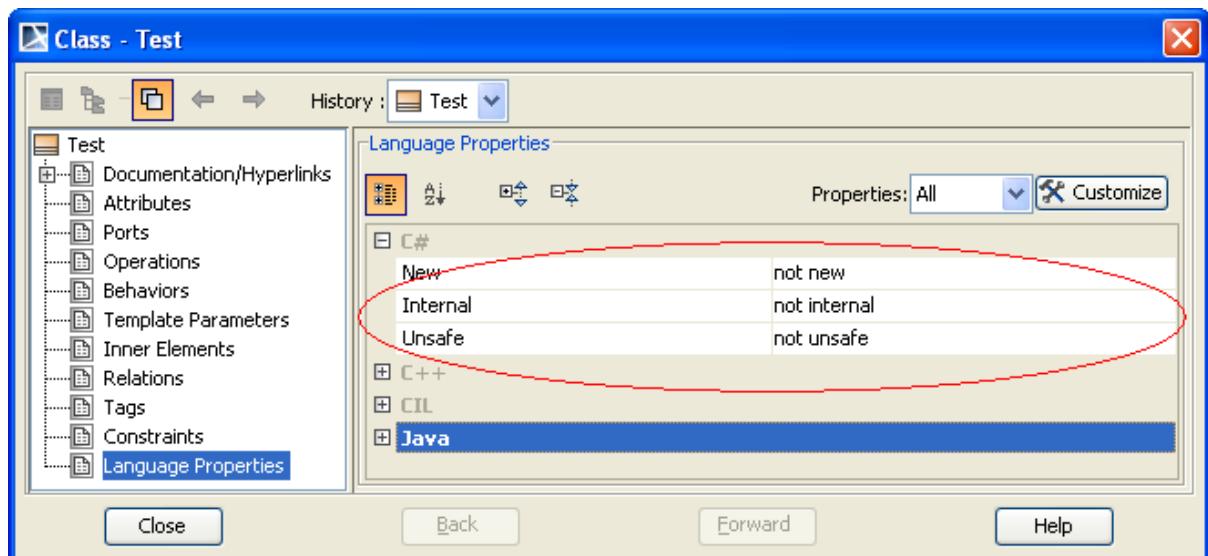
Figure 7 -- Update C# Language Properties and Profiles Activity Diagram

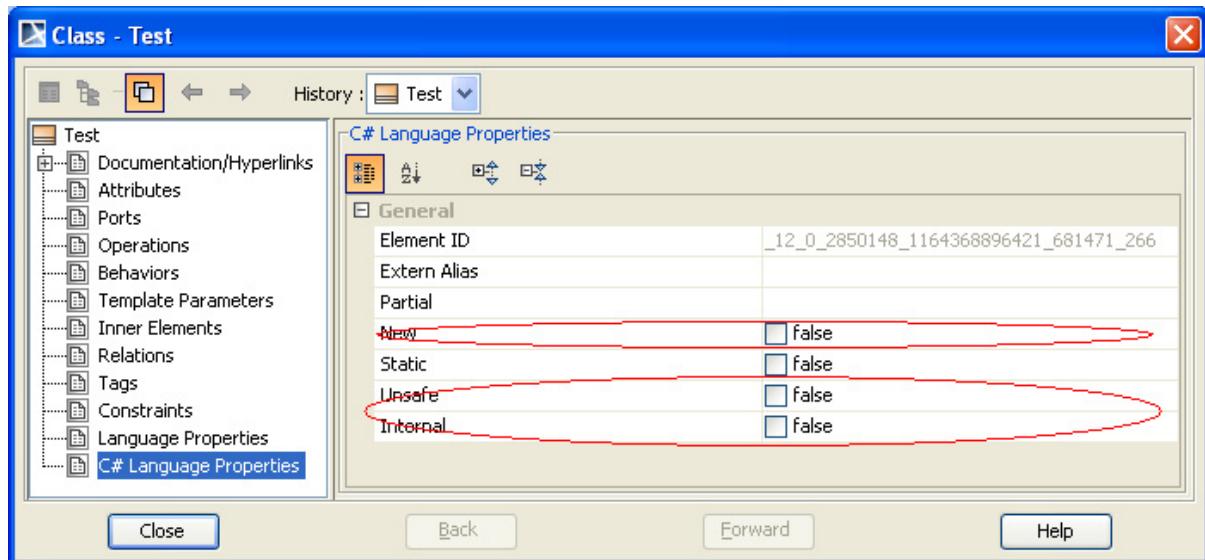
■ Mapping

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)

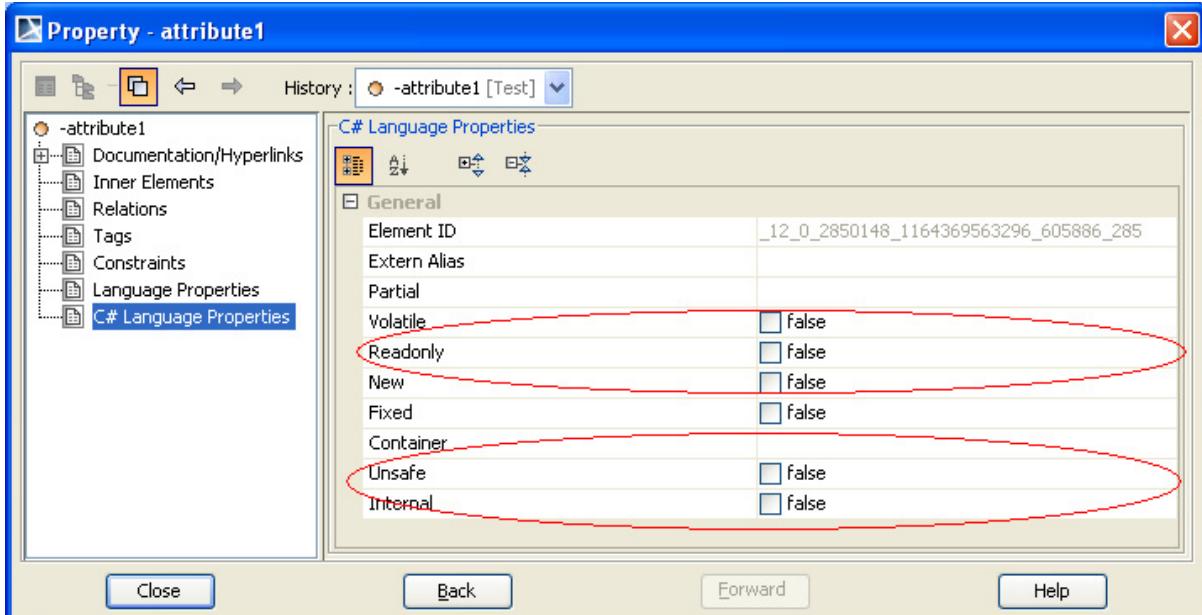
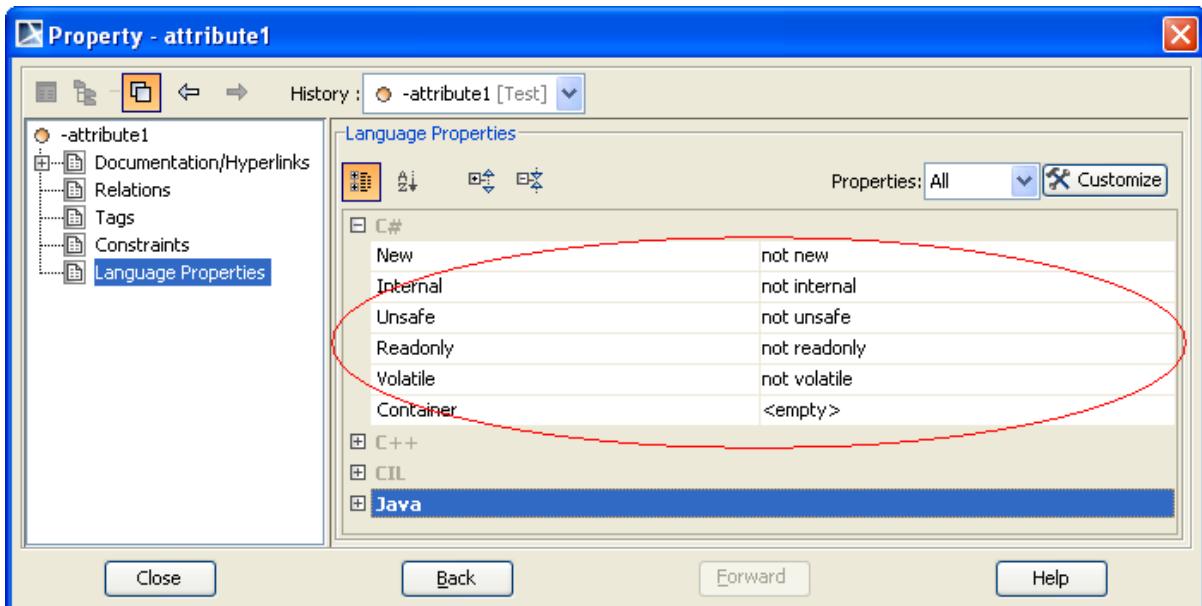
Class





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")

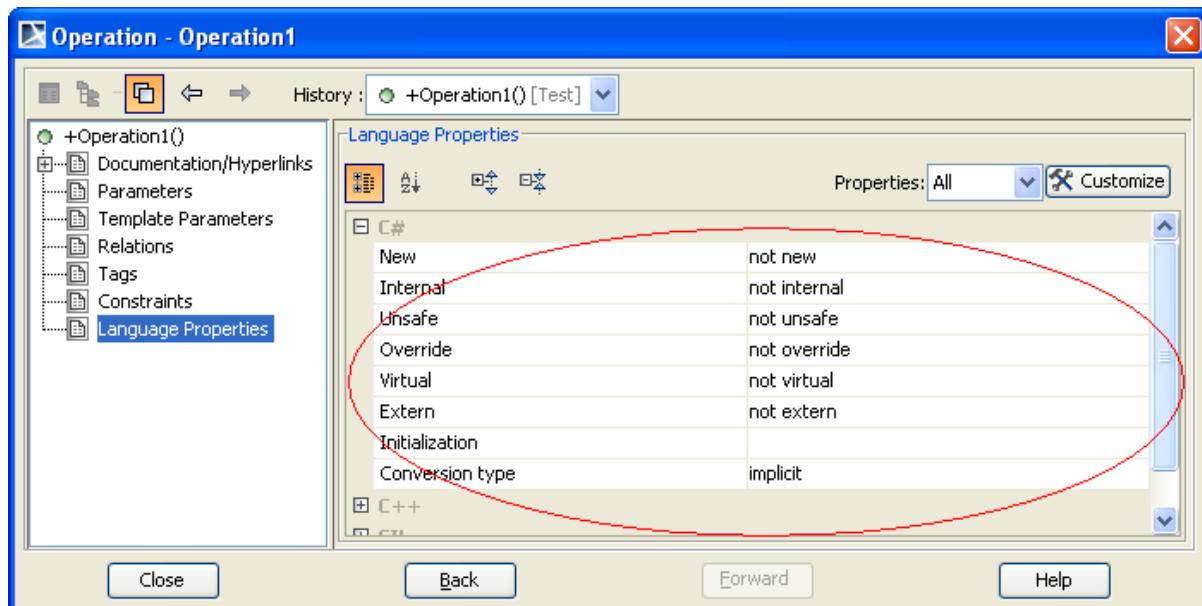
Attribute

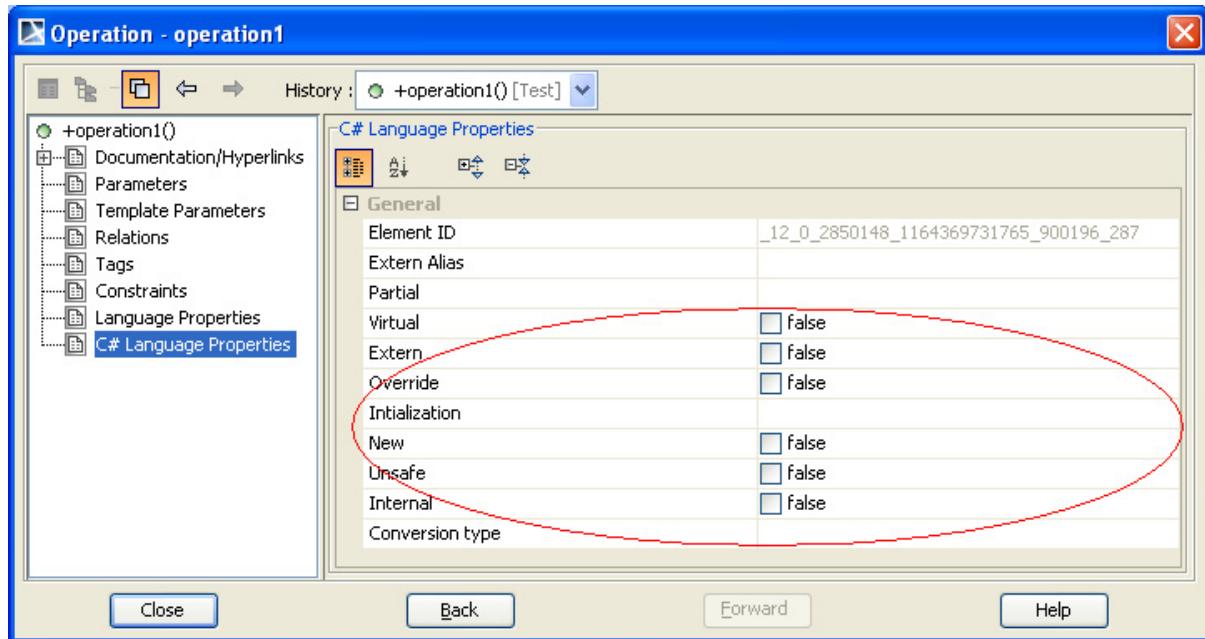


Old Value	Translation
New	Mapped to tag "new" of <<C#LanguageProperty>> (The usage for this is to apply <<C#Attribute>> and set value to tag "new")

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 "virtual" 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.

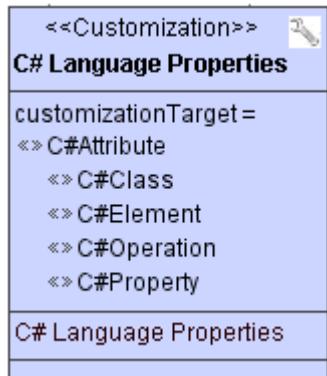


Figure 8 -- Customization Class

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.

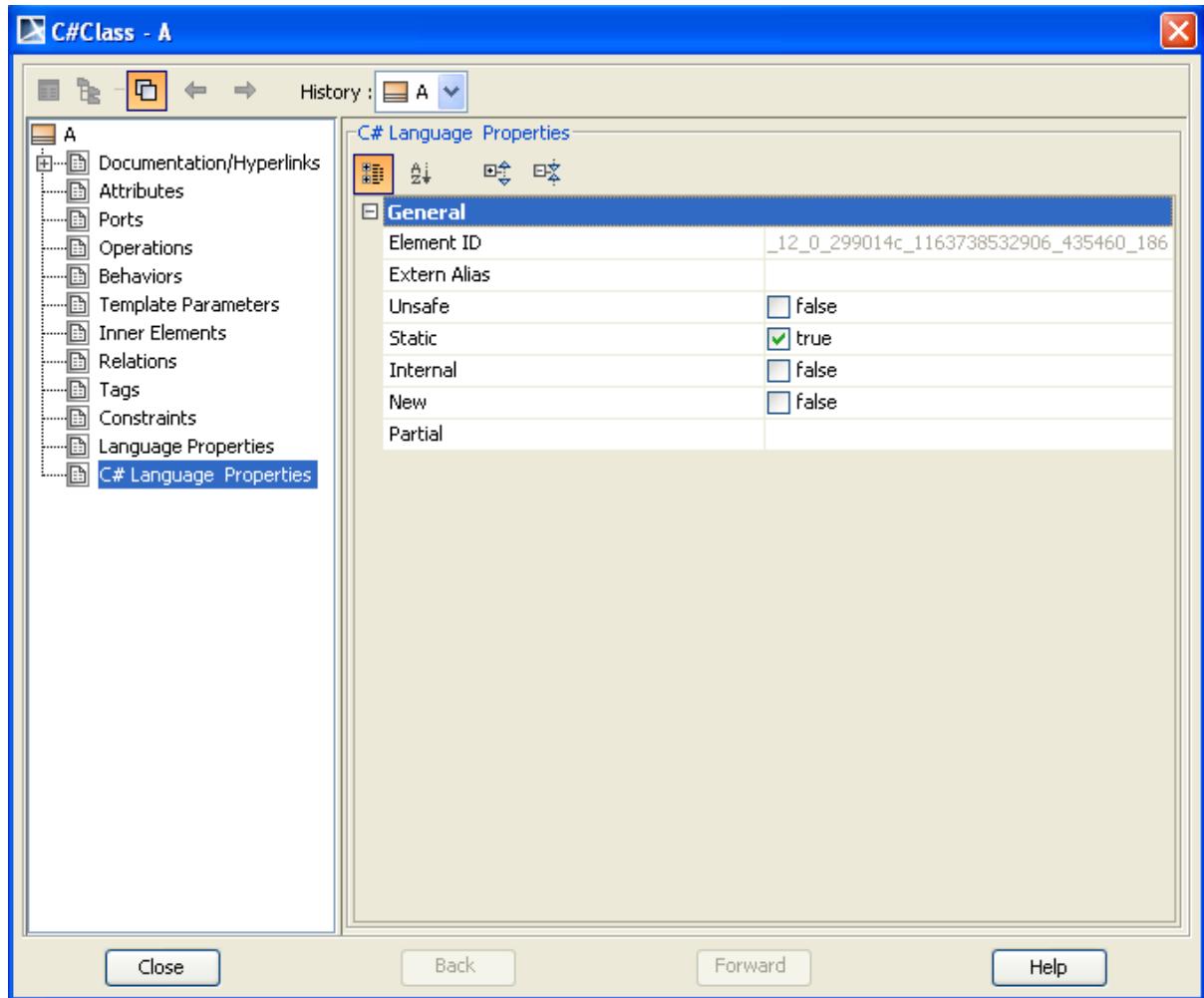


Figure 9 -- DSL Properties

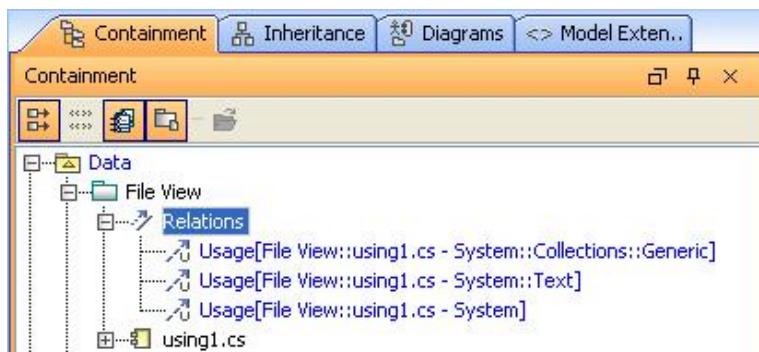
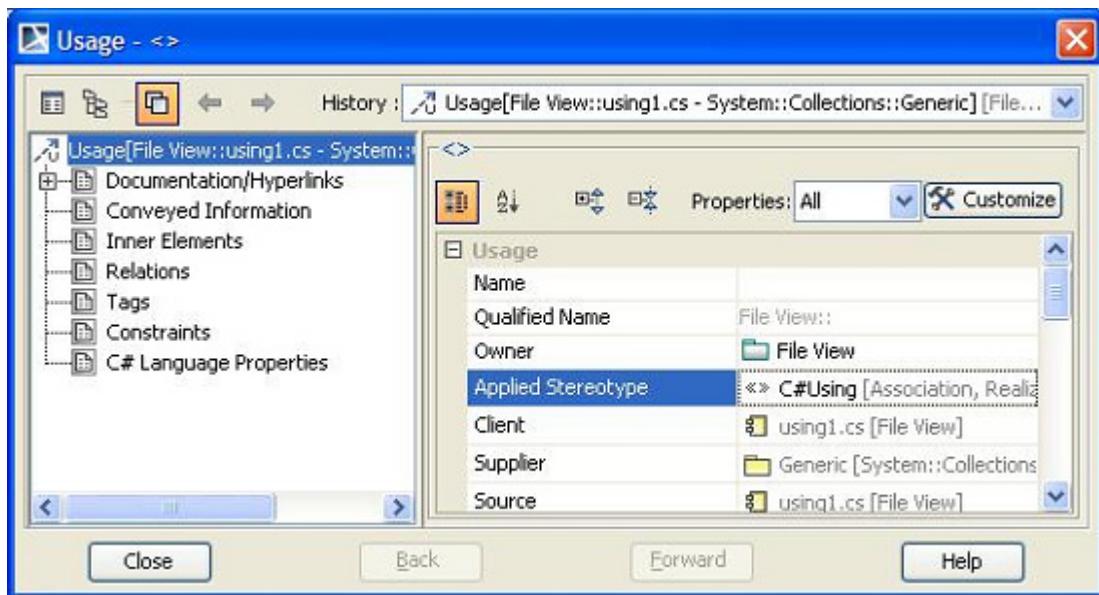
Using Directive Mapping

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>	<p>The MD-UML diagram illustrates the mapping of the provided C# code to UML components. At the top center is a yellow component box labeled <<component>> using1.cs. Three dashed arrows point from this box to three separate purple rectangular boxes below it, each labeled with a namespace name: Generic, Text, and System. Each of these three boxes has a small blue arrow pointing downwards, indicating they are sub-components of the main using1.cs component.</p>

Open usage dependency specification
apply <<C#Using>> stereotype.
For using namespace, leave the name empty.



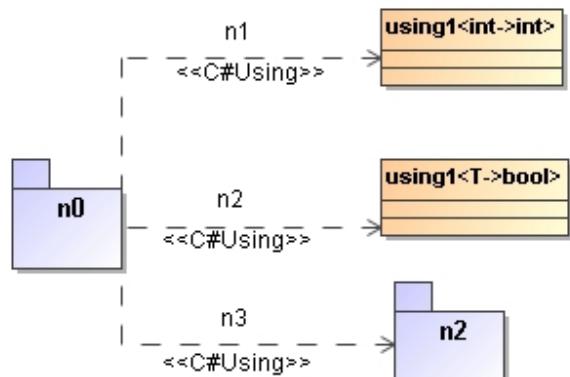
The following example shows the mapping of C# Using alias. The usage dependency for C# Using alias declared in the namespace will be created under that namespace.

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

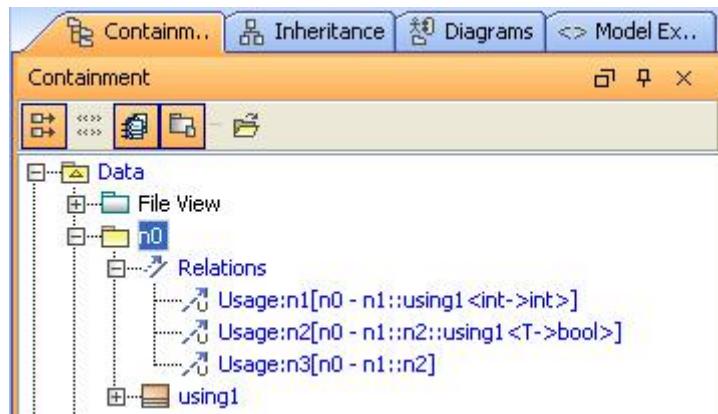
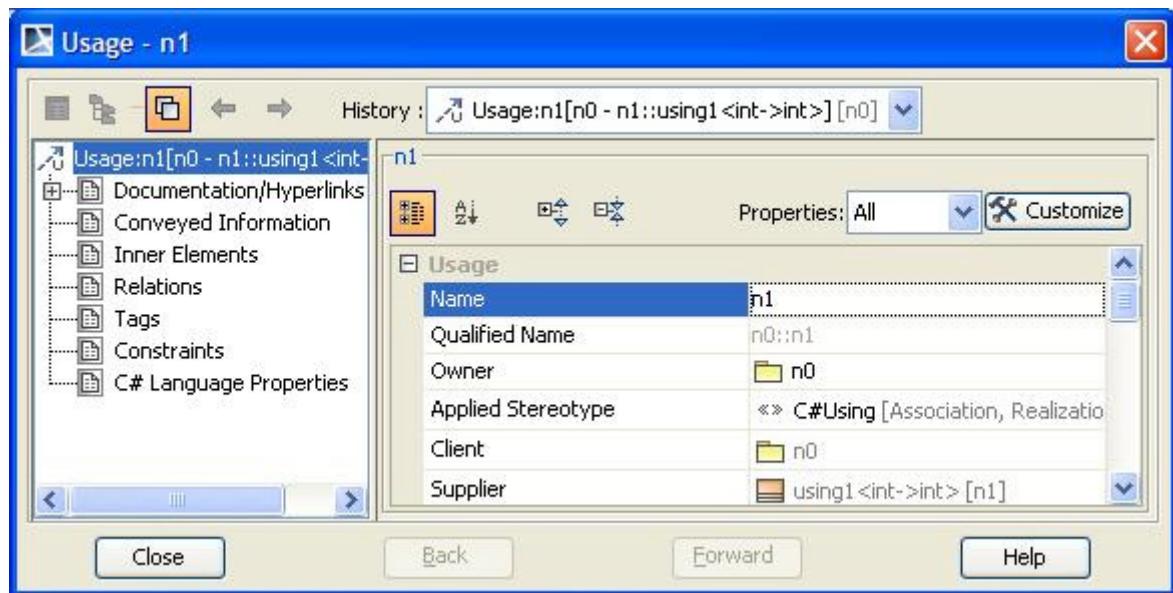
C# CODE ENGINEERING

Mapping

```
namespace n0{    using n1 =  
n1.using1<int>;    using n2 =  
n1.n2.using1<bool>;    using n3 =  
n1.n2;    class using1 : n2 {  
n1 a;        n2 b;  
n3.using1<float> c;    } } namespace  
n1{    namespace n2 {        class  
using1<T> {    }    class  
using1<T> {    } }
```



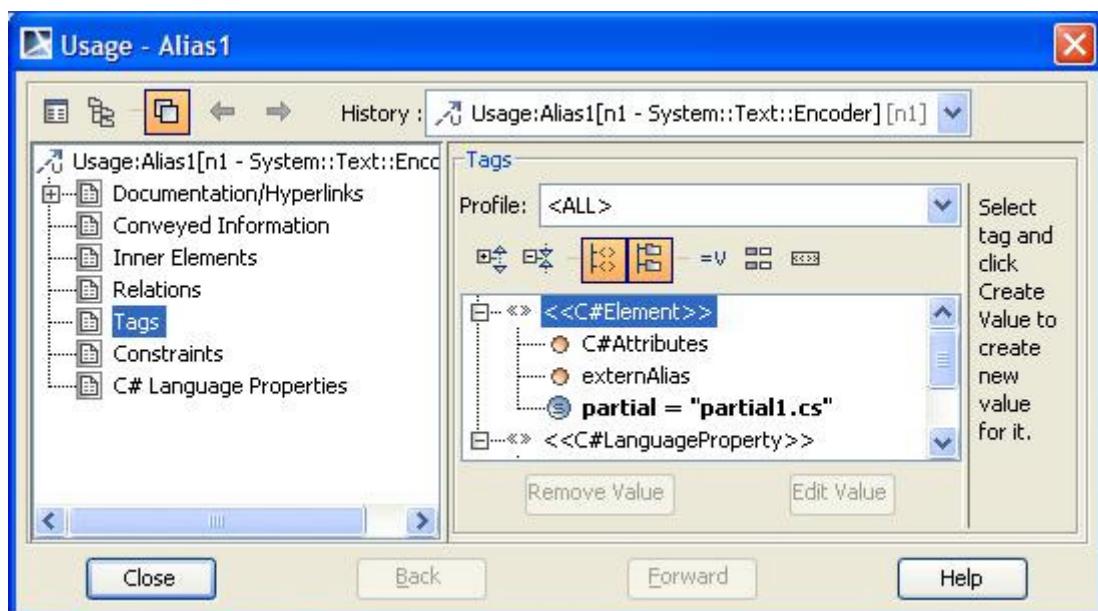
Open usage dependency specification
Apply <<C#Using>> stereotype.
For using alias, enter the alias name.



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
------	--------

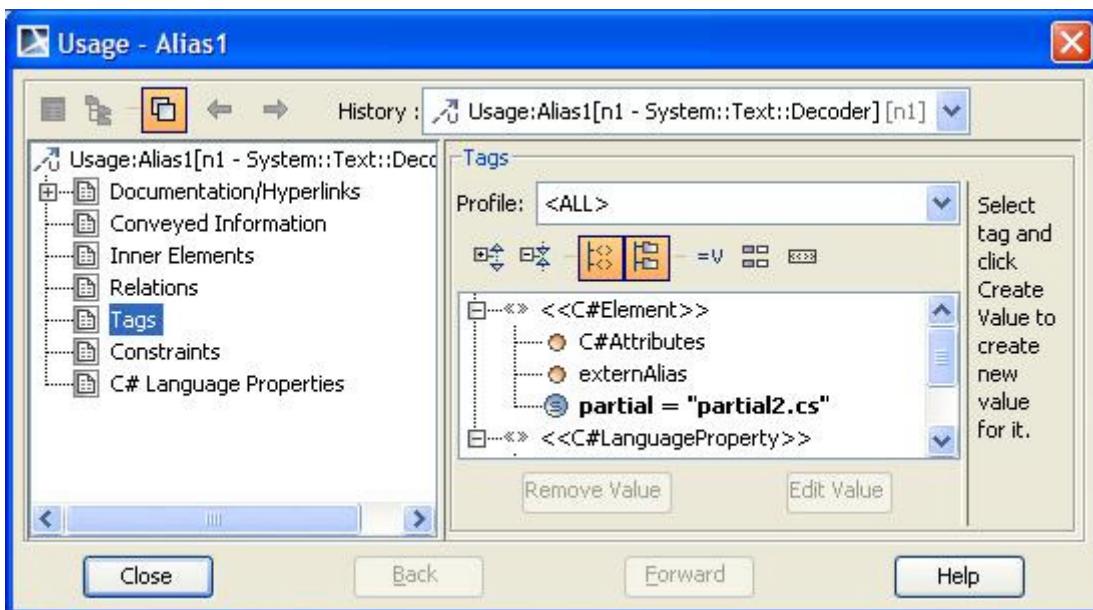
```
//partial1.cs
namespace n1 {
    using LL =
System.Text.Encoder;
    partial class using1
    {
        LL a;
    }
}
```



C# CODE ENGINEERING

Mapping

```
//partial2.cs
namespace n1 {
    using LL =
System.Text.Decoder;
    partial class using1
    {
        LL b;
    }
}
```



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.

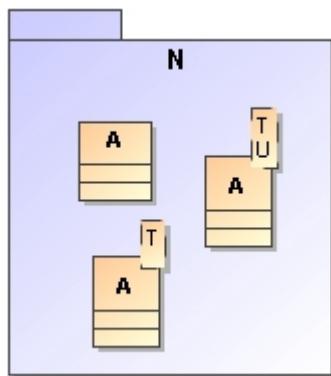


Figure 10 -- Generic Classes

How to manually create class A in the model, and Figure 12 shows the error message when trying to create class A.

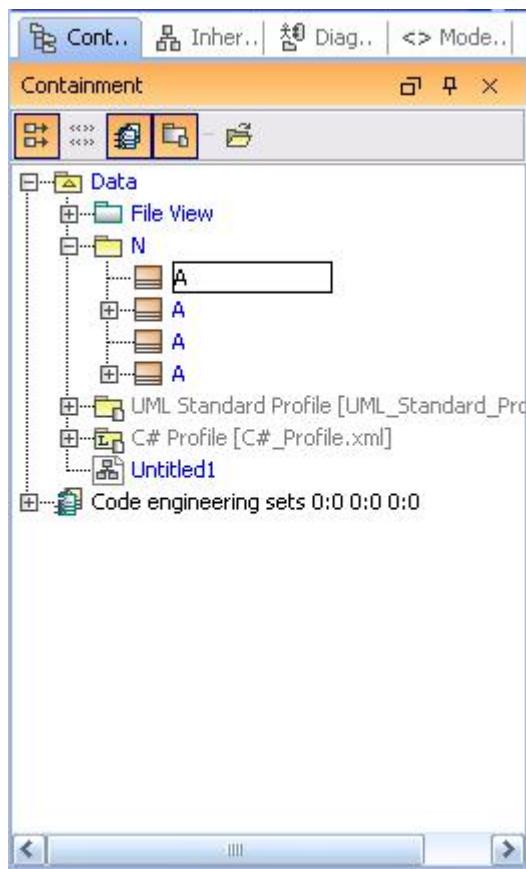


Figure 11 -- Creating Class A in The Model



Figure 12 -- 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>

```
//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;
    }
}
```

```
public partial class PartialA{

int a;

string actionB( )
{return ;}

public class B
{int b; }

public class C
{int c; }

}
```

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 appear any more.

DATABASE ENGINEERING

MagicDraw UML has the tools that forward engineer UML constructs to DDL script. The resulting DDL script depends on the selected DDL language dialect.

Retrieve DB Info dialog box

IMPORTANT! Since MagicDraw version 9.0, **Retrieve DB Info** dialog box has been moved to the **Round Trip Set** dialog box.

The Retrieve DB info function allows you to retrieve the information from the existing database or ODBC source. Database structure is retrieved as UML model, using class diagram elements.

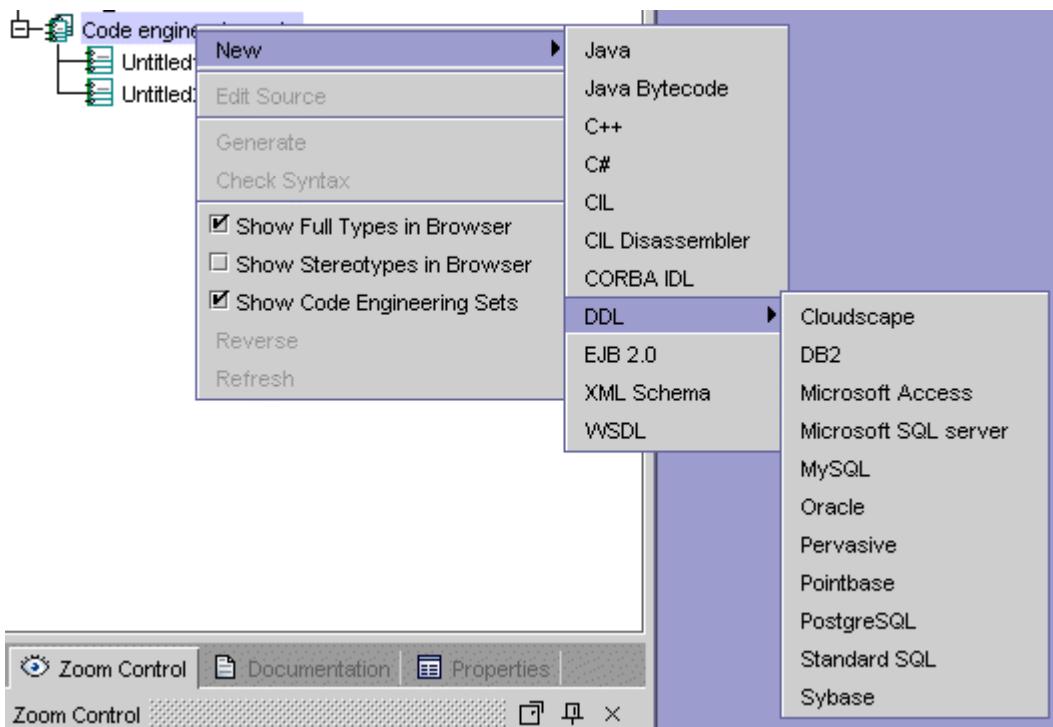
The mapping of the DB to UML is the same as in DDL reverse engineering.

NOTE As the retrieve DB info function uses JDBC bridge, you may need to have JDBC drivers. All types of drivers are valid for making a connection. The driver should be able to retrieve the database metadata information. The same applies to the database; it should be able to provide this information.

To open the **Retrieve DB Info** dialog box dialog box

-
1. From the **Code Engineering Sets** shortcut menu, choose **New**.

2. Choose DDL and then choose the database vendor you need. (possible choices include: **Cloudscape**, **DB2**, **Microsoft Access**, **Microsoft SQL Server**, **MySQL**, **Oracle**, **Pervasive**, **Pointbase**, **PostgreSQL**, **Standard SQL**, and **Sybase**.



3. The new set is created.
4. From the created set shortcut menu, choose **Edit**. The **Round Trip Set** dialog box appears. Select the **Reverse from DB** option.

- From the **Tools** menu, choose **Quick Reverse** and then choose DDL and required database vendor. The **Round Trip Set** dialog box appears. Select the **Reverse from DB** option.



Figure 1 -- Retrieve DB Info dialog box

Box name	Function
Recently Used	Contains the list of the recently used reverse templates. Choose the one you need and click Apply .
DB Connection URL	The connection URL for the selected profile.

Box name	Function
Driver Files	<p>Contains .jar and .zip files or directories with JDBC driver's classes.</p> <p>To choose the files or directories you want to add or remove, click the “...” button. The Select Files and/or Directories dialog box appears.</p> <p>NOTE If the driver file is empty, Driver Class is searched from the classpath.</p>
Driver Class	<p>Contains the connection driver class.</p> <p>Click the “...” button and the list of available driver classes that are available in the selected driver files is displayed.</p> <p>NOTE Only in the files that are selected in the Driver Files list, the system searches for driver classes .</p>
Username	Type the username to connect to the database.
Password	Type the password to connect to the database.
Catalog	<p>Contains a name of the selected Catalog.</p> <p>To retrieve the list of available Catalogs from the database, click the “...” button and select the catalog. The catalog name appears in the Catalog text box.</p> <p>NOTE Only when all other properties in this dialog box are correctly defined, the list of catalogs can be retrieved.</p>
Schema	<p>Contains a name of the selected Schema.</p> <p>To retrieve the list of available Schemas from the database, click the “...” button and select the schema. The schema name appears in the Schema text box.</p> <p>NOTE Only when all other properties in this dialog box are correctly defined, the list of schemas can be retrieved.</p>
Property Name	<p>The name of the JDBC driver property.</p> <p>NOTE: If using Oracle drivers, while retrieving db info from Oracle db:</p> <ul style="list-style-type: none"> ● To retrieve comments on table and column, set property as remarks=true. ● To connect to a db as sysdba, set property as internal_logon=sysdba.
Debug JDBC Driver	If selected, all output from a JDBC driver will be directed to Message Window.
Relode Driver	By default, the Reload Driver check box is selected. If you want that driver to not be reloaded, clear the check box.

Forward engineering to DDL script

This section describes to what data model constructs MagicDraw constructs are converted.

Packages

The Database is represented as a Package with the <<database>> stereotype. Each View or Table can be assigned to a Schema where the Schema is represented as a Package with the <<schema>> stereotype.

A Package, depending on a package stereotype, is mapped to one of the following DDL constructs:

- A Database, if the package has the <<database>> stereotype.
Elements: CREATE DATABASE <database_name>
Database name is equal to the package name.
- A Schema, if the package has the <<schema>> stereotype.
Elements: CREATE SCHEMA [<database_name>.]<schema_name>
Schema name is equal to the package name. Schema database name is the name of a package that contains schema package.
- Otherwise it is not mapped.

NOTE

If a package has no stereotype and the EnableDefaultStereotypes property is true, the <<database>> stereotype is used for the first level packages, and the <<schema>> stereotype is used for the second level packages.

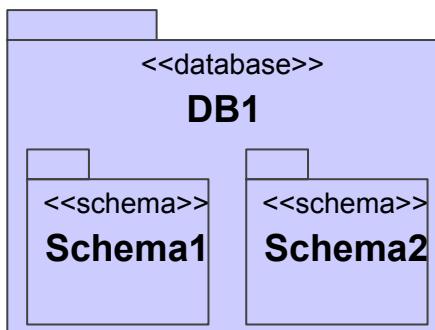


Figure 2 -- Package mapping example

DDL script, generated for example showed in Package mapping example12, creates one Database and two Schemas:

```
CREATE DATABASE DB1;
CREATE SCHEMA DB1.Schema1;
CREATE SCHEMA DB1.Schema2;
```

Classes

A class, depending on class stereotype, is mapped to one of the following DDL constructs:

- Table, if the class has the <>table<> stereotype.
Elements: CREATE TABLE [<schema_name>.]<table_name> (<column_and_constraint_list>)
Table name is equal to the class name. Table schema name is the name of package that contains table class.
- View, if the class has the <>view<> stereotype.
Elements: CREATE VIEW [<schema_name>.]<view_name> [(<column_list>)] AS SELECT <derived_column_list> FROM <table_list>.
View name is equal to the class name. The view schema name is the name of a package that contains view class. Table list within view “FROM” clause are derived from dependencies between the view class and tables classes.
- Otherwise it is not mapped.

NOTE

If a class has no stereotype and the EnableDefaultStereotypes property is true, the class is treated as a class with the <>table<> stereotype.

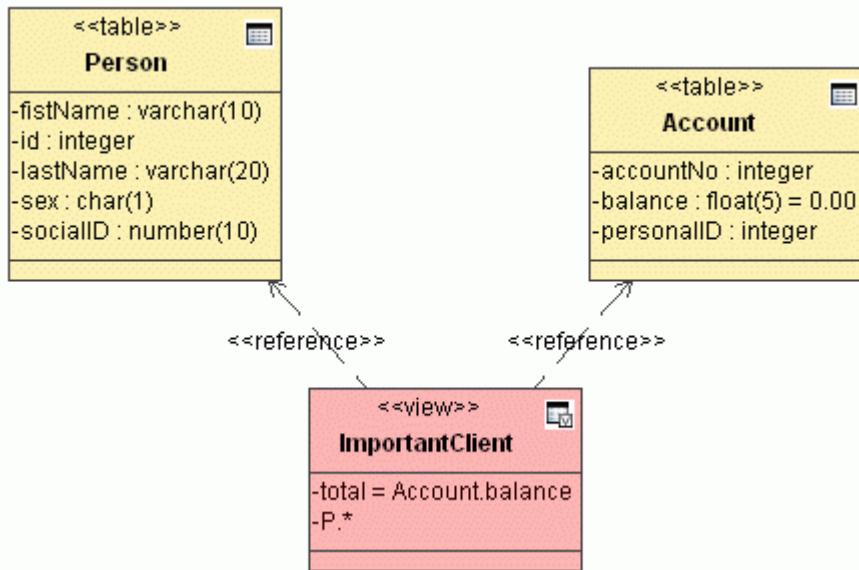


Figure 3 -- Class mapping example

DLL script, generated for classes is showed in Class mapping example. It has two Table definition statements (CREATE TABLE) and view definition statement (CREATE VIEW):

```

CREATE TABLE Person (
    id integer,
    socialId number (10),
    lastName varchar (20),
    firstName varchar (10),
    sex char (1)
);
CREATE TABLE Account (
    accountNo integer,
    balance float (5) DEFAULT 0.0,
    personalId integer);
CREATE VIEW ImportantClient
AS SELECT P.*, Account.balance as total
FROM Person, Account;
  
```

Attributes

An attribute of a class with the `<<table>>` or the `<<view>>` stereotype, depending on an attribute stereotype, is mapped to one of the following DDL constructs:

- A Column of a Table, if the class that contains an attribute has the <<table>> stereotype. A Column name is equal to the name of an attribute. A Column type is equal to the type of an attribute. Column default value is equal to the initial value of an attribute (if any).
- A Column of a View, if the class that contains an attribute has the <<view>> stereotype. Elements: [<column_expression> AS] <column_name> A Column name is equal to the name of an attribute. Column expression is equal to the initial value of an attribute (if any).

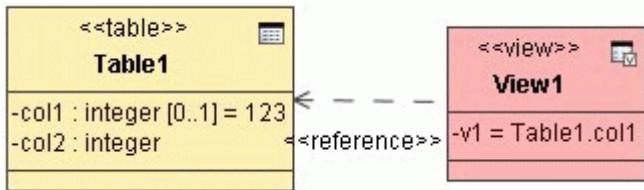


Figure 4 -- Attributes mapping example

Col1 attribute of the Table1 class (see Attributes mapping example14) is mapped to the col1 column of a Table1 table, and col1 attribute of the View1 class is mapped to the col1 column of a View1 view. There is DDL script for attributes mapping example:

```

CREATE TABLE Table1 (
    col1 integer DEFAULT 123
);
CREATE VIEW view1
    AS SELECT Table1.col1 AS v1
    FROM Table1;
  
```

Operations

An operation (method) of a class with the <<table>> stereotype, depending on an operation's stereotype, is mapped to one of the following DDL constructs:

- An Index for a Table, if the operation has the <<index>> stereotype;
Elements: CREATE INDEX [<schema_name>.]<index_name> ON <table_name>(<column_list>) An Index name is equal to the name of an operation. Names of comma-delimited set of column for the index are equal to the parameter names of an operation.
- Tags with names: trigger action time, trigger event and triggered action has meaning in DDL CG have meaning in the DDL CG if are specified for an operation with stereotype <. The "implementation" tag specified for operation with stereotype trigger has no meaning for the DDL code generation since MD 6.0 version. The "implementation" tag was split to three

tags: "trigger action time", "trigger event" and "triggered action". Values of these tags will be used then generating definition of a trigger. ::= CREATE TRIGGER ON [REFERENCING]

- A constraint, if the Operation has the <>PK>>, <>unique>>, or <>check>> stereotype. Elements: [ALTER TABLE <table_name> ADD] CONSTRAINT <constraint_name> Constraint name is equal to the name of an operation.
 - PRIMARY KEY (<pk_column_list>) Primary key column list contains all attributes with the <>PK>> stereotype.
 - UNIQUE (<unique_column_list>) Unique constraint element <unique_column_list> is generated from operation's parameter names.
 - CHECK (<check_expression>) Check constraint element <check_expression> is generated from tag named Implementation.
Otherwise it is not mapped.

NOTES

- IndexNamePrefix property specifies an optional naming standard that is added to the beginning of the name for each generated index.
- TriggerNamePrefix property specifies an optional naming standard that is added to the beginning of the name for each generated trigger.

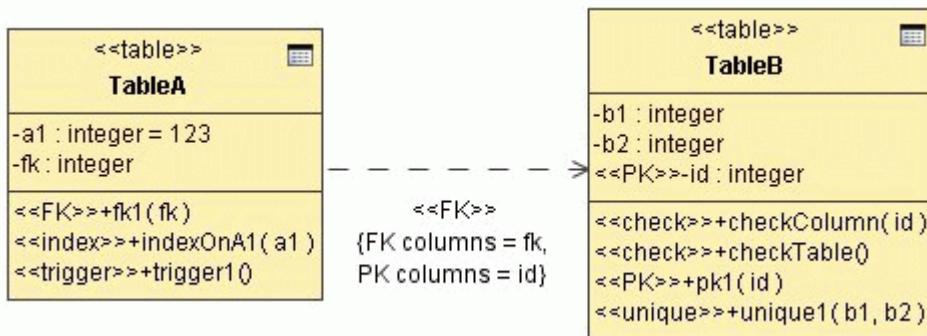


Figure 5 -- Operations mapping example

An example of the DDL script for operations mapping:

```

CREATE TABLE TableB (
  id integer
  CONSTRAINT checkColumn CHECK /*<check_expression>*/,
  b1 integer,
  b2 integer,
  CONSTRAINT pk1 PRIMARY KEY (id),
  CONSTRAINT checkTable CHECK /*<check_expression>*/,
  ...
)
  
```

```

CONSTRAINT unique1 UNIQUE (b1,b2)
);

CREATE TABLE TableA (
fk integer,
a1 integer DEFAULT 123,
CONSTRAINT fk1 FOREIGN KEY (fk) REFERENCES TableB(id)
);
CREATE INDEX indexOnA1 ON TableA(a1);
CREATE TRIGGER trigger1 ON TableA /*<triggered_SQL_statement>*/;

```

Relationship cardinalities

One-to-many (1:N) relationship

One-to-many (1:N) relationship is mapped to dependency with <>FK>> stereotype.

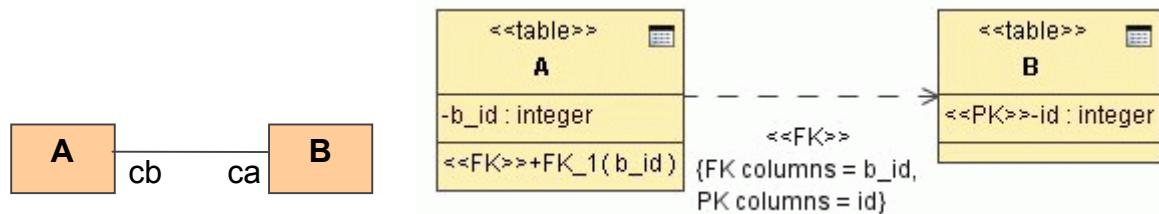


Figure 6 -- Cardinalities of 1:N relationship

For 1:N relationship allowed cardinalities for end A are 0, 1, N. For end B allowed cardinalities are 0, 1 (see Cardinalities of 1:N relationship16 -- Cardinalities of 1:N association).

Concrete cardinalities of A and B ends are mapped to different data model constraints (if any):

1. If A end does not allow N cardinality, the UNIQUE constraint is assigned to b_id column.
When b_id column has the UNIQUE constraint, every instance of A class references a unique instance of B class (if any). Given B class instance is associated with the unique instance of A class (if any). This means that A end has no N cardinality.
2. If B end does not allow 0 cardinality, NOT NULL constraint is assigned to b_id column.
When b_id column has NOT NULL constraint, every A class references some B class. This means that ca role does not allow 0 cardinality.

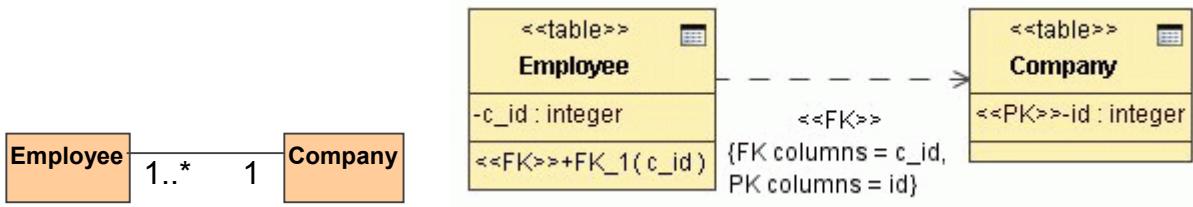


Figure 7 -- 1:N relationship and mapped representation

Script creating 1:N relationship:

```

CREATE TABLE Company (
    id INTEGER NOT NULL PRIMARY KEY;
CREATE TABLE Employee (c_id INTEGER NOT NULL,
    CONSTRAINT FK_1 FOREIGN KEY (c_id) REFERENCES Company (id));
  
```

One-to-one (1:1) relationship

One-to-one (1:1) relationship is handled as a special case of one-to-many relationships, where end B allowed cardinalities are 0, 1 and end A allowed cardinalities are 0, 1 (see Cardinalities of 1:N relationship16 -- Cardinalities of 1:N association).

1:1 relationship is mapped to a dependency with <<FK>> stereotype. The 1:1 cardinality must be forced through constraints.

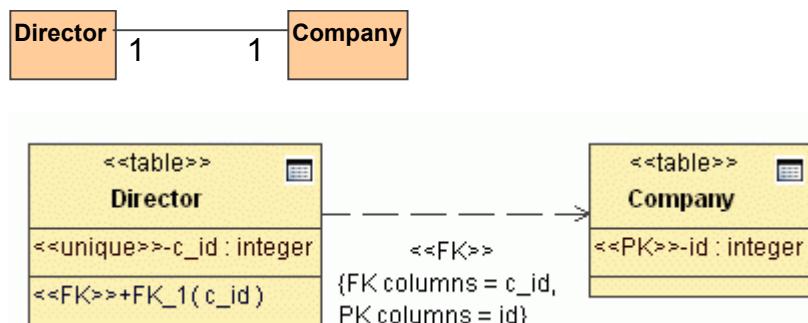


Figure 8 -- 1:1 relationship and mapped representation

A script that creates a 1:1 relationship:

```

CREATE TABLE Company (
    id INTEGER NOT NULL PRIMARY KEY
);
CREATE TABLE Director (
    c_id INTEGER NOT NULL UNIQUE,
  
```

```

CONSTRAINT FK_1 FOREIGN KEY (c_id) REFERENCES Company (id)
);

```

Nevertheless, this DDL script with these constraints does not ensure a strict 1:1 relationship – Company table may have rows that do not have their counterpart rows within Director table.

Many-to-many (N:M) relationship

Many-to-many (N:M) associations are not handled by MagicDraw UML. The N:M relationship can be achieved by using two 1:N relationships with intermediate table.

Inheritance

Single inheritance

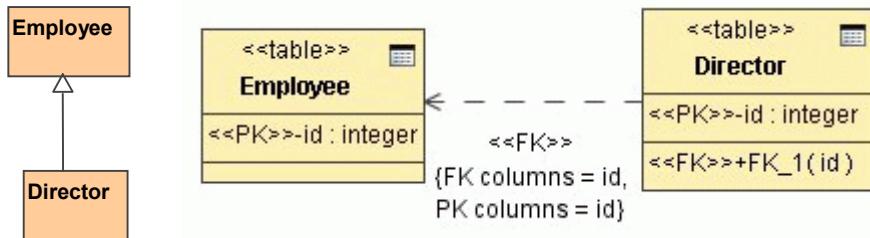


Figure 9 -- Single Inheritance and mapped representation

Single inheritance can be modeled with the 1:1 relationship, by creating the foreign key constraint on primary key in the derived class.

A script that creates a single inheritance:

```

CREATE TABLE Employee (
    id INTEGER NOT NULL PRIMARY KEY
);
CREATE TABLE Director (
    id INTEGER NOT NULL PRIMARY KEY,
    CONSTRAINT FK_1 FOREIGN KEY (id) REFERENCES Employee(id)
);

```

NOTE The Employee table may have rows that do not have their counterpart rows within Director table.

Multiple inheritance

The mapping of a multiple inheritance is similar to the mapping of a single inheritance.

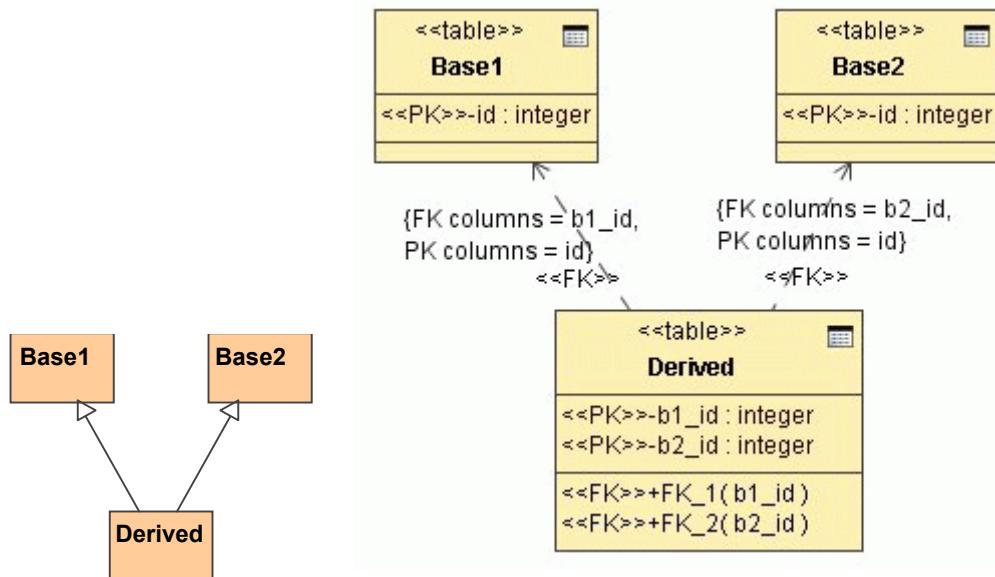


Figure 10 -- Multiple Inheritance and mapped representation

A script that creates a multiple inheritance:

```

CREATE TABLE Base1 (
    id INTEGER NOT NULL PRIMARY KEY
);
CREATE TABLE Base2 (
    id INTEGER NOT NULL PRIMARY KEY
);
CREATE TABLE Derived (
    b1_id INTEGER NOT NULL,
    b2_id INTEGER NOT NULL,
    PRIMARY KEY (b1_id, b2_id),
    CONSTRAINT FK_1 FOREIGN KEY (b1_id) REFERENCES Base1(id),
    CONSTRAINT FK_2 FOREIGN KEY (b2_id) REFERENCES Base2(id)
);

```

Not supported UML constructs

Constructs that are not mapped into DDL script, because this would lead to a generation of an illegal DDL code:

- Duplicated names are not allowed.

NOTE

Uppercase and lowercase letters are equivalent.

- Database package cannot contain two schema packages with the same name.
- Schema package cannot contain two UML constructs that are mapped to the schema elements such as table classes, view classes, index operations, and trigger operations that have the same name.
- Table class cannot have two column attributes or constraint operations with the same name.
- View class cannot have two column attributes with the same name.
- Table class cannot have two operations with the <<PK>> stereotype, because a table can have only one primary key (if any).
- References to non-existing columns are illegal.
 - The parameter name of an operation with a DDL stereotype (<<index>>, <<trigger>>, <<PK>>, <<unique>>, <<check>>) must be the name of an existing column attribute.
- Supported attribute multiplicity can be [not specified], [0..1], and [1]. All other attribute multiplicities are not supported.

Reverse engineering for DDL script

Information about a specific database structure acquired reversing DDL script or from JDBC is mapped to the MagicDraw UML constructs as described below.

Database

A Database is a system for a data storage and controlled access to the stored data. It is the biggest construct that a data model supports.

A package, which is used with the <<database>> stereotype, represents a database in the MagicDraw UML model. The database that is modeled as a package must have a name.

Example of a DDL script:

```
CREATE DATABASE BankDB;
```

Representation using UML concepts:

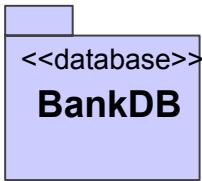


Figure 11 -- Database example

See also: CurrentDatabaseName property

Schema

A package with the <<schema>> stereotype within the package with the <<database>> stereotype represents a database schema.

Example of a DDL script:

```
CREATE SCHEMA Public;
```

Example rewritten using a qualified schema name “BankDB.Public”:

```
CREATE SCHEMA BankDB.Public;
```

Representation using UML concepts:

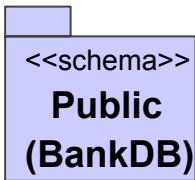


Figure 22 – Schema example

NOTE There can be more than one schema associated to a database.

See also: CurrentSchemaName property

Table

A table is the basic modeling structure of a relational database. It represents a set of records of the same structure, also called rows. Each of these records contains data. Information about the structure of a table is stored in the database itself.

A class with the <<table>> stereotype represents a relational table in a schema of a database.

Example of a DDL script for table:

```
CREATE TABLE Account (
    accountNo INTEGER NOT NULL,
    personId INTEGER NOT NULL,
    balance FLOAT(5) DEFAULT 0.0 NOT NULL
);
```

This example may be rewritten instead of “Account” using qualified table name “BankDB.Public.Account”.

Representation using UML concepts:

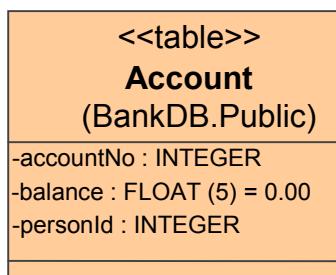


Figure 12 -- Table example

Hosting the table in the schema package creates an association of a table to a schema.

Column

A table contains columns. A column must have a defined name and data type; a default value and several constraints are optional.

Example: balance FLOAT(5) DEFAULT 0.0

A column is represented as an attribute. A name, data type, and initial value of an attribute are set according to the name, data type, and default value of the column.

The stereotype of an attribute is set according to column constraints. Stereotype can be not specified, <<unique>>, or <<PK>>.

Constraint

A constraint is a rule applied to the structure of a database. This rule extends the structure of a database and can be applied to a column or a table.

In general, a constraint may be represented as an operation with an appropriate stereotype and parameter list containing the list of column names that constraint concerns. An operation name is equal to the name of a constraint, but, when a constraint has no name specified, operation is unnamed.

All listed constraints (null, not null, uniqueness, primary key, foreign key, and check) are implemented in the following example:

```
CREATE TABLE Person (
    id INTEGER NOT NULL PRIMARY KEY,
    socialId NUMBER(10) NOT NULL UNIQUE
    CONSTRAINT checkSocialId CHECK(socialId>0),
    lastName VARCHAR(20) NOT NULL,
    firstName VARCHAR(10) NOT NULL,
    sex CHAR(1) NULL
);

CREATE TABLE Account (
    accountNo INTEGER NOT NULL,
    balance FLOAT(5) DEFAULT 0.0 NOT NULL
    CONSTRAINT checkBalance CHECK(balance>=0),
    personalId INTEGER NOT NULL,
    CONSTRAINT PK_Account0 PRIMARY KEY (accountNo, personalId),
    CONSTRAINT FK_Account1 FOREIGN KEY (personalId) REFERENCES Person(id)
);
```

Representation using UML concepts:

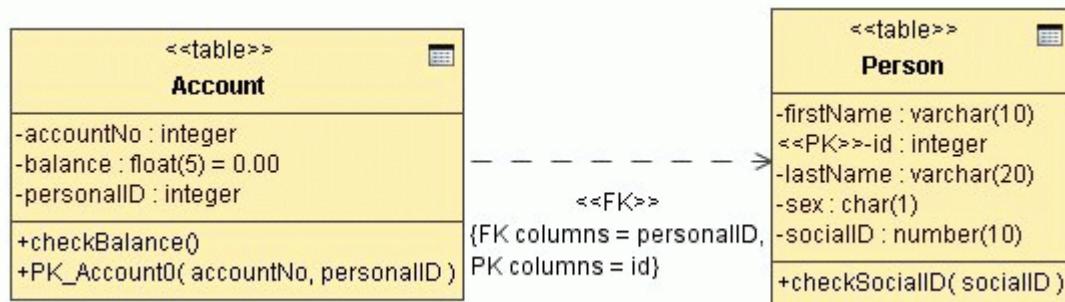


Figure 13 -- Constraints example

Not null, Null

NOTE A Null constraint is not defined in SQL-92 standard but some dialects use it. Microsoft SQL Server allows setting a flag, which indicates that all columns have Not Null constraint, which is set by default. A Null constraint that is assigned for a column overrides the default Not Null constraint and allows null values.

Not Null constraint indicates that the value for an attribute is required; Null constraint indicates that the value is optional.

Example:

```
firstName VARCHAR(10) NOT NULL,  
sex CHAR(1) NULL
```

Not null constraint and null constraint are modeled as the multiplicity of an attribute.

Multiplicity may be indicated by placing a multiplicity indicator in brackets after the name of an attribute. A multiplicity of 0..1 provides a possibility of null values, for an example: sex [0..1]: CHAR(1)

In the absence of a multiplicity indicator, an attribute holds exactly one value.

See also: ColumnDefaultNullability, AttributeDefaultMultiplicity, GenerateNullConstraint, GenerateNotNullConstraint properties.

Example of a DDL script:

```
CREATE TABLE NullableExample (  
    dontcare INTEGER,  
    optional INTEGER NULL,  
    required INTEGER NOT NULL  
) ;
```

Representation using UML concepts:

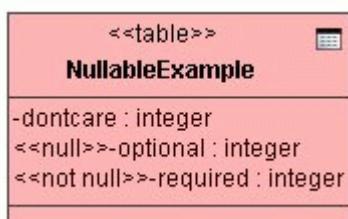


Figure 14 -- Not null, null constraints example

Uniqueness

Uniqueness constraint indicates that the value of a column must be unique within the table.

Uniqueness constraint is modeled as the <<unique>> stereotype applied to an attribute and/or operation with the <<unique>> stereotype and a parameter list.

Example of a DDL script:

```
CREATE TABLE UniqueExample (
    col1 INTEGER CONSTRAINT uniqueColumn UNIQUE,
    col2 INTEGER,
    col3 INTEGER,
    CONSTRAINT uniqueCombination UNIQUE (col2, col3)
);
```

Representation using UML concepts:

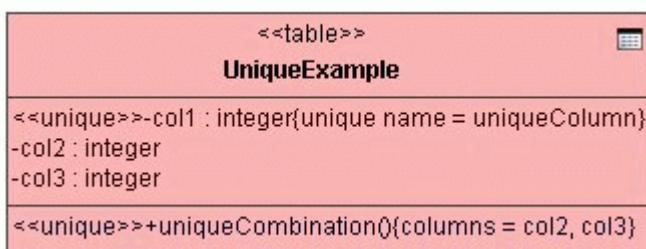


Figure 15 -- Unique constraint example

Primary Key

Primary Keys uniquely identify a row in a table. They mark an attribute as the Primary Key or the part of the Primary Key. The attribute must be of a scalar type. If more than one Primary Key attribute is identified, a concatenated primary key is generated.

A Primary Key is represented as the <<PK>> stereotype on an attribute and/or an operation with the <<PK>> stereotype and the parameter list.

Example of a DDL script:

```
CREATE TABLE PKColumnExample (
    col1 INTEGER Constraint pkColumn PRIMARY KEY
);
CREATE TABLE PKColumnExample (
    col2 INTEGER,
    col3 INTEGER,
    CONSTRAINT pkCombination PRIMARY KEY (col2, col3)
```

) ;

Representation using UML concepts:

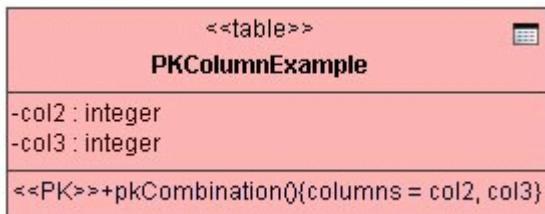


Figure 16 -- Primary key constraint examples

Foreign Key

A Foreign Key constraint represents a relationship to another table.

Foreign Key constraint is represented as a dependency with the <<FK>> stereotype to the target table.

CONSTRAINT <fkname> FOREIGN KEY (<linkCols>) REFERENCES <targetTable>(<targetCols>)

UML concept	FK element	Example
Dependency name	<fkname>	fk
Dependency source class	<linkTable>	FKLink
"FK columns" tagged value on the dependency	<linkCols>	I1,I2
Dependency target class	<targetTable>	FKTarget
"PK columns" tagged value on the dependency	<targetCols>	t1, t2

Example of a DDL script:

```
CREATE TABLE FKTTarget1 (
  t INTEGER PRIMARY KEY
);
CREATE TABLE FKLink1 (
  r INTEGER CONSTRAINT fk1 REFERENCES FKTTarget1(t),
);

CREATE TABLE FKTTarget (
  t1 INTEGER,
  t2 INTEGER,
  PRIMARY KEY (t1, t2)
```

```
);
CREATE TABLE FKLink (
    r1 INTEGER,
    r2 INTEGER,
CONSTRAINT fk2 FOREIGN KEY (r1,r2) REFERENCES FKTTarget(t1,t2)
);
```

Representation using UML concepts:

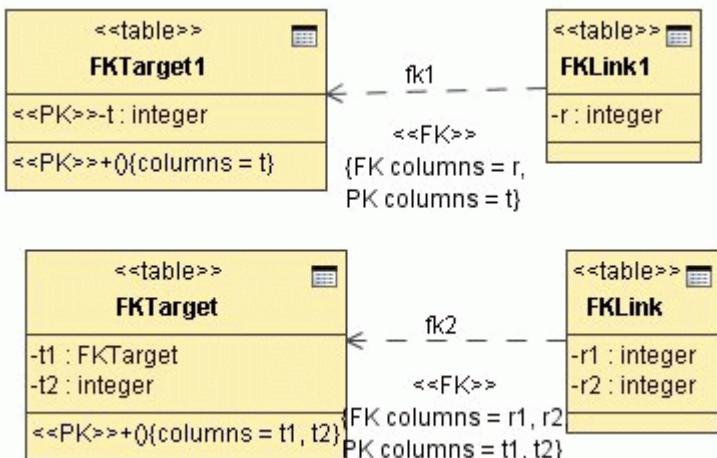


Figure 17 -- Foreign key constraint example

Check

The Check constraint checks the value of data according to a given expression.

The Check constraint is represented as an operation with the `<<check>>` stereotype. Operation's name is equal to the name of the check constraint or, if the constraint name is not specified, the name "unnamed" is generated.

There are a column check constraint and table check constraint:

For column check constraint operation's parameter list contains one parameter with the name that equals to the name of the attribute for which check constraint is assigned.

Example: Table check constraint operation has no parameters.

```
CREATE TABLE CheckExample (
balance INTEGER CONSTRAINT checkBalance CHECK(balance>=0)
start INTEGER, -- period start balance
income INTEGER CHECK(income>=0), -- unnamed check constraint
outcome INTEGER,
```

```
CONSTRAINT checkInOut CHECK(start+income-outcome = balance)
);
```

Representation using UML concepts:

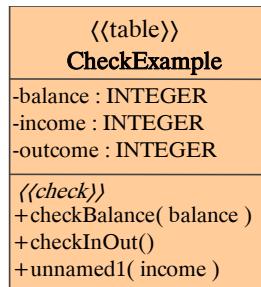


Figure 18 -- Check constraint example

Unnamed constraint representation as a stereotype of an attribute

Some column characteristics (column primary key constraint, column uniqueness constraint) that apply to one specific column may be represented as stereotype of that attribute.

Example DDL script:

```
CREATE TABLE ConstraintExample1 (
  col1 INTEGER PRIMARY KEY,
  col2 INTEGER UNIQUE
);
```

Representation using UML concepts:

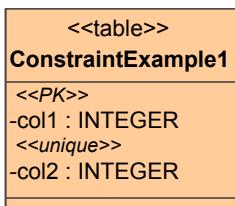


Figure 19 -- Column with unnamed constraint example

Only one constraint can be represented as stereotype, because only one stereotype can be assigned for the attribute. If there are other constraints, they should be represented as operations.

Although attribute may have several such constraints assigned, in practice there is no need for more than one. If attribute has primary key constraint it is unique too.

Example DDL script:

```
-- here unique constraint is unnecessary
CREATE TABLE ConstraintExample2 (
  col1 INTEGER PRIMARY KEY UNIQUE
);
```

Representation using UML concepts:

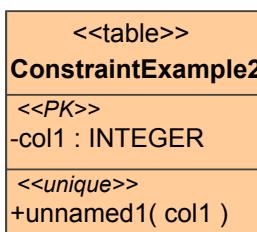


Figure 20 -- Column with two unnamed constraints example

Index

An Index is a physical data structure that speeds up the access to data. It does not change the quality or the quantity of data retrieved. The index specifies the columns included and optionally the uniqueness of the index. An index can include multiple columns or just a single column.

An index is represented as an operation with the <<index>> stereotype and parameter list. The operation name is equal to the index name and the parameter list of the operation is a column names for the index. The uniqueness of the index is ignored (not mapped).

DDL script for an index example that creates the Person table and two indexes:

```

CREATE TABLE Person (
  id INTEGER NOT NULL PRIMARY KEY,
  socialId NUMBER(10) NOT NULL UNIQUE,
  lastName VARCHAR(20) NOT NULL,
  firstName VARCHAR(10) NOT NULL,
  sex CHAR(1)
);
CREATE UNIQUE INDEX bySocialId ON Person(socialId);
CREATE INDEX byName ON Person(lastName,firstName);
```

Representation using UML concepts:

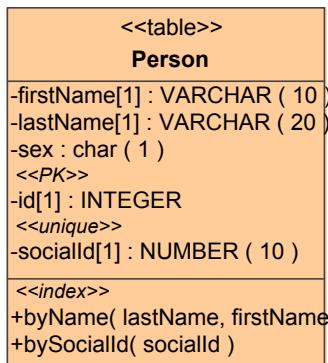


Figure 21 -- Index example

Trigger

A trigger is an activity executed by the DBMS as a side effect or instead of a modification of a table or view to ensure consistent system behavior on data operations.

A Trigger is represented as an operation with the <<trigger>> stereotype.

DDL script for trigger example that creates trigger:

```

CREATE TRIGGER logActions BEFORE INSERT OR DELETE OR UPDATE
  ON Person
  <triggered SQL statement>;
  
```

Representation using UML concepts:

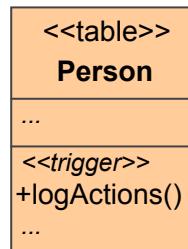


Figure 22 -- Trigger example

View

A View is a construct for creating a virtual table based on one or more existing tables or views.

A class with the <>view>> stereotype represents a view in a schema of a database. The View column is modeled as an attribute. Relationships between the View and its underlying tables ("FROM" clause) are modeled as dependencies with <>reference>> stereotype. Referenced table alias is modeled as the name of the dependency, referenced column list as the "columns" tagged value on the dependency

View element	UML concept	Script example
View	Class with the <>view>> stereotype	CREATE VIEW
View name	Class name	
Column	Attribute	
Column name	Attribute name	
Derived column	Attribute	
- <expression> AS <name>	default value <expression>, name <name>	Account.balance AS total
- *	Attribute name	*
-<tablename>.*	Attribute name	P.*
-<columnname>	Attribute name	Balance
- <tablename>.<columnname>	Attribute name <columnname> and default value <tablename>.<columnname>	Account.balance

View element	UML concept	Script example
-<expression>	Generated unique attribute name and default value <expression>	
Table reference	Dependency	
Referenced table alias	Dependency name	P
Referenced column list	"columns" tagged value on the dependency	Id,firstName,lastName

DDL script for view example:

```

CREATE TABLE Person (
    id INTEGER NOT NULL,
    socialId NUMBER(10) NOT NULL,
    lastName VARCHAR(20) NOT NULL,
    firstName VARCHAR(10) NOT NULL,
    sex CHAR(1)
);
CREATE TABLE Account (
    accountNo INTEGER NOT NULL,
    balance FLOAT(5) DEFAULT 0.0 NOT NULL,
    personalId INTEGER NOT NULL
);
CREATE VIEW ImportantClient
AS SELECT P.*, Account.balance as total
FROM Person as P(id,firstName,lastName), Account
WHERE balance >= 1000000.00 AND personId = P.id;

```

Representation using UML concepts:

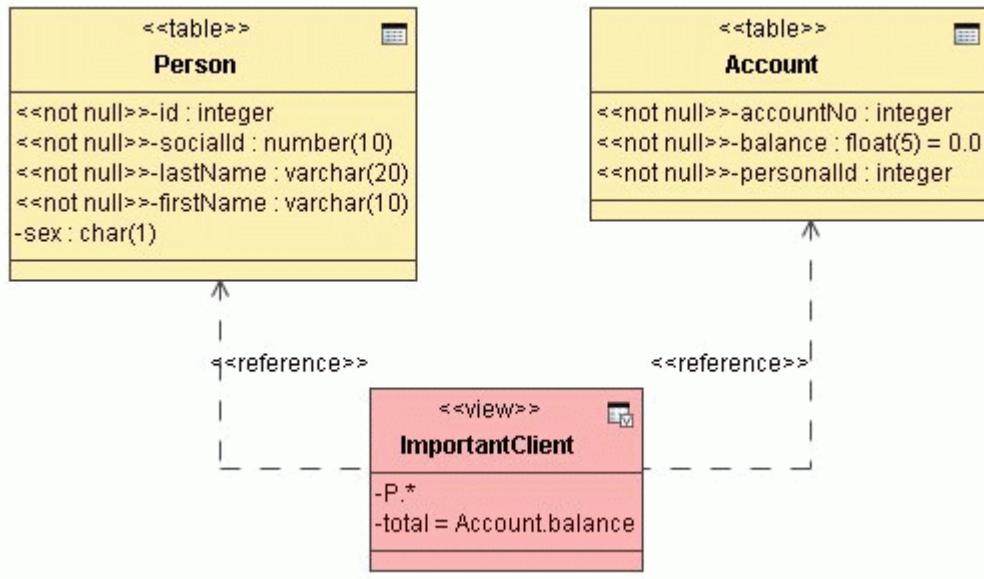


Figure 23 -- View example

DDL dialects

This section reviews DDL implementations from different vendors. Specific implementation usually states compliance to some level of SQL standard and provides some extensions.

Standard SQL2

For SQL2 statements supported by MagicDraw UML see Section Supported SQL statements, “Supported SQL statements”, on page 438.

MagicDraw UML schema package is located within a database package. Database definition statement is not the part of the SQL2 standard - it is an analogue of a Database (a Catalog).

NOTE A Catalog has no explicit definition statement. If a database package for a Catalog does not exist, it should be created (when it is referred for the first time).

Cloudscape

Informix Cloudscape v3.5 dialect has no database definitions statement. A database package with the name specified by CurrentDatabaseName property is used.

This dialect has CREATE INDEX and CREATE TRIGGER statements that are not the part of a SQL2 standard but that should be taken into account while reversing DDL script of this dialect.

This dialect has some syntax differences from SQL2 standard because of extensions (e.g. some schema definition statements can have PROPERTIES clause). These extensions are ignored while reversing.

Oracle Oracle8

Oracle Oracle8 dialect has CREATE DATABASE, CREATE INDEX, and CREATE TRIGGER statements that are not the part of SQL2 standard but that should be taken into account while reversing DDL script of this dialect.

This dialect has some syntax differences from SQL2 standard because of extensions (e.g. some schema definition statements can have STORAGE clause). These extensions are ignored while reversing.

Oracle Oracle8 has object oriented DDL statements (CREATE TYPE and CREATE TYPE BODY). Additional object oriented Oracle8 schema objects are Object Types, Nested Object Types, Nested Table, VARRAY, Object Tables, and Object Views.

Stereotypes for MagicDraw constructs

The following table lists stereotypes that are used with MagicDraw UML constructs to represent a database structure:

Model Item	Stereotype	Description	Default item stereotype for forward engineering
Package	<<database>>	See “Database” on page 418	If EnableDefaultStereotypes property is true, <<database>> stereotype is used for first level packages, and <<schema>> stereotype is used for second level packages;
	<<schema>>	See “Schema” on page 419	Otherwise none.

Model Item	Stereotype	Description	Default item stereotype for forward engineering
Class	<<table>> <<view>>	See “Table” on page 419 See “View” on page 429	If EnableDefaultStereotypes property is true, <<table>> stereotype is used; Otherwise none.
Attribute	<<PK>> <<unique>>	See “Primary Key” on page 423 See “Uniqueness” on page 423	None
Operation	<<PK>> <<unique>> <<check>> <<index>> <<trigger>>	See “Primary Key” on page 423 See “Operations” on page 412 See “Check” on page 425 See “Index” on page 427 See “Trigger” on page 428	None
Dependency	<FK>> <<reference>>	See “Relationship cardinalities” on page 414 See “Relationship cardinalities” on page 414	

Properties of code engineering set for DDL

There are two separate properties sets, stored as properties of code engineering set for DDL:

- Properties for DDL script generation,
- Properties for DDL script reverse engineering.

Properties for DDL script reverse engineering and generation



Figure 24 -- CG Properties Editor dialog box. DDL properties

Property name	Values list	Description
<hr/>		
Reverse engineering features		
<hr/>		

Property name	Values list	Description
Column default nullability	Dialect default (default), not specified , NULL , NOT NULL	If column has no NULL or NOT NULL constraint specified, the value of this property is used.
Create catalog sets current catalog	True (default), false	Specifies whether create catalog statement changes current catalog name.
Create schema sets current schema	True (default), false	Specifies whether create schema statement changes current schema name.
Default catalog name	DefaultCatalogNone (default), DefaultCatalogPackage , any entered by the user	Specifies current database name. Used when DDL script does not specify database name explicitly.
Default schema name	DefaultSchemaNone (default), DefaultSchema-Package , any entered by the user	Specifies current schema name. Used when DDL script does not specify schema name explicitly.
Drop statements	Deferred (default), Immediate , Ignored	Specifies whether execution of drop statements may be deferred, or must be executed, or must be ignored. Deferred drop may be enabled if elements are recreated later. This will save existing views. Attribute stereotypes, multiplicity and default value always are not dropped immediately.
Map Null/not Null constraints to	Stereotypes (default), Multiplicity	When parsing DDLs, the null/not null constraints are modeled as either stereotypes or multiplicity.
Map foreign keys	True (default), false	A dependency with <<FK>> stereotype is created, to represent Foreign Key.
Map indexes	True (default), false	An operation with <<index>> stereotype is added into class, to represent index.

Property name	Values list	Description
Map triggers	True (default), false	An operation with <>trigger<> stereotype is added into class to represent trigger.
Map views	True (default), false	A class with <>view<> stereotype is created to represent view.
Generation features		
Default attribute multiplicity	0 , 0..1 , any entered by user	If the attribute multiplicity is not specified, the value of this property is used.
Generate Null constraint	True , false (default)	If true, generates NULL constraint for column attribute with [0..1] multiplicity. If DBMS, you use, support NULL, you can enable this to generate NULL constraints. See also: GenerateNotNullConstraint, AttributeDefaultMultiplicity
Generate extended index name	True , false (default)	If true, generates index name of the form: TableName_IndexName.
Generate extended trigger name	True , false (default)	If true, generates trigger name of the form: TableName_TriggerName.
Generate index for primary key	True (default), false	If the DBMS, you use, requires explicit indexes for primary key, you may enable explicit index creation using this flag. See also: GenerateIndexForUnique
Generate index for unique	True (default), false	If the DBMS, you use, requires explicit indexes for primary key or unique columns, may enable explicit index creation using this flag. See also: GenerateIndexForPK

Generate not Null constraint	True (default), false	If true, generates NOT NULL constraint for column attribute with [1] multiplicity. If you set GenerateNullConstraint, you may wish to do not generate NOT NULL constrain. See also: GenerateNullConstraint, AttributeDefaultMultiplicity
Generate qualified names	True (default), false	If value of Generate Qualified Names check box is true, package name is generated before the table or view name. For example: <<database>> package MQOnline includes <<table>> class libraries. Then check box Generate Qualified Names is selected as true in generated source would be written: DROP TABLE MQOnline.libraries; Then check box Generate Qualified Names is selected as false, in generated source would be written: DROP TABLE libraries;
Generate quoted identifiers	True, false (default)	Specifies whether DDL code generator should generate quoted names of identifiers.
Object creation mode	The Object Creation Mode combo box has the following options: only CREATE statements DROP & CREATE statements CREATE OR REPLACE statements (only for Oracle dialect; default for this dialect) DROP IF EXISTS & CREATE statements (only for MySQL dialect; default for this dialect).	

Supported SQL statements

This section lists SQL statements that MagicDraw UML supports (that are parsed and mapped into UML constructs).

The following table provides SQL2 SQL schema statements that are supported and that are NOT supported in MagicDrawTM UML:

SQL schema statement	Supported	(Yes/No)
SQL schema definition statement	Schema definition	Yes
	Table definition	Yes
	View definition	Yes
	Alter table statement	Yes
	Grant statement	No
	Domain definition	No
	Assertion definition	No
	Character set definition	No
	Collation definition	No
	Translation definition	No
SQL schema manipulation statement	Drop table statement	Yes
	Drop view statement	Yes
	Revoke statement	No
	Alter domain statement	No
	Drop assertion statement	No
	Drop domain statement	No
	Drop character set statement	No
	Drop collation statement	No
	Drop translation statement	No

Some SQL schema statements (e.g. schema definition, table definition) allow implicit catalog name and unqualified schema name. In addition to SQL schema statements, the following SQL session statements must be supported:

- Set catalog statement - sets the current default catalog name.

- Set schema statement - sets the current default unqualified schema name.

MagicDraw supports the following widely used by dialects statements that are not the part of SQL2:

- Database definition statement (CREATE DATABASE) that creates database
- Index statements (CREATE INDEX, DROP INDEX) that create an index on table and remove it
- Trigger statements (CREATE TRIGGER, DROP TRIGGER) that create a trigger on table and remove it.

The following table provides details on mapping on the supported SQL schema manipulation statements into MagicDraw constructs:

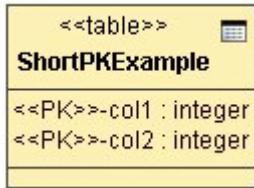
DDL Statement or Concept	Action, model Item	Description	Visible
Alter table statement	Modify class	Elements: table name and alter table action. Alter table action – one of: add column, add table constraint, alter column, drop table constraint, drop column.	Yes
Add column definition	Define attribute	Elements: column definition.	Yes
Add table constraint definition	Define method	Elements: table constraint definition.	Yes
Alter column definition	Modify attribute	Elements: mandatory column name, default clause (for add default statement only).	Yes
Drop table constraint definition	Delete method	Elements: constraint name, drop behavior	Yes
Drop column definition	Delete attribute	Elements: column name, drop behavior	Yes
Drop schema statement	Delete package	Elements: schema name, drop behavior	Yes
Drop table statement	Delete class	Elements: table name, drop behavior	Yes
Drop view statement	Delete class	Elements: table name, drop behavior	Yes
Drop behavior	Action property	Modifiers: CASCADE, RESTRICT	No

Tips

Short representation for primary key constraint

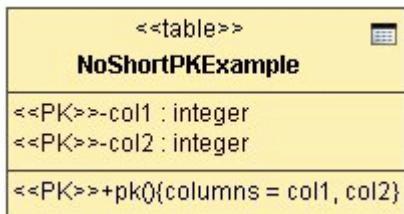
When primary key is made out of several columns, and columns order is not important, short primary key representation may be used. Columns that make up the primary key must be marked with <>PK>> stereotype. Because a table can contain only one (if any) primary key, these columns are concatenated into one primary key.

Short primary key representation using UML concepts and generated DDL script:



```
CREATE TABLE ShortPKExample (
    col1 INTEGER,
    col2 INTEGER,
    PRIMARY KEY (col1, col2)
);
```

If the order of columns within the primary key is important, constraint representation as an operation with parameters must be used:



```
CREATE TABLE NoShortPKExample (
    col1 INTEGER,
    col2 INTEGER,
    PRIMARY KEY (col2, col1)
);
```

Primary key constraint with overhead info

Consider such primary key definition using UML concepts and generated DDL script:

<<table>>
PKWithOverheadExample
-col2 : INTEGER <i><<unique>></i> -col1 : INTEGER
<<PK>> +pk(col1, col2)

```
CREATE TABLE PKWithOverheadExample (
    col1 INTEGER UNIQUE,
    col2 INTEGER,
    PRIMARY KEY (col1, col2)
);
```

This representation is valid, but the primary key contains overhead info (specifically column col2). Unique column alone may be used as a valid primary key:

<<table>>
PKWithoutOverheadExample
-col2 : INTEGER <<PK>> -col1 : INTEGER

```
CREATE TABLE PKWithoutOverheadExample (
    col1 INTEGER PRIMARY KEY,
    col2 INTEGER
);
```

CORBA IDL MAPPING TO UML

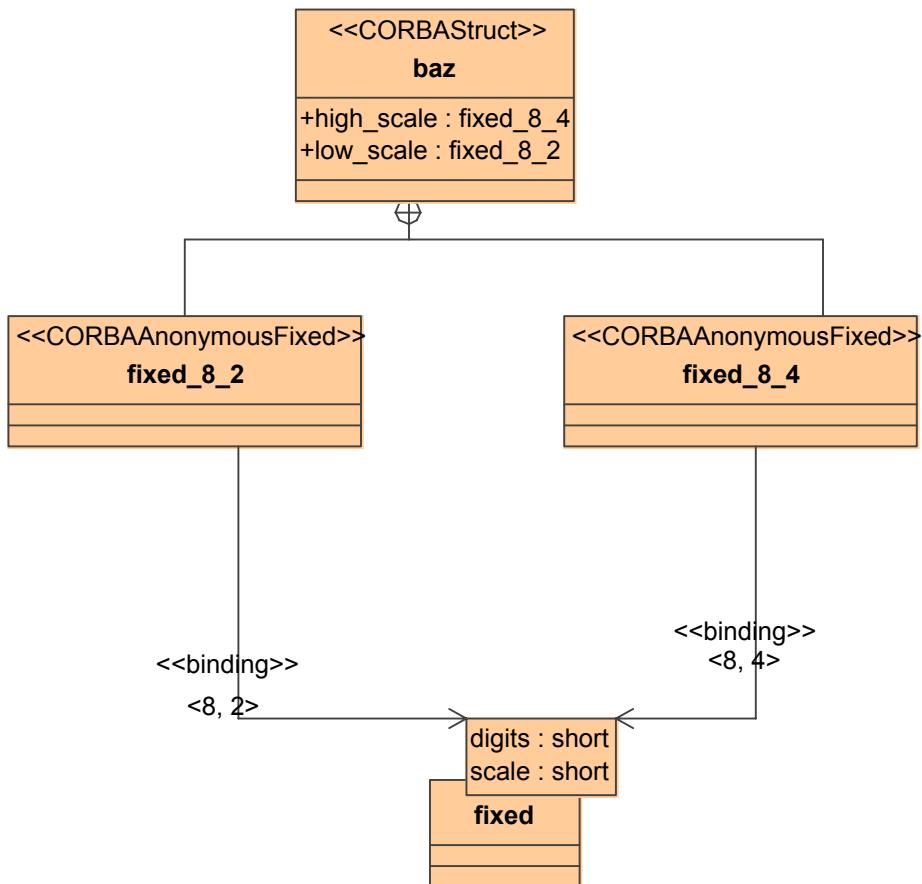
CORBA IDL mapping to UML is based on UML™ Profile for CORBA™ Specification, Version 1.0, April 2002. http://www.omg.org/technology/documents/formal/profile_corba.htm.

Differences between UML Profile for CORBA specification and mapping in MagicDraw is listed below. Most differences are present because MagicDraw does not fully support UML 1.4.

- CORBA IDL exception is mapped to Core::Class, instead of CommonBehavior::Exception. MagicDraw does not support CommonBehavior::Exception.
- Constraints defined in UML Profile for CORBA specification are not checked in MagicDraw.
- For storing constants, UML class instead of UtilityClass is used.
- Use simplified “1” multiplicity instead of “1...1”.
- 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 binded with 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	CORBA IDL Module

CORBA IDL MAPPING TO UML

Button	Shortcut key	Model Element
	I	CORBA IDL Interface
	V	CORBA IDL Value
	G	Generalization
		Truncatable Generalization
		Value Support Generalization
		CORBA IDL Association
	I	Interface

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.e., 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»	<p>Specializes the standard UML Stereotype «file». Indicates that the Artifact describes a Java Source File.</p> <p>NOTE The compiled “*.class” file is not modeled by this profile.</p>

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.
JavaCollection	Attribute or AssociationEnd	A String containing the name of the Java Collection Type used to implement an Attribute or Association End with complex multiplicity.
JavaNative	Operation or Method	A Boolean value indicating whether or not the Operation or Method is native.
JavaThrows	Operation or Method	A comma-delimited list of names of Java Exception Classes.

JavaFinal	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.
«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».

«EJBRM	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.
«EJRNR	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.
«EJRL	Dependency	Indicates a Dependency between a client EJB Role Name Reference (Design Model) and a supplier EJB Role Name (Deployment Model).
«EJSB	Class	Indicates that the Class represents an EJB Session Bean. Specializes «EJBEnterpriseBean».
«EJS	Operation	An Operation that is a select method in an EJB Entity Bean. (EJB 2.0 only)
«EJSRR	Association	Indicates a Dependency between an EJB client, and an EJB Role Name Reference supplier.

Tagged Values

Tagged Value	Applies To	Definition
EJBAbstractSchemaName	Class «EJBEntityBean»	A String representing the abstract name used for a schema associated with a CMP 2.0 entity bean.
EJBAcknowledgeMode	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.
EJBComponentInterface	Dependency «EJBReference» or «EJBLocalReference»	The name of the EJB Enterprise Bean's "Component" Interface (local or remote).
EJBCmpVersion	Class «EJBEntityBean»	An enumeration with values 1.x or 2.x. Indicates the type of CMP used by the EJB Entity Bean
EJBCmrFieldType	AssociationEnd «EJBRelationshipRole»	Type expression property of the supplier UML Association End «EJBRelationshipRole».
EJBDisplayName	Class «EJBEnterpriseBean»	A String with the name for the EJB to be displayed by tools
EJBEnvEntries	Class «EJBEnterpriseBean»	A string of XML tags, designating the environment entries used by the EJB Enterprise Bean.
EJBHomeInterface	Dependency «EJBReference» or «EJBLocalReference»	The name of the EJB Enterprise Bean's Home Interface (local or remote).
EJBLink	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.

EJBMessageDrivenDestination	Class «EJBMessageDriven-Bean»	An XML tag, designating the type of destination and the durability used by the EJB Message Driven Bean.
EJBMessageSelector	Class «EJBMessageDriven-Bean»	A String specifying the JMS message selector to be used in determining which messages a message-driven bean is to receive.
EJBNameInJAR	Class «EJBEnterpriseBean»	The name used for the EJB Enterprise Bean in the EJB-JAR. Defaults to the name of the EJB Remote Interface.
EJBPersistenceType	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.
EJBQueryString	Operation «EJBFinder» or «EJBSelect»	The EJB QL statement corresponding to the method. It is ignored for BMP Entity Beans
EJBReentrant	Class «EJBEntityBean»	A Boolean value indicating whether or not the EJB Entity Bean can be called re-entrantly.
EJBRefName	Dependency «EJBReference» or «EJBLocalReference»	The name of the EJB Enterprise Bean referenced.
EJBRefType	Dependency «EJBReference» or «EJBLocalReference»	The type of referenced EJB Enterprise Bean, one of the strings Entity or Session.
EJBRelationshipRoleDescription	AssociationEnd «EJBRelationshipRole»	A String specifying a description an EJB relationship role.
EJBRelationshipRoleName	AssociationEnd «EJBRelationshipRole»	The name of an EJB relationship role.

EJBResources	Class «EJBEnterpriseBean»	A string of XML tags designating the resource factories used by the EJB Enterprise Bean.
EJBResourcesEnv	Class «EJBEnterpriseBean»	A string of XML tags designating the environment entries used by the EJB Enterprise Bean.
EJBResultTypeMapping	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.
EJBSessionType	«EJBSessionBean»	Stateful or Stateless. Indicates whether or not the EJB Session Bean maintains state.
EJBTransType	Class «EJBSessionBean» or Class «EJMMessageDrivenBean»	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.
EJBVersion	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
«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, etc.
«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
«EJBRequiresNew»	A new transaction is required for this method
«EJBSupports»	Transactions are supported for this method

Tagged Values

Tagged Value	Applies To	Definition
EJBDisplayName	Artifact «EJB-JAR»	A String with the name for the Archive to be displayed by tools
EJBMethodDescriptor	Dependency «EJB-Method»	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.
EJBUnchecked	Actor «EJBMethodPermission»	A Boolean that when “true”, indicates that permission to invoke the specified methods should not be checked before invocation. Default is “false”
EJBUseCallerIdentity	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.

XML SCHEMA

Reference: <http://www.w3.org/TR/xmlschema-2/>

XML Schema Mapping to UML Elements

Defined stereotypes

Stereotype name	Base Stereotype	Applies to	Defined TagDefinitions
XSDcomponent		Class Attribute AssociationEnd Binding Generalization Comment Component	id – string Details: The base and abstract stereotype for all XML Schema stereotypes used in UML profile
XSDattribute	XSDcomponent	Attribute	fixed – some fixed element value form - (<i>qualified</i> <i>unqualified</i>) refString – string representation of reference to other attribute. ref – actual reference to other attribute use - (<i>optional</i> <i>prohibited</i> <i>required</i>) : optional

XSDelement	XSDcomponent	Attribute AssociationEnd	<p>abstract – (true false)</p> <p>block - (extension restriction substitution)</p> <p>final - (extension restriction)</p> <p>fixed – some fixed element value</p> <p>form - (<i>qualified</i> <i>unqualified</i>)</p> <p>nillable – (true false)</p> <p>refString – string representation of reference to other attribute.</p> <p>ref – actual reference to other attribute</p> <p>substitutionGroup – actual reference to UML ModelElement</p> <p>substitutionGroupString – string representation of substitution group</p> <p>key_unique_keyRef – a list of referenced UML Attributes</p> <p>sequenceOrder – a number in sequence order</p>
XSDcomplexType	XSDcomponent	Class	<p>block – (<i>extension</i> <i>restriction</i>)</p> <p>final – (<i>extension</i> <i>restriction</i>)</p> <p>mixed – (true false)</p>
XSDsimpleContent		Class	simpleContentId – string
XSDcomplexContent		Class	complexContentId – string complexContentMixed
XSDgroup	XSDcomponent	Class	

XML SCHEMA

XML Schema Mapping to UML Elements

XSDgroupRef	XSDcomponent	Attribute AssociationEnd	sequenceOrder – a number in sequence order
XSDall		Class	allId – string maxOccurs minOccurs
XSDchoice		Class	choiceId – string maxOccurs minOccurs sequenceOrder – a number in sequence order
XSDsequence		Class	sequenceId – string maxOccurs minOccurs sequenceOrder – a number in sequence order
XSDrestriction	XSDcomponent	Generalization	
XSDextension	XSDcomponent	Generalization	
XSDattributeGroup	XSDcomponent	Class	
XSDsimpleType	XSDcomponent	Class	final - (#all (list union restriction))
XSDlist	XSDcomponent	Class	listId - string
XSDunion	XSDcomponent	Class	unionId - string
XSDannotation	XSDcomponent	Comment	appInfoSource appInfoContent source xml:lang
XSDany	XSDcomponent	Attribute	namespace – string processContents - (lax skip strict); default strict sequenceOrder – a number in sequence order

XSDanyAttribute	XSDcomponent	Attribute	namespace – string processContents - (<i>lax</i> <i>skip</i> <i>strict</i>); default strict
XSDschema	XSDcomponent	Class	attributeFormDefault blockDefault elementFormDefault finalDefault targetNamespace – reference to some ModelPackage version xml:lang
XSDnotation	XSDcomponent	Attribute	public system
XSD redefine	XSDcomponent	Class	
XSDimport	XSDcomponent	Permision <<import>>	schemaLocation
XSDinclude	XSDcomponent	Component	
XSDminExclusive	XSDcomponent	Attribute	fixed = boolean : false
XSDminInclusive	XSDcomponent	Attribute	fixed = boolean : false
XSDmaxExclusive	XSDcomponent	Attribute	fixed = boolean : false
XSDmaxInclusive	XSDcomponent	Attribute	fixed = boolean : false
XSDtotalDigits	XSDcomponent	Attribute	fixed = boolean : false
XSDfractionDigits	XSDcomponent	Attribute	fixed = boolean : false
XSDlength	XSDcomponent	Attribute	fixed = boolean : false
XSDminLength	XSDcomponent	Attribute	fixed = boolean : false
XSDmaxLength	XSDcomponent	Attribute	fixed = boolean : false
XSDwhiteSpace	XSDcomponent	Attribute	fixed = boolean : false value
XSDpattern	XSDcomponent	Attribute	
XSDenumeration	XSDcomponent	Attribute	

XSDunique		Attribute	selector field
XSDkey		Attribute	selector field
XSDkeyref		Attribute	selector field refer – UML Attribute referString - String
XSDnamespace		ModelPackage	
xmlns		Permission	

attribute

- XML schema attribute maps to UML Attribute with stereotype *XSDattribute*.
- *default* maps to initial UML Attribute or AssociationEnd value.
- *annotation* – to UML Attribute or AssociationEnd documentation.
- *name* – to UML Attribute or AssociationEnd name.
- *type* or content *simpleType* – to UML Attribute or AssociationEnd type.

Other attributes or elements maps to corresponding tagged values.

```
<attribute
    default = string
    fixed = string
    form = (qualified      | unqualified     )
    id = ID
    name = NCName
    ref = QName
    type = QName
    use = (optional     | prohibited    | required   ) ••• optional
{any attributes with non-schema namespace . . .} >
Content: (annotation? , (simpleType?))>
</attribute>
```

Example:

```
<xs:attribute name="age" type="xs:positiveInteger" use="required"/>
```

ref value is generated from ref or refString TaggedValue.

One of ref or name must be present, but not both.

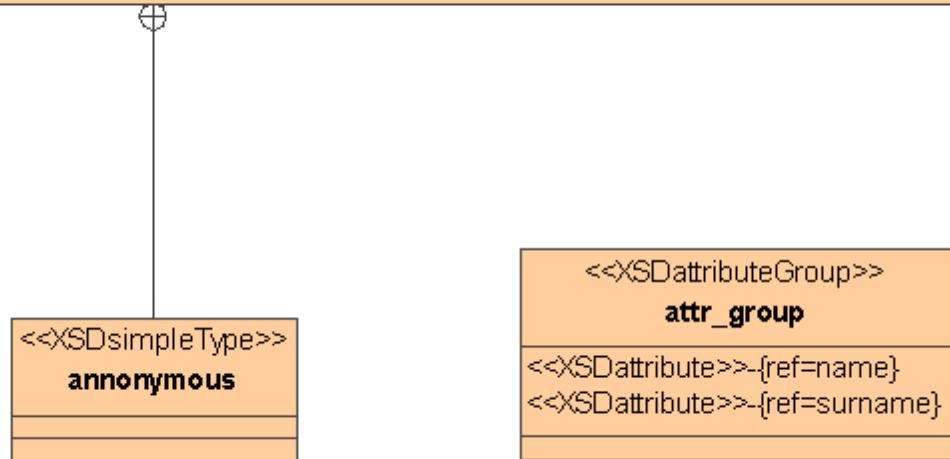
If ref is present, then all of [simpleType](#), form and type must be absent.

type and [simpleType](#) must not both be present.

attribute UML Model example:

```
<<XSDschema>>
schemaxsd
{targetNamespace=http://nomagic.com}

<<XSAttribute>>-address : anonymous{fixed=fixed_value, form=qualified, use=optional}
<<XSAttribute>>-name : string = min{fixed=fixed_value, form=qualified, use=optional}
<<XSAttribute>>-surname : string
```



```
<xsschema xmlns:nm = "http://nomagic.com" xmlns:xs = "http://www.w3.org/2001/XMLSchema" targetNamespace = "http://nomagic.com"
```

```
<xs:attribute name = "name" type = "xs:string" default = "minde"
fixed = "fixed_value" form = "qualified" use = "optional" >
    <xs:annotation>
        <xs:documentation>name attribute
documentation</xs:documentation>
    </xs:annotation>
</xs:attribute>
<xs:attribute name = "address" fixed = "fixed_value" form =
"qualified" use = "optional" >
    <xs:annotation>
        <xs:documentation>surname attribute
documentation</xs:documentation>
    </xs:annotation>
<xs:simpleType>
    <xs:restriction base = "xs:string" />
</xs:simpleType>
</xs:attribute>
<xs:attribute name = "surname" type = "xs:string" />
<xs:attributeGroup name = "attr_group" >
    <xs:attribute ref = "nm:name" >
        <xs:annotation>
            <xs:documentation>reference
documentation</xs:documentation>
        </xs:annotation>
    </xs:attribute>
    <xs:attribute ref = "nm:surname" />
</xs:attributeGroup>
</xs:schema>
```

element

Maps to UML Attribute or UML AssociationEnd with stereotype *XSDelement*.

- annotation – to UML Attribute or UML AssociationEnd documentation.
- default - to initial UML Attribute or UML AssociationEnd value.
- maxOccurs - to multiplicity upper range. Value unbounded maps to asterisk in UML.
- minOccurs – to multiplicity lower range.
- name – to UML Attribute or UML AssociationEnd name.

- type or content (simpleType | complexType) – to UML Attribute or UML AssociationEnd type.

Other properties maps to corresponding tagged values.

XML Representation Summary: element Element Information Item

```
<element
    abstract = boolean : false
    block = (#all | List of (extension | restriction | substitution))
    default = string
    final = (#all | List of (extension | restriction))
    fixed = string
    form = (qualified | unqualified)
    id = ID
    maxOccurs = (nonNegativeInteger | unbounded) : 1
    minOccurs = nonNegativeInteger : 1
    name = NCName
    nillable = boolean : false
    ref = QName
    substitutionGroup = QName
    type = QName
    {any attributes with non-schema namespace . . .}>
  Content: (annotation? , ((simpleType | complexType)? , (unique | key | keyr
</element>
```

ref value is generated from ref or refString TaggedValue.

One of ref or name must be present, but not both.

If ref is present, then all of complexType, simpleType, key, keyref, unique, nillable, default, fixed, form, block and type must be absent, i.e. only minOccurs, maxOccurs, id are allowed in addition to ref, along with annotation.

Example:

```

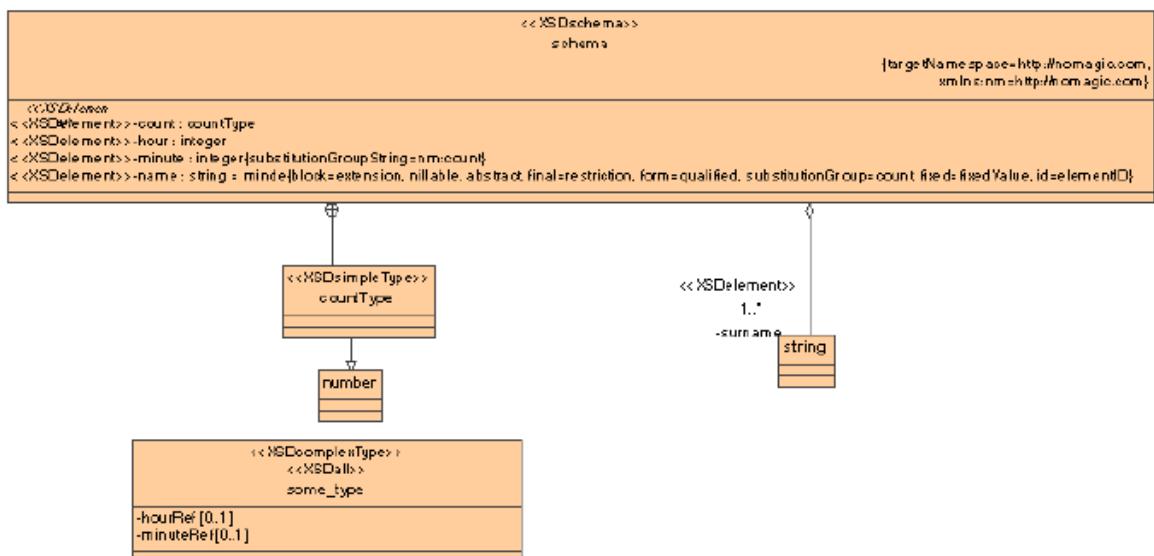
<xs:element name="PurchaseOrder" type="PurchaseOrderType" />

<xs:element name="gift">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="birthday" type="xs:date"/>
      <xs:element ref="PurchaseOrder" />

    </xs:sequence>
  </xs:complexType>
</xs:element>

```

element UML Model example:



```

<xs:schema xmlns:nm = "http://nomagic.com" xmlns:xs =
"http://www.w3.org/2001/XMLSchema" targetNamespace = "http://nomagic.com" >
    <xs:element name = "name" type = "xs:string" default = "minde" id = "elementID"
abstract = "true" block = "extension" final = "restriction" fixed = "fixedValue" form =
"qualified" nillable = "true" substitutionGroup = "nm:count" >
        <xs:annotation >
            <xs:documentation >element name documentation</xs:documentation>
        </xs:annotation>
    </xs:element>
    <xs:element name = "count" >
        <xs:annotation >
            <xs:documentation >element count documenation</xs:documentation>
        </xs:annotation>
        <xs:simpleType >
            <xs:restriction base = "xs:number" />
        </xs:simpleType>
    </xs:element>
    <xs:element name = "hour" type = "xs:integer" />
    <xs:element name = "minute" type = "xs:integer" substitutionGroup = "nm:count" />
    <xs:element name = "surname" type = "xs:string" minOccurs = "1" maxOccurs =
"unbounded" />
    <xs:complexType name = "some_type" >
        <xs:all >
            <xs:element ref = "nm:hour" minOccurs = "0" maxOccurs = "1" >
                <xs:annotation >
                    <xs:documentation >hour ref
documentatuion</xs:documentation>
                </xs:annotation>
            </xs:element>
            <xs:element ref = "nm:minute" minOccurs = "0" maxOccurs = "1" />
        </xs:all>
    </xs:complexType>

```

complexType

Complex type maps to UML Class with stereotype XSDcomplexType.

- abstract - to UML Class abstract value(true | false).
- annotation - to UML Class documentation.
- attribute – to inner UML Class Attribute or UML Association End.
- attributeGroup – to UML AssociationEnd or UML Attribute with type XSDattributeGroup.

name – to UML Class name.

This class also can have stereotypes XSDsimpleContent, XSDcomplexContent, XSDall, XSDchoice, XSDsequence.

No stereotype – the same as “XSDsequence”.

Generalization between complex type and other type has stereotype XSDrestriction or XSDextension. We assume stereotype XSDextension if generalization do not have stereotype.

Some complex mapping:

- complexType with simpleContent – to UML Class. This class must be derived from other class and can must have stereotype XSDsimpleContent.
- complexType with complexContent – to UML Class. This class must be derived from other class and must have stereotype XSDcomplexContent.

complexType with group, all, choice or sequence – to UML class with appropriate stereotype.

```
<complexType
    abstract = boolean : false
    block = (#all | List of (extension | restriction))
    final = (#all | List of (extension | restriction))
    id = ID
    mixed = boolean : false
    name = NCName
    {any attributes with non-schema namespace . . .}
    Content: (annotation?, (simpleContent | complexContent | ((group | all
    (attribute | attributeGroup)*, anyAttribute?))))
</complexType>
```

When the <simpleContent> alternative is chosen, the following elements are relevant, and the remaining property mappings are as below. Note that either <restriction> or <extension> must be chosen as the content of <simpleContent>

```

<simpleContent
  id = ID
  {any attributes with non-schema namespace . . .}>
  Content: (annotation?, (restriction | extension))
</simpleContent>
<restriction
  base = QName
  id = ID
  {any attributes with non-schema namespace . . .}>
  Content: (annotation?, (simpleType?, (minExclusive | minInclusive | maxExclusive | maxInclusive | totalDigits | fractionDigits | length | minLength | maxLength | whiteSpace | pattern)*), ((attribute | attributeGroup)*, anyAttribute?))
</restriction>
<extension
  base = QName
  id = ID
  {any attributes with non-schema namespace . . .}>
  Content: (annotation?, ((attribute | attributeGroup)*, anyAttribute?))
</extension>
<attributeGroup
  id = ID
  ref = QName
  {any attributes with non-schema namespace . . .}>
  Content: (annotation?)
</attributeGroup>
<anyAttribute

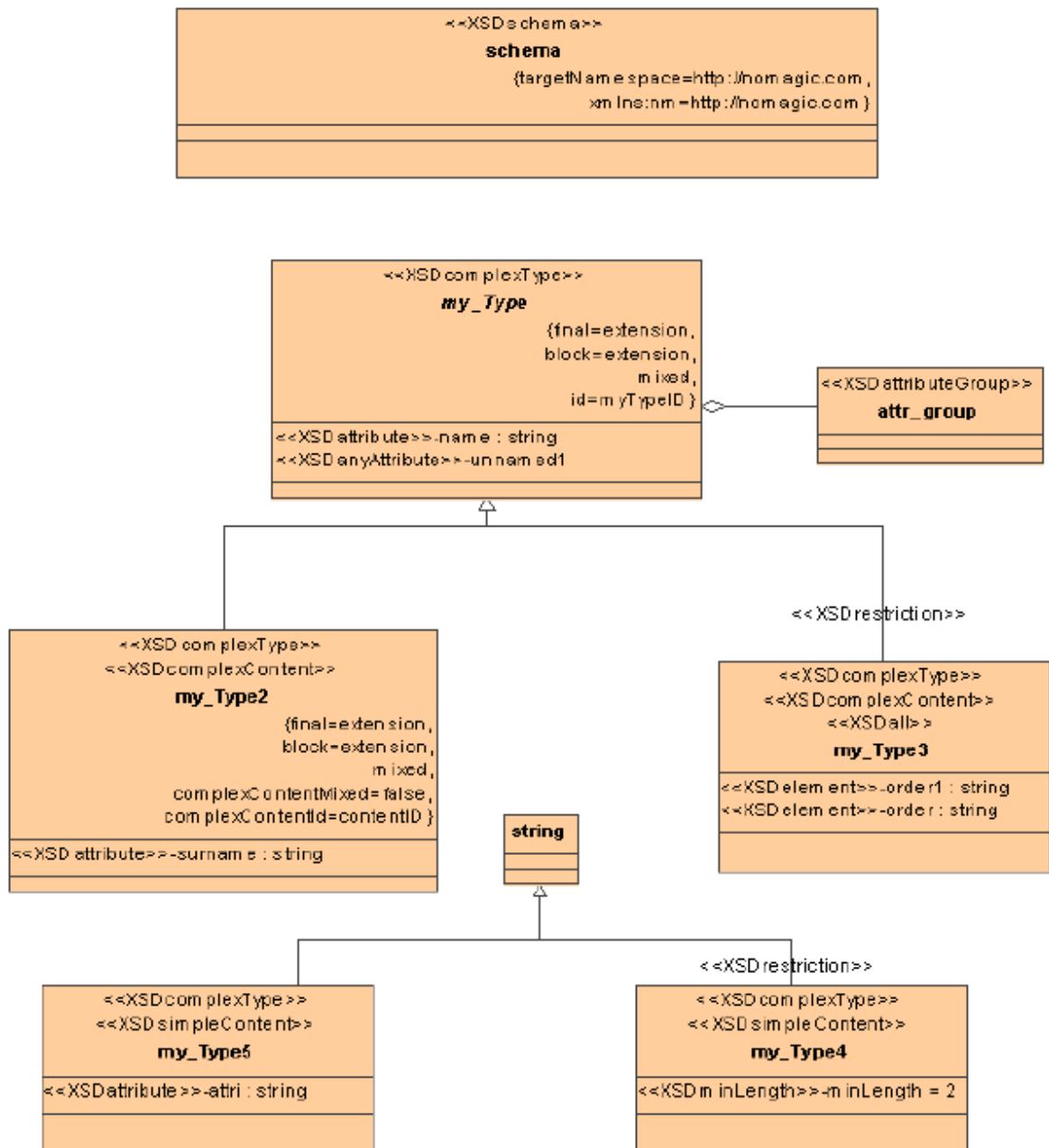
```

When the <complexContent> alternative is chosen, the following elements are relevant (as are the <attributeGroup> and <anyAttribute> elements, not repeated here), and the additional property mappings are as below. Note that either <restriction> or <extension> must be chosen as the content of <complexContent>, but their content models are different in this case from the case above when they occur as children of <simpleContent>.

The property mappings below are also used in the case where the third alternative (neither <simpleContent> nor <complexContent>) is chosen. This case is understood as shorthand for complex content restricting the **ur-type definition**, and the details of the mappings should be modified as necessary.

```
<complexContent
  id = ID
  mixed = boolean
  {any attributes with non-schema namespace . . .}>
  Content: (annotation?, (restriction | extension))
</complexContent>
<restriction
  base = QName
  id = ID
  {any attributes with non-schema namespace . . .}>
  Content: (annotation?, (group | all | choice | sequence)?, ((attribute |
attributeGroup)*, anyAttribute?))
</restriction>
<extension
  base = QName
  id = ID
  {any attributes with non-schema namespace . . .}>
  Content: (annotation?, ((group | all | choice | sequence)?, ((attribute |
attributeGroup)*, anyAttribute?)))
</extension>
```

complexType UML Model example:



```
<?xml version='1.0' encoding='Cp1252'?>
<xs:schema xmlns:nm = "http://nomagic.com" xmlns:xs =
"http://www.w3.org/2001/XMLSchema" targetNamespace = "http://nomagic.com" >
    <xs:complexType name = "my_Type2" block = "extension" final =
"extension" mixed = "true" >
        <xs:annotation >
            <xs:documentation >my_type2
documentation</xs:documentation>
        </xs:annotation>
```

```
= "contentID" mixed = "false" >
<e = "nm:my_Type" >
  te name = "surname" type = "xs:string" />

  ype3" >
  base = "nm:my_Type" >
    lement name = "order" type = "xs:string"
    lement name = "order1" type = "xs:string"
```

' XML Representation Summary: **attributeGroup** Element Information Item

```
<attributeGroup
  id = ID
  name = NCName
  ref = QName
  {any attributes with non-schema namespace . . .}>
  Content: (annotation? , ((attribute | attributeGroup)*, anyAttribute?)
</attributeGroup>
```

ype5" >

ie = "xs:string" >
te name = "attri" type = "xs:string" />

/pe" abstract = "true" block = "extension"
)" mixed = "true" >

l >my_type

name" type = "xs:string" />
 = "nm:attr_group" />

ttr_group" />

attributeGroup

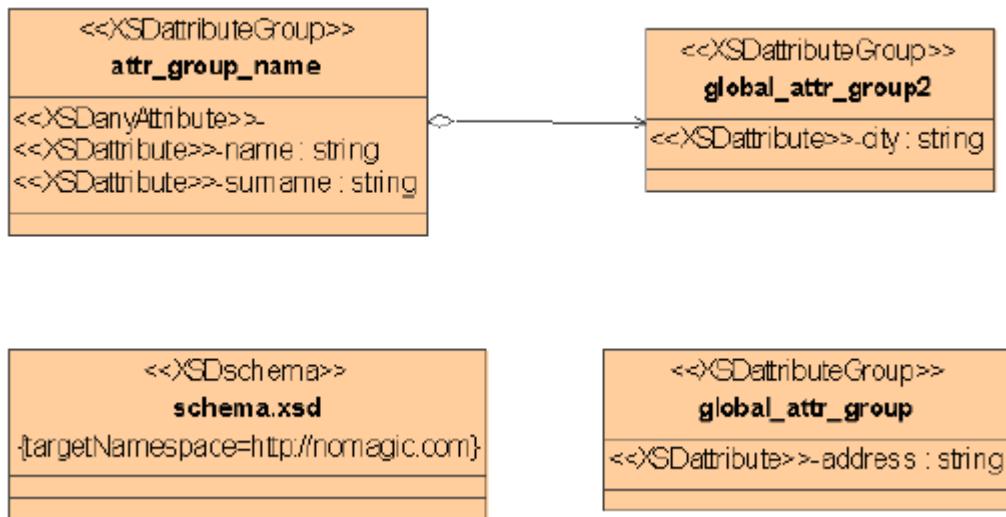
attributeGroup maps to simple UML Class with stereotype XSDattributeGroup.

- name – to UML Class name
 - annotation – to UML Class documentation
 - attribute – to inner UML Attribute or AssociationEnd with XSDattribute
 - stereotype.
 - attributeGroup - inner attributeGroup always must be just reference. Such reference maps to Attribute or AssociationEnd with type of referenced attributeGroup. The opposite Association End kind must be aggregated and it must be navigable.
 - anyAttribute – to inner UML Attribute with stereotype *XSDanyAttribute*.

If reference is generated, name is not generated.

When an `<attributeGroup>` appears as a daughter of `<schema>` or `<redefine>`, it corresponds to an attribute group definition as below. When it appears as a daughter of `<complexType>` or `<attributeGroup>`, it does not correspond to any component as such.

attributeGroup UML Model example:



```
<xs:schema xmlns:nm = "http://nomagic.com" xmlns:xs =
"http://www.w3.org/2001/XMLSchema" targetNamespace = "http://nomagic.com"
>
    <xs:attributeGroup name = "global_attr_group" >
        <xs:attribute name = "address" type = "xs:string" />
    </xs:attributeGroup>
    <xs:attributeGroup name = "attr_group_name" >
        <xs:annotation >
            <xs:documentation >attribute group
documentation</xs:documentation>
        </xs:annotation>
        <xs:attribute name = "surname" type = "xs:string" />
    </xs:attributeGroup>
    <xs:attribute name = "name" type = "xs:string" >
        <xs:annotation >
            <xs:documentation >name attribute
documentation</xs:documentation>
        </xs:annotation>
    </xs:attribute>
    <xs:attributeGroup ref = "nm:global_attr_group2" >
        <xs:annotation >
            <xs:documentation >reference
documentation</xs:documentation>
        </xs:annotation>
    </xs:attributeGroup>
    <xs:anyAttribute />
</xs:attributeGroup>
<xs:attributeGroup name = "global_attr_group2" >
    <xs:attribute name = "city" type = "xs:string" />
</xs:attributeGroup>
</xs:schema>
```

simpleType

Maps to UML Class with stereotype *XSDsimpleType*.

XML Representation Summary: **simpleType** Element Information Item

```

<simpleType
    final = (#all | (list | union | restriction))
    id = ID
    name = NCName
    {any attributes with non-schema namespace . . .}
    Content: (annotation?, (restriction | list | union))
</simpleType>
<restriction
    base = QName
    id = ID
    {any attributes with non-schema namespace . . .}
    Content: (annotation?, (simpleType?, (minExclusive | minInclusive | maxExclusive | maxInclusive | totalDigits | fractionDigits | length | minLength | maxLength | whiteSpace | pattern)*))
</restriction>
<list
    id = ID
    itemType = QName
    {any attributes with non-schema namespace . . .}
    Content: (annotation?, (simpleType?))
</list>
<union
    id = ID
    memberTypes = List of QName
    {any attributes with non-schema namespace . . .}
    Content: (annotation?, (simpleType*))
</union>
```

Example:

```

<xss:simpleType name="fahrenheitWaterTemp">
  <xss:restriction base="xss:number">
    <xss:fractionDigits value="2"/>
    <xss:minExclusive value="0.00"/>
    <xss:maxExclusive value="100.00"/>
  </xss:restriction>
</xss:simpleType>
```

The XML representation of a simple type definition.

restriction

To specify restriction generalization must be used between this class and super class. This generalization has or do not have *XSDrestriction* stereotype. Restriction id and annotation maps to Generalization properties.

In order to have inner simpleType element, parent of this Generalization must be inner Class of outer UML Class.

list

UML Class must have additional stereotype *XSDlist*.

Binding between this class and XSD:list must be provided.

“itemsType” maps to UML TemplateArgument from Binding.

union

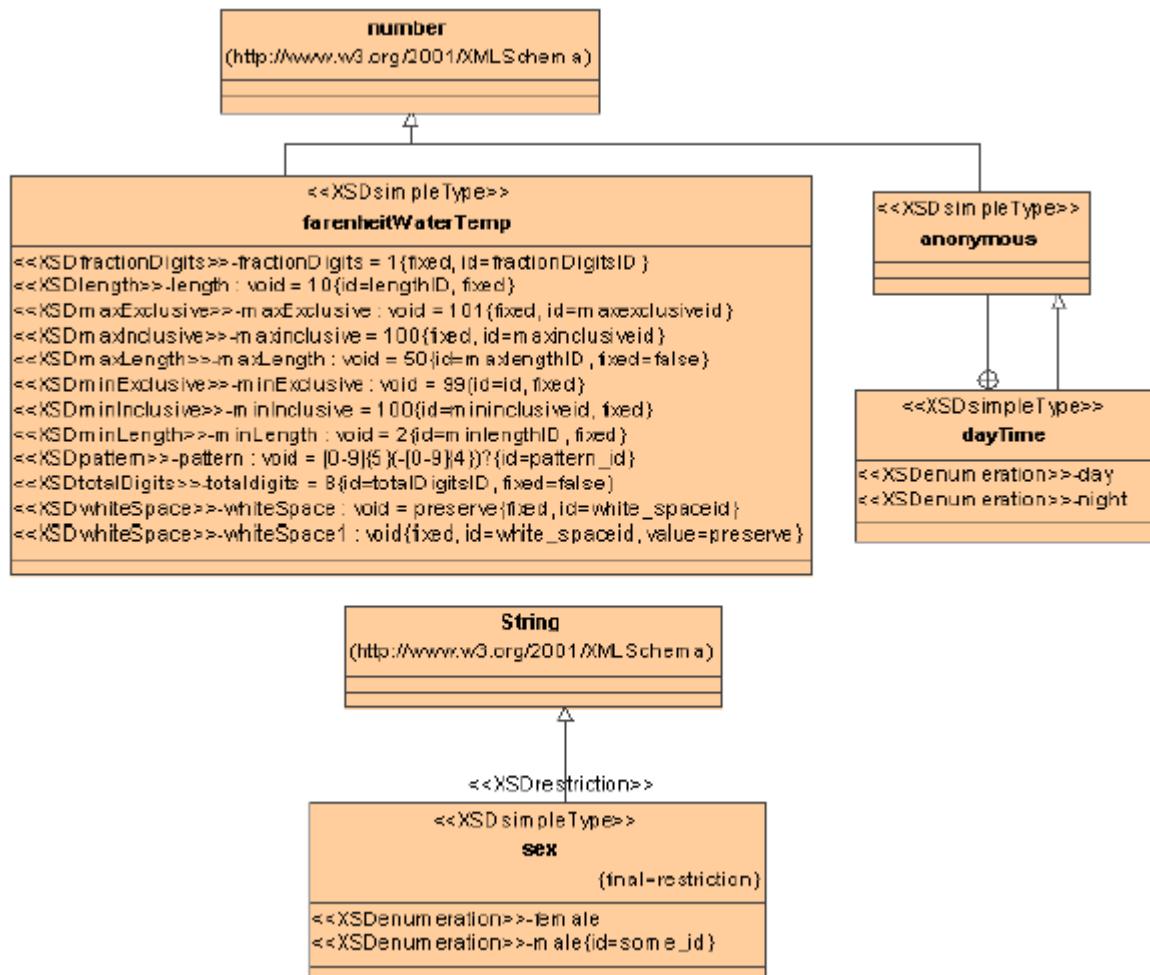
UML Class must have additional stereotype *XSDunion*.

“memberTypes” and inner simpleTypes maps to several UML Generalizations between this simple type and members types.

In order to have inner simpleType element, parent of this Generalization must be inner Class of outer UML Class.

simpleType UML Model example:

Restriction example



```
<xs:schema xmlns:nm = "http://nomagic.com" xmlns:xs =
"http://www.w3.org/2001/XMLSchema" targetNamespace = "http://nomagic.com" >
    <xs:simpleType name = "farenheitWaterTemp" >
        <xs:annotation >
            <xs:documentation >documentation of simple
type</xs:documentation>
        </xs:annotation>
        <xs:restriction base = "xs:number" >
            <xs:annotation >
                <xs:documentation >documentation of
restriction</xs:documentation>
            </xs:annotation>
            <xs:pattern id = "pattern_id" value = "[0-9]{5}(-[0-
9]{4})?" >
                <xs:annotation >
                    <xs:documentation >pattern
doc</xs:documentation>
```



```
        </xs:annotation>
    </xs:pattern>
    <xs:whiteSpace id = "white_spaceid" fixed = "true" value =
"preserve" >
        <xs:annotation >
            <xs:documentation >white space
doc</xs:documentation>
        </xs:annotation>
    </xs:whiteSpace>
    <xs:whiteSpace id = "white_spaceid" fixed = "true" value =
"preserve" >
        <xs:annotation >
            <xs:documentation >white space
doc</xs:documentation>
        </xs:annotation>
    </xs:whiteSpace>
    <xs:maxLength id = "maxlengthID" fixed = "false" value =
"50" >
        <xs:annotation >
            <xs:documentation >max length
documentation</xs:documentation>
        </xs:annotation>
    </xs:maxLength>
    <xs:minLength id = "minlengthID" fixed = "true" value =
"2" >
        <xs:annotation >
            <xs:documentation >min length
documentation</xs:documentation>
        </xs:annotation>
    </xs:minLength>
    <xs:length id = "lengthID" fixed = "true" value = "10" >
        <xs:annotation >
            <xs:documentation >length
documentation</xs:documentation>
        </xs:annotation>
    </xs:length>
    <xs:fractionDigits id = "fractionDigitsID" fixed = "true"
value = "1" >
        <xs:annotation >
            <xs:documentation >fraction digits
documentation</xs:documentation>
        </xs:annotation>
    </xs:fractionDigits>
    <xs:totalDigits id = "totalDigitsID" fixed = "false" value
= "8" >
        <xs:annotation >
            <xs:documentation >total digits
id</xs:documentation>
        </xs:annotation>
    </xs:totalDigits>
    <xs:maxInclusive id = "maxinclusiveid" fixed = "true"
value = "100" >
```



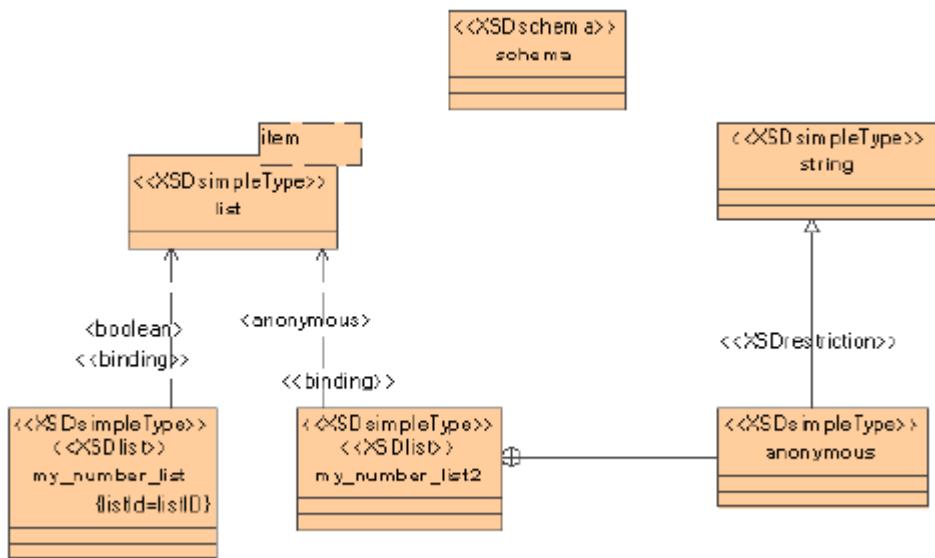
```
        <xs:annotation>
            <xs:documentation>min inclusive
documentation</xs:documentation>
        </xs:annotation>
    </xs:minInclusive>
    <xs:maxExclusive id = "maxexclusiveid" fixed = "true"
value = "101" >
        <xs:annotation>
            <xs:documentation>max exclusive
documentation</xs:documentation>
        </xs:annotation>
    </xs:maxExclusive>
    <xs:minExclusive id = "id" fixed = "true" value = "99" >
        <xs:annotation>
            <xs:documentation>min exclusive
documentation</xs:documentation>
        </xs:annotation>
    </xs:minExclusive>
</xs:restriction>
</xs:simpleType>
<xs:simpleType name = "dayTime" >
    <xs:annotation>
        <xs:documentation>day time
documentation</xs:documentation>
    </xs:annotation>
    <xs:restriction >
        <xs:annotation>
            <xs:documentation>restriction
documentation</xs:documentation>
        </xs:annotation>
    <xs:simpleType >
        <xs:restriction base = "xs:number" />
    </xs:simpleType>
    <xs:enumeration value = "day" >
        <xs:annotation>
            <xs:documentation>day
value</xs:documentation>
        </xs:annotation>
    </xs:enumeration>
    <xs:enumeration value = "night" >
        <xs:annotation>
            <xs:documentation>night
value</xs:documentation>
        </xs:annotation>
    </xs:enumeration>
</xs:restriction>
</xs:simpleType>
<xs:simpleType name = " sex" final = "restriction" >
    <xs:annotation>
        <xs:documentation>documentation of simple type
restriction</xs:documentation>
    </xs:annotation>
```

```

        </xs:annotation>
        </xs:enumeration>
    </xs:restriction>
</xs:simpleType>
</xs:schema>

```

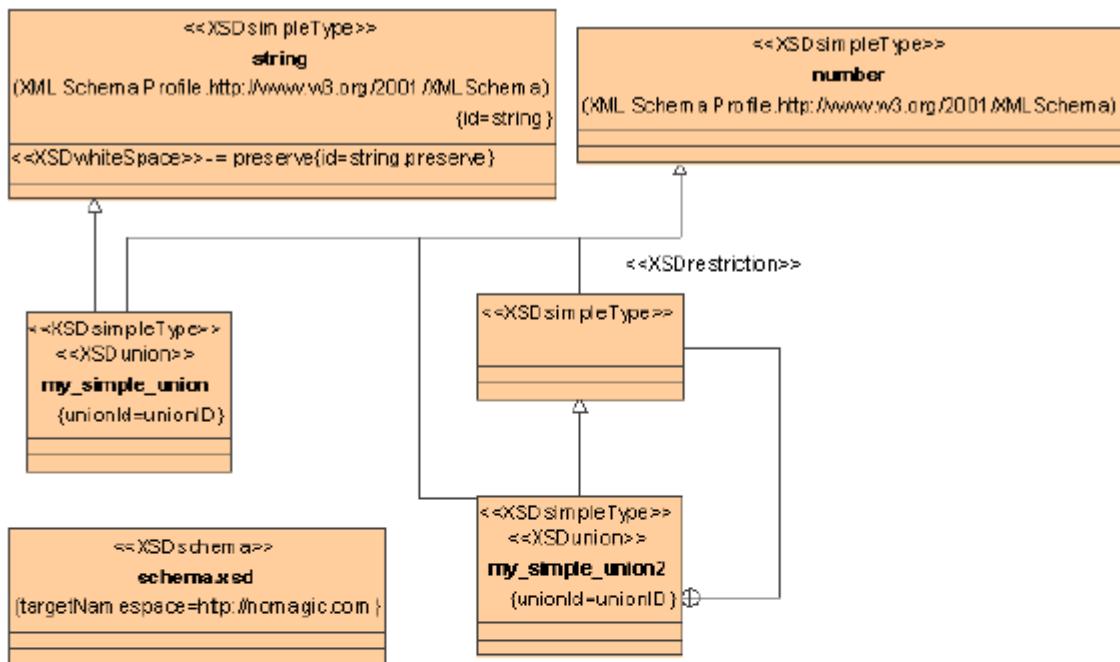
list example



```
<?xml version='1.0' encoding='UTF-8'?>

<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" >
    <xs:simpleType name="my_number_list2" >
        <xs:list >
            <xs:simpleType >
                <xs:restriction base="xs:string" />
            </xs:simpleType>
        </xs:list>
    </xs:simpleType>
    <xs:simpleType name="my_number_list" >
        <xs:annotation >
            <xs:documentation >my list
documentation</xs:documentation>
        </xs:annotation>
        <xs:list itemType="xs:boolean" />
    </xs:simpleType>
</xs:schema>
```

union example



```

<xs:schema xmlns:nm = "http://nomagic.com" xmlns:xs =
"http://www.w3.org/2001/XMLSchema" targetNamespace = "http://nomagic.com" >
  <xs:simpleType name = "my_simple_union" >
    <xs:union id = "unionID" memberTypes = "xs:string xs:number" />
  </xs:simpleType>
  <xs:simpleType name = "my_simple_union2" >
    <xs:annotation >
      <xs:documentation >very important
      documentation</xs:documentation>
    </xs:annotation>
    <xs:union id = "unionID" memberTypes = "xs:number" >
      <xs:simpleType >
        <xs:restriction base = "xs:number" />
      </xs:simpleType>
    </xs:union>
  </xs:simpleType>
</xs:schema>
  
```

minExclusive

Maps to UML Attribute with stereotype *XSDminExclusive*. Name and type of such attribute does not make sense.

- value – to Attribute initial value.

XML Representation Summary: **minExclusive** Element Information Item

```
<minExclusive
    fixed = boolean : false
    id = ID
    value = anySimpleType
    {any attributes with non-schema namespace . . .}
    Content: (annotation?)
</minExclusive>
{value} ·must· be in the ·value space· of {base type definition}.
```

Example

The following is the definition of a ·user-derived· datatype which limits values to integers greater than or equal to 100, using ·minExclusive·.

```
<simpleType name='more-than-ninety-nine'>
    <restriction base='integer'>
        <minExclusive value='99'/>
    </restriction>
</simpleType>
```

Note that the ·value space· of this datatype is identical to the previous one (named 'one-hundred-or-more').

minExclusive UML Model example

For an example, see “simpleType UML Model example:” on page 483.

maxExclusive

Maps to UML Attribute with stereotype *XSDmaxExclusive*. Name and type of such attribute does not make sense.

- value – to Attribute initial value.

XML Representation Summary: **maxExclusive** Element Information Item

```
<maxExclusive
    fixed = boolean : false
    id = ID
    value = anySimpleType
    {any attributes with non-schema namespace . . .}>
    Content: (annotation?)
</maxExclusive>
{value} ·must· be in the ·value space· of {base type definition}.
```

Example

The following is the definition of a ·user-derived· datatype which limits values to integers less than or equal to 100, using ·maxExclusive·.

```
<simpleType name='less-than-one-hundred-and-one'>
    <restriction base='integer'>
        <maxExclusive value='101' />
    </restriction>
</simpleType>
```

Note that the ·value space· of this datatype is identical to the previous one (named 'one-hundred-or-less').

maxExclusive UML Model example

For an example, see “simpleType UML Model example:” on page 483.

minInclusive

Maps to UML Attribute with stereotype *XSDminInclusive*. Name and type of such attribute does not make sense.

- value – to Attribute initial value.

XML Representation Summary: **minInclusive** Element Information Item

```
<minInclusive
    fixed = boolean : false
    id = ID
    value = anySimpleType
    {any attributes with non-schema namespace . . .}>
    Content: (annotation?)
</minInclusive>
{value} ·must· be in the ·value space· of {base type definition}.
```

Example

The following is the definition of a ·user-derived· datatype which limits values to integers greater than or equal to 100, using ·minInclusive·.

```
<simpleType name='one-hundred-or-more'>
    <restriction base='integer'>
        <minInclusive value='100' />
    </restriction>
</simpleType>
```

minInclusive UML Model example

For an example, see “simpleType UML Model example.” on page 483

maxInclusive

Maps to UML Attribute with stereotype *XSDmaxInclusive*. Name and type of such attribute does not make sense.

- value – to Attribute initial value.

XML Representation Summary: maxInclusive Element Information Item

```
<maxInclusive
    fixed = boolean : false
    id = ID
    value = anySimpleType
    {any attributes with non-schema namespace . . .}
    Content: (annotation?)
</maxInclusive>
{value} ·must· be in the ·value space· of {base type definition}.
```

Example

The following is the definition of a ·user-derived· datatype which limits values to integers less than or equal to 100, using ·maxInclusive·.

```
<simpleType name='one-hundred-or-less'>
  <restriction base='integer'>
    <maxInclusive value='100' />
  </restriction>
</simpleType>
```

maxInclusive UML Model example

For an example, see “simpleType UML Model example:” on page 483

totalDigits

Maps to UML Attribute with stereotype *XSDtotalDigits*. Name and type of such attribute does not make sense.

- value – to Attribute initial value.

XML Representation Summary: totalDigits Element Information Item

```
<totalDigits  
    fixed = boolean : false  
    id = ID  
    value = positiveInteger  
    {any attributes with non-schema namespace . . .}>  
Content: (annotation?)  
</totalDigits>
```

Example

The following is the definition of a ‘user-derived’ datatype which could be used to represent monetary amounts, such as in a financial management application which does not have figures of \$1M or more and only allows whole cents. This definition would appear in a schema authored by an “end-user” and shows how to define a datatype by specifying facet values which constrain the range of the ‘base type’ in a manner specific to the ‘base type’ (different than specifying max/min values as before).

```
<simpleType name='amount'>  
    <restriction base='decimal'>  
        <totalDigits value='8' />  
        <fractionDigits value='2' fixed='true' />  
    </restriction>  
</simpleType>
```

totalDigits UML Model example

For an example, see “simpleType UML Model example:” on page 483

fractionDigits

Maps to UML Attribute with stereotype *XSDfractionDigits*. Name and type of such attribute does not make sense.

- value – to Attribute initial value.

XML Representation Summary: fractionDigits Element Information Item

```
<fractionDigits  
    fixed = boolean : false  
    id = ID  
    value = nonNegativeInteger  
    {any attributes with non-schema namespace . . .}>  
    Content: (annotation)>  
</fractionDigits>
```

Example

The following is the definition of a ‘user-derived’ datatype which could be used to represent the magnitude of a person’s body temperature on the Celsius scale. This definition would appear in a schema authored by an “end-user” and shows how to define a datatype by specifying facet values which constrain the range of the ‘base type’.

```
<simpleType name='celsiusBodyTemp'>  
    <restriction base='decimal'>  
        <totalDigits value='4' />  
        <fractionDigits value='1' />  
        <minInclusive value='36.4' />  
        <maxInclusive value='40.5' />  
    </restriction>  
</simpleType>
```

fractionDigits UML Model example

For an example, see “simpleType UML Model example:” on page 483

length

Maps to UML Attribute with stereotype *XSDlength*. Name and type of such attribute does not make sense.

- value – to Attribute initial value.

XML Representation Summary: `length` Element Information Item

```
<length
  fixed = boolean : false
  id = ID
  value = nonNegativeInteger
  {any attributes with non-schema namespace . . .}>
  Content: (annotation?)
</length>
```

Example

The following is the definition of a ‘user-derived’ datatype to represent product codes which must be exactly 8 characters in length. By fixing the value of the length facet we ensure that types derived from `productCode` can change or set the values of other facets, such as pattern, but cannot change the length.

```
<simpleType name='productCode'>
  <restriction base='string'>
    <length value='8' fixed='true' />
  </restriction>
</simpleType>
```

length UML Model example

For an example, see “`simpleType` UML Model example.” on page 483

minLength

Maps to UML Attribute with stereotype `XSDminLength`. Name and type of such attribute does not make sense.

- `value` – to Attribute initial value.

XML Representation Summary: **minLength** Element Information Item

```
<minLength
  fixed = boolean : false
  id = ID
  value = nonNegativeInteger
  {any attributes with non-schema namespace . . .}>
  Content: (annotation?)
</minLength>
```

Example

The following is the definition of a ‘user-derived’ datatype which requires strings to have at least one character (i.e., the empty string is not in the ‘value space’ of this datatype).

```
<simpleType name='non-empty-string'>
  <restriction base='string'>
    <minLength value='1'/>
  </restriction>
</simpleType>
```

minLength UML Model example

For an example, see “simpleType UML Model example:” on page 483

maxLength

Maps to UML Attribute with stereotype *XSDmaxLength*. Name and type of such attribute does not make sense.

- value – to Attribute initial value.

XML Representation Summary: **maxLength** Element Information Item

```
<maxLength
  fixed = boolean : false
  id = ID
  value = nonNegativeInteger
  {any attributes with non-schema namespace . . .}>
  Content: (annotation?)
</maxLength>
```

Example

The following is the definition of a ‘user-derived’ datatype which might be used to accept form input with an upper limit to the number of characters that are acceptable.

```
<simpleType name='form-input'>
  <restriction base='string'>
    <maxLength value='50' />
  </restriction>
</simpleType>
```

maxLength UML Model example

For an example, see “simpleType UML Model example.” on page 483

whiteSpace

Maps to UML Attribute with stereotype `XSDwhiteSpace`. Name and type of such attribute does not make sense.

- value – to Attribute initial value.

XML Representation Summary: `whiteSpace` Element Information Item

```
<whiteSpace
  fixed = boolean : false
  id = ID
  value = (collapse | preserve | replace)
  (any attributes with non-schema namespace . . .)
  Content: (annotation?)
</whiteSpace>
```

Example

The following example is the datatype definition for the `token` ‘built-in’ ‘derived’ datatype.

```
<simpleType name='token'>
  <restriction base='normalizedString'>
    <whiteSpace value='collapse' />
  </restriction>
</simpleType>
```

whiteSpace UML Model example

For an example, see “simpleType UML Model example:” on page 483

pattern

Maps to UML Attribute with stereotype *XSDpattern*. Name and type of such attribute does not make sense.

- value – to Attribute initial value or TaggedValue with name ‘value’.

XML Representation Summary: pattern Element Information Item

```
<pattern
  id = ID
  value = anySimpleType
  {any attributes with non-schema namespace . . .}
  Content: (annotation?)
</pattern>
{value} ·must· be a valid ·regular expression·.
```

Example

The following is the definition of a ·user-derived· datatype which is a better representation of postal codes in the United States, by limiting strings to those which are matched by a specific ·regular expression·:

```
<simpleType name='better-us-zipcode'>
  <restriction base='string'>
    <pattern value='[0-9]{5}(-[0-9]{4})?' />
  </restriction>
</simpleType>
```

pattern UML Model example

For an example, see “simpleType UML Model example:” on page 483

enumeration

Maps to UML Attribute with stereotype XSDEnumeration.

- value – to Attribute name.

XML Representation Summary: enumeration Element Information Item

```
<enumeration
  id = ID
  value = anySimpleType
  {any attributes with non-schema namespace . . .}>
  Content: (annotation?)
</enumeration>
```

Example

The following example is a datatype definition for a ·user-derived· datatype which limits the values of dates to the three US holidays enumerated. This datatype definition would appear in a schema authored by an "end-user" and shows how to define a datatype by enumerating the values in its ·value space·. The enumerated values must be type-valid literals for the ·base type·.

```
<simpleType name='holidays'>
  <annotation>
    <documentation>some US holidays</documentation>
  </annotation>
  <restriction base='gMonthDay'>
    <enumeration value='--01-01'>
      <annotation>
        <documentation>New Year's day</documentation>
      </annotation>
    </enumeration>
    <enumeration value='--07-04'>
      <annotation>
        <documentation>4th of July</documentation>
      </annotation>
    </enumeration>
    <enumeration value='--12-25'>
      <annotation>
        <documentation>Christmas</documentation>
      </annotation>
    </enumeration>
  </restriction>
</simpleType>
```

enumeration UML Model example

For an example, see “simpleType UML Model example” on page 483

unique

Maps to UML Attribute added into some UML Class.

```
<unique
  id = ID
  name = NCName
  {any attributes with non-schema namespace . . .}>
  Content: (annotation?, (selector, field+))
</unique>
```

unique UML Model example

For an example, see “keyref UML Model example” on page 504

key

Maps to UML Attribute added into some UML Class.

- name – to Attribute name.
- id – to TaggedValue.

```
<key
  id = ID
  name = NCName
  {any attributes with non-schema namespace . . .}>
  Content: (annotation?, (selector, field+))
</key>
```

key UML Model example

For an example, see “keyref UML Model example” on page 504

keyref

Maps to UML Attribute added into some UML Class.

- refer – to value of “refer” or “referString” TaggedValue.
- name – to Attribute name.

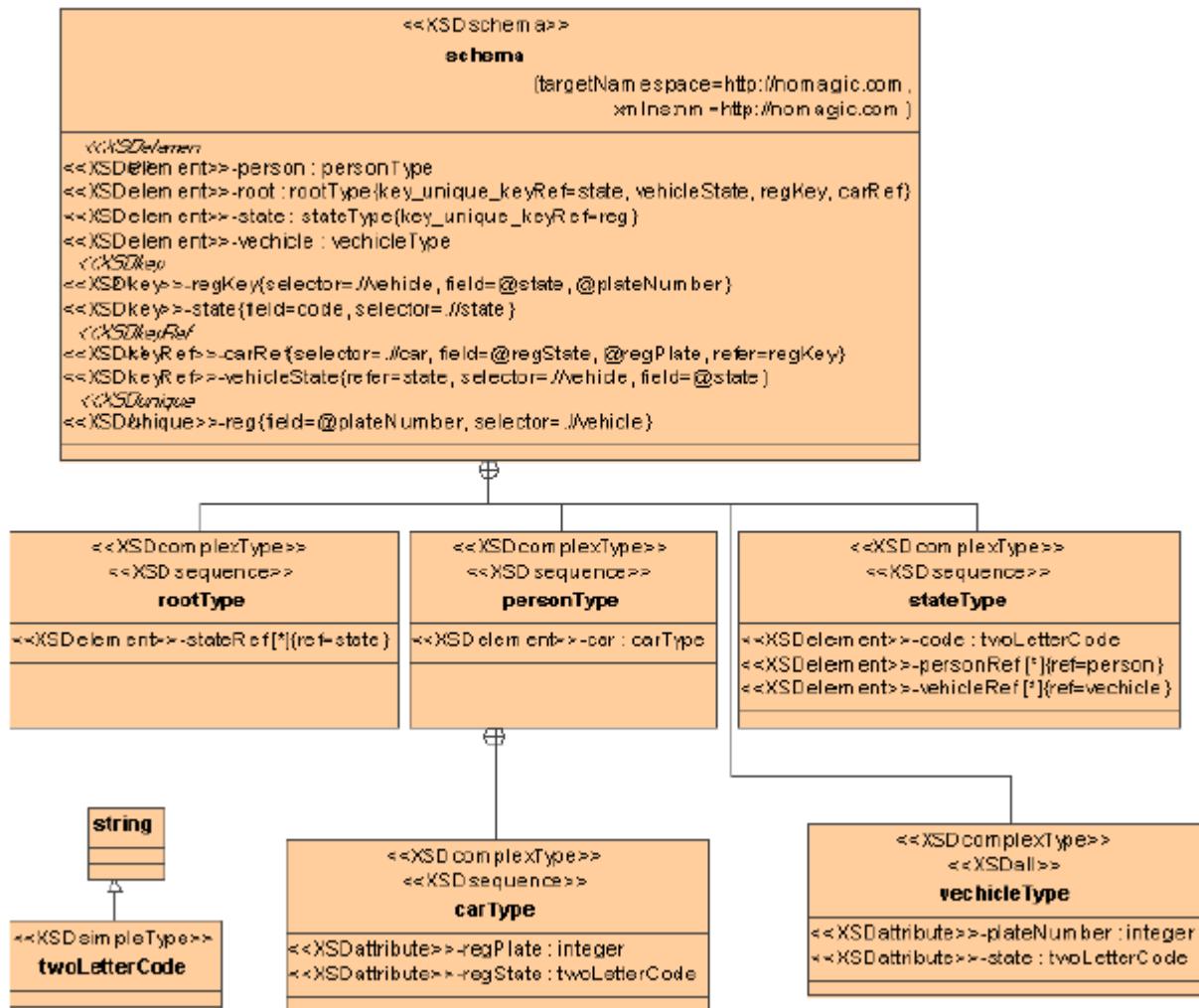
- id – to TaggedValue.

```

<keyref
  id = ID
  name = NCName
  refer = QName
  (any attributes with non-schema namespace . . .)
  Content: (annotation?, (selector, field+) )
</keyref>

```

keyref UML Model example



```
<xs:schema xmlns:nm = "http://nomagic.com" xmlns:xs =
"http://www.w3.org/2001/XMLSchema" targetNamespace = "http://nomagic.com" >
    <xs:element name = "vechicle" >
        <xs:complexType >
            <xs:all />
                <xs:attribute name = "plateNumber" type = "xs:integer" />
                <xs:attribute name = "state" type = "nm:twoLetterCode" />
        </xs:complexType>
    </xs:element>
    <xs:element name = "state" >
        <xs:complexType >
            <xs:sequence >
                <xs:element name = "code" type = "nm:twoLetterCode" />
```



```

                <xs:element ref = "nm:vechicle" maxOccurs =
"unbounded" />
                <xs:element ref = "nm:person" maxOccurs =
"unbounded" />
            </xs:sequence>
        </xs:complexType>
        <xs:unique name = "reg" >
            <xs:annotation >
                <xs:documentation >unique
documentation</xs:documentation>
            </xs:annotation>
            <xs:selector xpath = "./vehicle" />
            <xs:field xpath = "@plateNumber" />
        </xs:unique>
    </xs:element>
    <xs:element name = "person" >
        <xs:complexType >
            <xs:sequence >
                <xs:element name = "car" >
                    <xs:complexType >
                        <xs:sequence />
                        <xs:attribute name = "regPlate" type =
"xs:integer" />
                        <xs:attribute name = "regState" type =
"nm:twoLetterCode" />
                    </xs:complexType>
                </xs:element>
            </xs:sequence>
        </xs:complexType>
    </xs:element>
    <xs:element name = "root" >
        <xs:complexType >
            <xs:sequence >
                <xs:element ref = "nm:state" maxOccurs = "unbounded"
/>
            </xs:sequence>
        </xs:complexType>
        <xs:key name = "state" >
            <xs:selector xpath = "./state" />
            <xs:field xpath = "code" />
        </xs:key>
        <xs:keyref name = "vehicleState" refer = "nm:state" >
            <xs:selector xpath = "./vehicle" />
            <xs:field xpath = "@state" />
        </xs:keyref>
        <xs:key name = "regKey" >
            <xs:annotation >
                <xs:documentation >key
documentation</xs:documentation>
            </xs:annotation>
            <xs:selector xpath = "./vehicle" />

```

```

    </xs:annotation>
    <xs:selector xpath = "./car" />
    <xs:field xpath = "@regState" />
    <xs:field xpath = "@regPlate" />
  </xs:keyref>
</xs:element>
<xs:simpleType name = "twoLetterCode" >
  <xs:restriction base = "xs:string" />
</xs:simpleType>
</xs:schema>

```

selector and field

Maps to UML TaggedValues named “selector” and “field” of UML Attribute representing key, keyRef or unique. “selector” tag has value representing “xpath” and “field” - list of values representing field “xpath”. ID values shall be skipped and annotation documentation will be applied to tagged value according to annotation rule (see:annotation). For field values annotation documentation shall be merged in one.

```

<selector
  id = ID
  xpath = a subset of XPath expression, see below
  {any attributes with non-schema namespace . . .}
  Content: (annotation?)
</selector>
<field
  id = ID
  xpath = a subset of XPath expression, see below
  {any attributes with non-schema namespace . . .}
  Content: (annotation?)
</field>

```

Example

```
<xs:key name="fullName">
  <xs:selector xpath=".//person"/>
  <xs:field xpath="forename"/>
  <xs:field xpath="surname"/>
</xs:key>

<xs:keyref name="personRef" refer="fullName">
  <xs:selector xpath=".//personPointer"/>
  <xs:field xpath="@first"/>
  <xs:field xpath="@last"/>
</xs:keyref>

<xs:unique name="nearlyID">
  <xs:selector xpath=".//*"/>
  <xs:field xpath="@id"/>
</xs:unique>
```

XML representations for the three kinds of identity-constraint definitions.

Example

```
<xs:element name="state">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="code" type="twoLetterCode"/>
      <xs:element ref="vehicle" maxOccurs="unbounded"/>
      <xs:element ref="person" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>

  <xs:key name="reg"> <!-- vehicles are keyed by their plate within states -->
    <xs:selector xpath=".//vehicle"/>
    <xs:field xpath="@plateNumber"/>
  </xs:key>
</xs:element>

<xs:element name="root">
  <xs:complexType>
    <xs:sequence>
      . . .
      <xs:element ref="state" maxOccurs="unbounded"/>
      . . .
    </xs:sequence>
  </xs:complexType>

  <xs:key name="state"> <!-- states are keyed by their code -->
    <xs:selector xpath=".//state"/>
    <xs:field xpath="code"/>
  </xs:key>

  <xs:keyref name="vehicleState" refer="state">
    <!-- every vehicle refers to its state -->
    <xs:selector xpath=".//vehicle"/>
    <xs:field xpath="@state"/>
  </xs:keyref>

  <xs:key name="regKey"> <!-- vehicles are keyed by a pair of state and plate number -->
    <xs:selector xpath=".//vehicle"/>
    <xs:field xpath="@state"/>
    <xs:field xpath="@plateNumber"/>
  </xs:key>

  <xs:keyref name="carRef" refer="regKey"> <!-- people's cars are a reference -->
    <xs:selector xpath=".//car"/>
    <xs:field xpath="@regState"/>
    <xs:field xpath="@regPlate"/>
  </xs:keyref>
</xs:element>
```

```
<xs:element name="person">
<xs:complexType>
<xs:sequence>
  .
  .
  <xs:element name="car">
    <xs:complexType>
      <xs:attribute name="regState" type="twoLetterCode" />
      <xs:attribute name="regPlate" type="xs:integer"/>
    </xs:complexType>
  </xs:element>
</xs:sequence>
</xs:complexType>
</xs:element>
```

A *state* element is defined, which contains a *code* child and some *vehicle* and *person* children. A *vehicle* in turn has a *plateNumber* attribute, which is an integer, and a *state* attribute. State's *code*'s are a key for them within the document. Vehicle's *plateNumber*'s are a key for them within states, and *state* and *plateNumber* is asserted to be a key for *vehicle* within the document as a whole. Furthermore, a *person* element has an empty car child, with *regState* and *regPlate* attributes, which are then asserted together to refer to *vehicles* via the *carRef* constraint. The requirement that a *vehicle*'s *state* match its containing *state*'s *code* is not expressed here.

selector and field UML Model example

For an example, see “keyref UML Model example” on page 504

annotation

Maps to UML Comment with or without stereotype XSDannotation.

Documentation's content maps to UML Comment body(name).

“documentation” maps as UML comment:

- “content” value shall be comment name
- “xml:lang” value – tag “xml:lang” value
- source value – tag “source” value

“appinfo” maps as tag value with name “appInfoSource”:

- “source” value will be tag value
- “content” will be documentation for tagged value

Appearing several annotation nodes on one element node, mapping shall be done in following way:

- “documentation” text shall be merged into one UML comment with merged content, but “content” and “xml:lang” tag values shall represent only first matched values
- “appInfo” shall have: “content” merged into one tag “appInfoSource” comment, but tag value shall represent first matched “appinfo”

XML Representation Summary: `annotation` Element Information Item

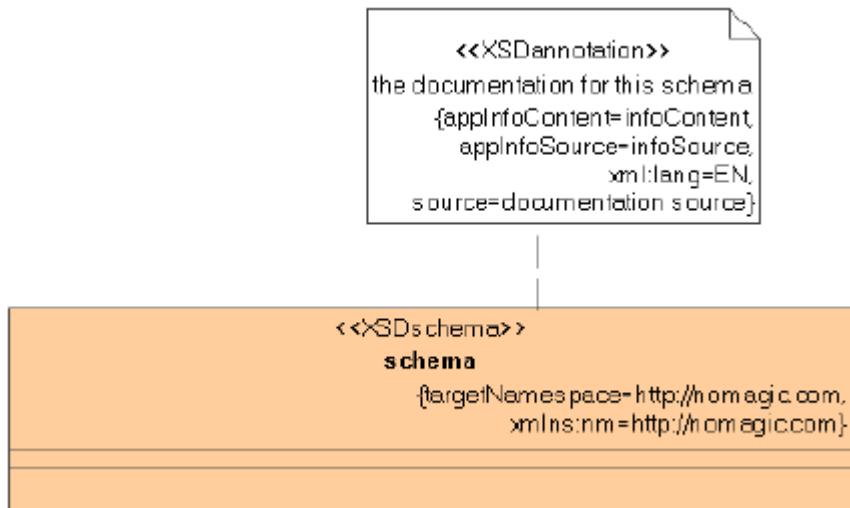
```
<annotation
  id = ID
  {any attributes with non-schema namespace . . .}>
  Content: (appinfo | documentation)*
</annotation>
<appinfo
  source = anyURI>
  Content: ({any})*
</appinfo>
<documentation
  source = anyURI
  xml:lang = language>
  Content: ({any})*
</documentation>
```

Example

```
<xss:simpleType fn:note="special">
  <xss:annotation>
    <xss:documentation>A type for experts only</xss:documentation>
    <xss:appinfo>
      <fn:specialHandling>checkForPrimes</fn:specialHandling>
    </xss:appinfo>
  </xss:annotation>
```

XML representations of three kinds of annotation.

annotation UML Model example



```

<xs:schema xmlns:nm = "http://nomagic.com" xmlns:xs =
"http://www.w3.org/2001/XMLSchema" targetNamespace = "http://nomagic.com" >
  <xs:annotation >
    <xs:appinfo source = "infoSource" >infoContent</xs:appinfo>

    <xs:documentation source = "documentation source" xml:lang =
"EN" >the documentation for this schema</xs:documentation>
  </xs:annotation>
</xs:schema>
  
```

compositors

Complex type maps to UML Class with stereotype `XSDcomplexType`. In order to have some group in complex type, the same UML Class also must have `XSDall`, `XSDchoice` or `XSDsequence` stereotype.

UML model can have ModelClass just with single stereotype `XSDall`, `XSDchoice` or `XSDsequence`. In this case such class maps to inner part of other group.

Elements order in sequence group is very important. Such elements are ordered according values of TaggedValue sequenceOrder.

```

<all
  id = ID
  maxOccurs = 1 : 1
  minOccurs = (0 | 1) : 1
  {any attributes with non-schema namespace . . .}>
  Content: (annotation?, element*)
</all>
<choice
  id = ID
  maxOccurs = (nonNegativeInteger | unbounded) : 1
  minOccurs = nonNegativeInteger : 1
  {any attributes with non-schema namespace . . .}>
  Content: (annotation?, (element | group | choice | sequence | any)*)
</choice>
<sequence
  id = ID
  maxOccurs = (nonNegativeInteger | unbounded) : 1
  minOccurs = nonNegativeInteger : 1
  {any attributes with non-schema namespace . . .}>
  Content: (annotation?, (element | group | choice | sequence | any)*)
</sequence>
```

Example

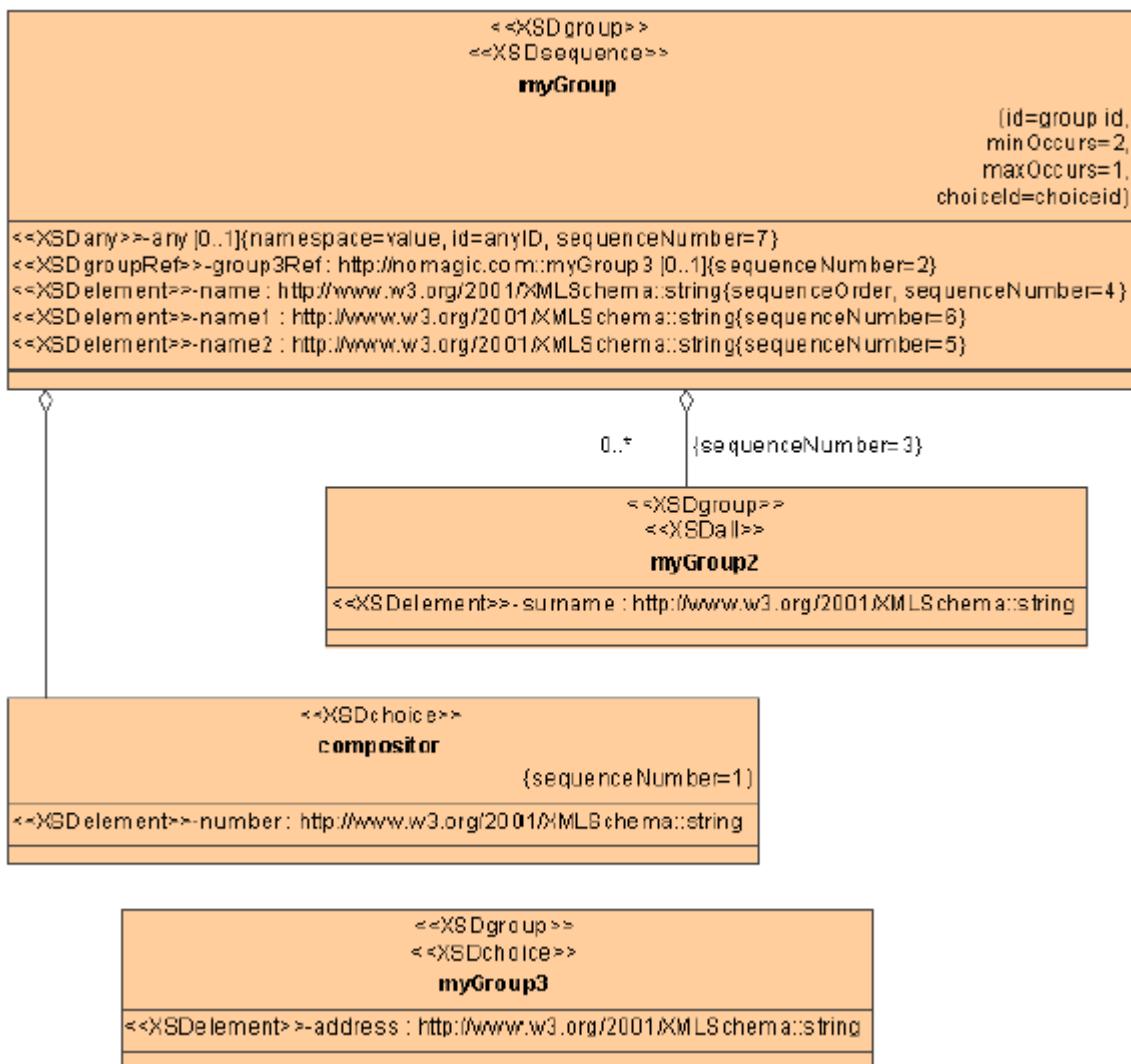
```

<xs:all>
  <xs:element ref="cats"/>
  <xs:element ref="dogs"/>
</xs:all>

<xs:sequence>
  <xs:choice>
    <xs:element ref="left"/>
    <xs:element ref="right"/>
  </xs:choice>
  <xs:element ref="landmark"/>
</xs:sequence>
```

XML representations for the three kinds of model group, the third nested inside the second.

compositors UML Model example



```
<?xml version='1.0' encoding='Cp1252'?>

<xs:schema xmlns:nm = "http://nomagic.com" xmlns:xs =
"http://www.w3.org/2001/XMLSchema" targetNamespace = "http://nomagic.com" >
    <xs:group name = "myGroup" >
        <xs:annotation >
            <xs:documentation >my group
documentation</xs:documentation>
        </xs:annotation>
        <xs:sequence minOccurs = "2" maxOccurs = "1" >
            <xs:choice >
                <xs:element name = "number" type = "xs:string" />
            </xs:choice>
            <xs:group ref = "nm:myGroup3" minOccurs = "0" maxOccurs =
"1" >
                <xs:annotation >
                    <xs:documentation >ref
documentation</xs:documentation>
                </xs:annotation>
            </xs:group>
            <xs:group ref = "nm:myGroup2" minOccurs = "0" maxOccurs =
"unbounded" >
                <xs:annotation >
                    <xs:documentation >another ref
documentation</xs:documentation>
                </xs:annotation>
            </xs:group>
            <xs:element name = "name" type = "xs:string" />
            <xs:element name = "name2" type = "xs:string" />
            <xs:element name = "name1" type = "xs:string" />
            <xs:any id = "anyID" namespace = "value" minOccurs = "0"
maxOccurs = "1" />
        </xs:sequence>
    </xs:group>
    <xs:group name = "myGroup3" >
        <xs:choice >
            <xs:element name = "address" type = "xs:string" />
        </xs:choice>
    </xs:group>
    <xs:group name = "myGroup2" >
        <xs:all >
            <xs:element name = "surname" type = "xs:string" />
        </xs:all>
    </xs:group>
</xs:schema>
```

group

Maps to UML Class with stereotype *XSDgroup*.

This class also may have stereotype *XSDall*, *XSDsequence* or *XSDchoice*.

If group has ref attribute, such group definition maps to UML Attribute or UML Association End. UML Attribute must have *XSDgroupRef* stereotype. This stereotype may be omitted for AssociationEnd.

XML Representation Summary: **group** Element Information Item

```
<group
  name = NCName>
  Content: (annotation?, (all | choice | sequence) )
</group>
<group
  ref = OName
  maxOccurs = (nonNegativeInteger | unbounded) : 1
  minOccurs = nonNegativeInteger : 1>
  Content: (annotation?)
</group>
```

Example

```
<xs:group name="myModelGroup">
  <xs:sequence>
    <xs:element ref="someThing"/>
    .
  </xs:sequence>
</xs:group>

<xs:complexType name="trivial">
  <xs:group ref="myModelGroup"/>
  <xs:attribute .../>
</xs:complexType>

<xs:complexType name="moreSo">
  <xs:choice>
    <xs:element ref="anotherThing"/>
    <xs:group ref="myModelGroup"/>
  </xs:choice>
  <xs:attribute .../>
</xs:complexType>
```

group UML Model example

For an example, see “compositors UML Model example” on page 515

any and anyAttribute

Maps to UML Attribute with stereotype *XSDany* or *XSDanyAttribute*.

maxOccurs - to multiplicity upper range. Value unbounded maps to asterisk in UML.

minOccurs – to multiplicity lower range.

annotation maps to Attribute documentation

Other properties to TaggedValues.

XML Representation Summary: *any* Element Information Item

```
<any
  id = ID
  maxOccurs = (nonNegativeInteger | unbounded) : 1
  minOccurs = nonNegativeInteger : 1
  namespace = ((##any | ##other) | List of (anyURI | (##targetNamespace |
##local)) ) : ##any
  processContents = (lax | skip | strict) : strict
  {any attributes with non-schema namespace . . .}
  Content: (annotation?)
</any>
<anyAttribute
  id = ID
  namespace = ((##any | ##other) | List of (anyURI | (##targetNamespace |
##local)) ) : ##any
  processContents = (lax | skip | strict) : strict
  {any attributes with non-schema namespace . . .}
  Content: (annotation?)
</anyAttribute>
```

Example

```
<xss:any processContents="skip"/>
<xss:any namespace="##other" processContents="lax"/>
<xss:any namespace="http://www.w3.org/1999/XSL/Transform"/>
<xss:any namespace="##targetNamespace"/>
<xss:anyAttribute namespace="http://www.w3.org/XML/1998/namespace"/>
```

XML representations of the four basic types of wildcard, plus one attribute wildcard.

any and anyAttribute UML Model example

```
<<XSD schema>>
schema
  (targetNamespace=http://nomagic.com,
  xmlns:nm=http://nomagic.com)
```

```
<<XSD attributeGroup>>
attr_group
```

```
<<XSD anyAttribute>>-any1{processContents=skip, namespace=http:\bla.bla.bla,id=anyID}
```

```
<<XSD group>>
<<XSD choice>>
```

```
my_type
```

```
<<XSD any>>-any[0..1]{id=anyID, processContents=strict, namespace=http:\bla}
```

```
<?xml version='1.0' encoding='Cp1252'?>

<xs:schema xmlns:nm = "http://nomagic.com" xmlns:xs =
"http://www.w3.org/2001/XMLSchema" targetNamespace = "http://nomagic.com"
>
    <xs:group name = "my_type" >
        <xs:choice >
            <xs:any id = "anyID" namespace = "http://bla"
processContents = "strict" minOccurs = "0" maxOccurs = "1" >
                <xs:annotation >
                    <xs:documentation >any
documentation</xs:documentation>
                </xs:annotation>
            </xs:any>
        </xs:choice>
    </xs:group>
    <xs:attributeGroup name = "attr_group" >
        <xs:anyAttribute id = "anyID" namespace = "http:\bla.bla.bla"
processContents = "skip" >
            <xs:annotation >
```

```

            <xs:documentation>any attribute
documentation</xs:documentation>
        </xs:annotation>
    </xs:anyAttribute>
</xs:attributeGroup>
</xs:schema>

```

schema

Maps to UML Class with stereotype *XSDschema*.

All schema global attributes and elements are mapped to UML Attributes of this class.

Name of this class should match file name or must be assigned to the component, which represents file.

“xmlns” xml tags maps to an permission link with stereotype <> and name, representing given prefix. Permission client is schema class and supplier package with name equal to the “xmlns” value.

XML Representation Summary: schema Element Information Item

```

<schema
    attributeFormDefault = (qualified | unqualified) : unqualified
    blockDefault = (#all | List of (extension | restriction | substitution))
    : ''
    elementFormDefault = (qualified | unqualified) : unqualified
    finalDefault = (#all | List of (extension | restriction)) : ''
    id = ID
    targetNamespace = anyURI
    version = token
    xml:lang = language
    {any attributes with non-schema namespace . . .}
    Content: ((include | import | redefine | annotation)*, (((simpleType |
complexType | group | attributeGroup) | element | attribute | notation),
annotation)*)
</schema>

```

Example

```

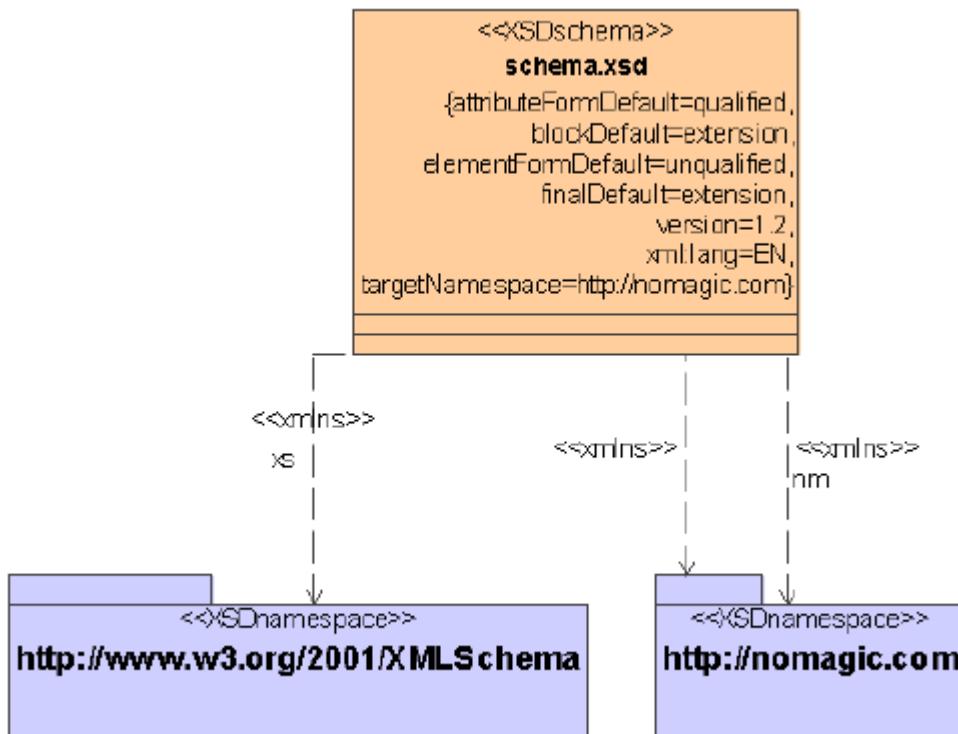
<xs:schema
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://www.example.com/example">

</xs:schema>

```

The XML representation of the skeleton of a schema.

schema UML Model example



```

<xss:schema xmlns:nm = "http://nomagic.com"
xmlns:xs = "http://www.w3.org/2001/XMLSchema"
xmlns = "http://nomagic.com"
attributeFormDefault = "qualified"
blockDefault = "extension"
elementFormDefault = "unqualified"
finalDefault = "extension"
targetNamespace = "http://nomagic.com"
version = "1.2"
xml:lang = "EN" />

```

notation

Maps to UML Attribute with stereotype *XSDnotation*. This attribute must be added into UML class with stereotype *XSDschema*.

- name maps to UML Attribute name
- annotation maps to UML Attribute documentation.

XML Representation Summary: notation Element Information Item

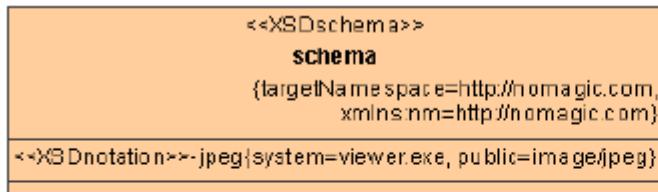
```
<notation
  id = ID
  name = NCName
  public = anyURI
  system = anyURI
  {any attributes with non-schema namespace . . .}>
  Content: (annotation?)
</notation>
```

Example

```
<xs:notation name="jpeg" public="image/jpeg" system="viewer.exe">
```

The XML representation of a notation declaration.

notation UML Model example



```

<xs:schema xmlns:nm = "http://nomagic.com"
  xmlns:xs = "http://www.w3.org/2001/XMLSchema"
  targetNamespace = "http://nomagic.com" >
  <xs:notation name = "jpeg" public = "image/jpeg" system = "viewer.exe"
/>
</xs:schema>
  
```

redefine

Maps to UML Class with stereotype *XSDredefine*. This class has inner UML Classes as redefined elements. Every redefined element must be derived from other UML class with stereotype *XSDsimpleType*, *XSDcomplexType*, *XSDgroup*, *XSDattributeGroup*. The name of this class shall match “schemaLocation” value.

If two “redefine” with the same schema location appears, they shall be merged to the one and the same class with a name “schemaLocation”.

Redefine Class must be inner class of XSDschema Class.

- annotation - to *XSDredefine* UML Class documentation
- schemaLocation – to *XSDredefine* UML Class name.

XML Representation Summary: `redefine` Element Information Item

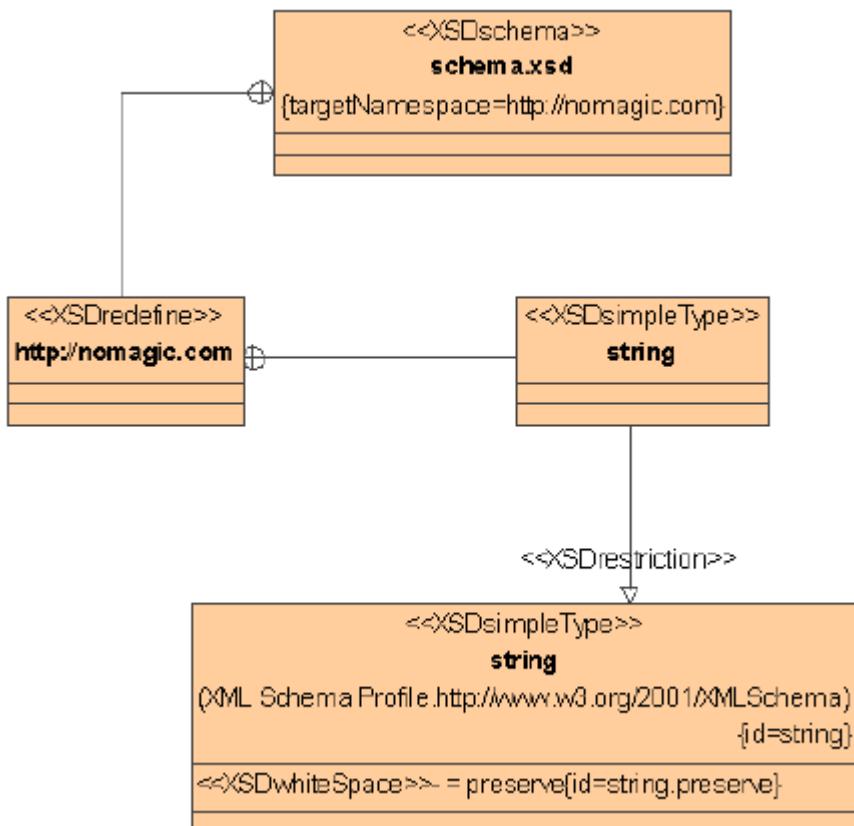
```
<redefine
  id = ID
  schemaLocation = anyURI
  {any attributes with non-schema namespace . . .}
  Content: (annotation | (simpleType | complexType | group |
attributeGroup) *)
</redefine>
```

Example

```
v1.xsd:  
  <xs:complexType name="personName">  
    <xs:sequence>  
      <xs:element name="title" minOccurs="0"/>  
      <xs:element name="forename" minOccurs="0" maxOccurs="unbounded"/>  
    </xs:sequence>  
  </xs:complexType>  
  
  <xs:element name="addressee" type="personName"/>  
  
v2.xsd:  
  <xs:redefine schemaLocation="v1.xsd">  
    <xs:complexType name="personName">  
      <xs:complexContent>  
        <xs:extension base="personName">  
          <xs:sequence>  
            <xs:element name="generation" minOccurs="0"/>  
          </xs:sequence>  
        </xs:extension>  
      </xs:complexContent>  
    </xs:complexType>  
  </xs:redefine>  
  
  <xs:element name="author" type="personName"/>
```

The schema corresponding to *v2.xsd* has everything specified by *v1.xsd*, with the *personName* type redefined, as well as everything it specifies itself. According to this schema, elements constrained by the *personName* type may end with a *generation* element. This includes not only the *author* element, but also the *addressee* element.

redefine UML Model example



```

<?xml version='1.0' encoding='UTF-8'?>

<xs:schema xmlns:nm="http://nomagic.com"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://nomagic.com" >
  <xs:redefine schemaLocation="http://nomagic.com" >
    <xs:simpleType name="string" >
      <xs:annotation >
        <xs:documentation >my
        documentation</xs:documentation>
      </xs:annotation>
      <xs:restriction base="xs:string" />
    </xs:simpleType>
  </xs:redefine>
</xs:schema>
  
```

import

Maps to UML Permission with stereotype *XSDimport*. Permission client must be schema class stereotypes <<XSDschema>> Component, supplier namespace Package *XSDnamespace*.

- namespace maps to supplier name.
- annotation maps to UML Attribute documentation.
- schemaLocation maps to TaggedValue.

XML Representation Summary: `import` Element Information Item

```
<import
  id = ID
  namespace = anyURI
  schemaLocation = anyURI
  {any attributes with non-schema namespace . . .}
  Content: (annotation?)
</import>
```

Example

The same namespace may be used both for real work, and in the course of defining schema components in terms of foreign components:

```
<schema xmlns="http://www.w3.org/2001/XMLSchema"
    xmlns:html="http://www.w3.org/1999/xhtml"
    targetNamespace="uri:mywork" xmlns:my="uri:mywork">

    <import namespace="http://www.w3.org/1999/xhtml"/>

    <annotation>
        <documentation>
            <html:p>[Some documentation for my schema]</html:p>
        </documentation>
    </annotation>

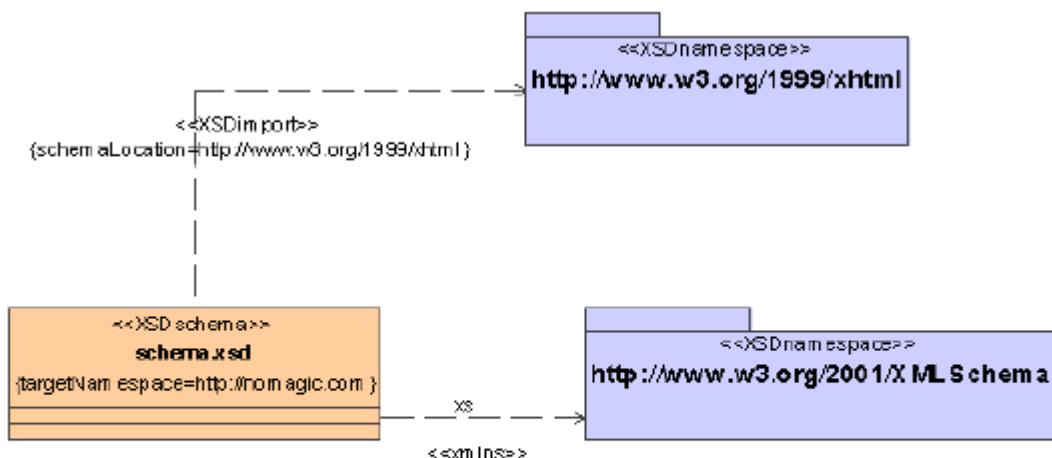
    . . .

    <complexType name="myType">
        <sequence>
            <element ref="html:p" minOccurs="0"/>
        </sequence>
        . .
    </complexType>

    <element name="myElt" type="my:myType"/>
</schema>
```

The treatment of references as ·**QNames**· implies that since (with the exception of the schema for schemas) the target namespace and the XML Schema namespace differ, without massive redeclaration of the default namespace either internal references to the names being defined in a schema document or the schema declaration and definition elements themselves must be explicitly qualified. This example takes the first option -- most other examples in this specification have taken the second.

import UML Model example



```

<xs:schema xmlns:nm = "http://nomagic.com" xmlns:xs =
"http://www.w3.org/2001/XMLSchema" targetNamespace = "http://nomagic.com" >
  <xs:import namespace = "http://www.w3.org/1999/xhtml" schemaLocation
= "http://www.w3.org/1999/xhtml" />
</xs:schema>
  
```

include

Maps to UML Component with stereotype XSDinclude. Component must be added into xsd file component.

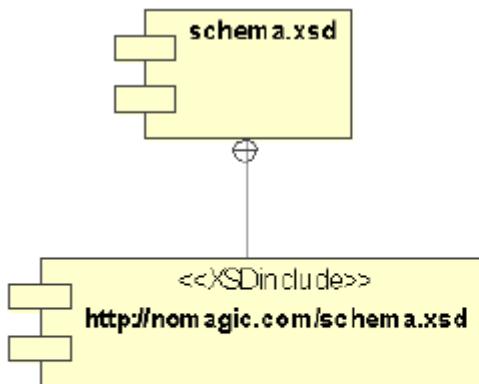
- annotation maps to UML Component documentation
- schemaLocation maps to UML Component name.

XML Representation Summary: include Element Information Item

```

<include
  id = ID
  schemaLocation = anyURI
  {any attributes with non-schema namespace . . .}
  Content: (annotation?)
/>
  
```

include UML Model example



```

<xs:schema xmlns:nm = "http://nomagic.com" xmlns:xs =
"http://www.w3.org/2001/XMLSchema" targetNamespace = "http://nomagic.com"
>
    <xs:include schemaLocation = "http://nomagic.com/schema.xsd" />
</xs:schema>

```

XML schema namespaces

Maps to UML Package with stereotype *XSDnamespace*. In order to define “*xmlns*” attribute in the schema file, Permission between *XSDnamespace* package and *XSDschema* class must be added into the model.

- The Permission name maps to namespace shortcut.

Example

```

<xs:schema
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://www.example.com/example">
</xs:schema>

```

The XML representation of the skeleton of a schema.

In order to generate such namespaces:

- UML model must have Package with name “*http://www.w3.org/2001/XMLSchema*”
- UML model must have Package with name “*http://www.example.com/example*”

- Permission with name “xs” must be added into model between XMLSchema Class and Package “<http://www.w3.org/2001/XMLSchema>”.
- Permission without name must be added into model between XMLSchema Class and Package “<http://www.w3.org/2001/XMLSchema>”.

XML schema namespaces UML Model example

For an example, see “schema UML Model example” on page 522.

XSD FILE CREATION WITH MAGICDRAW

New XML Schema code engineering language is added into MagicDraw engine in order to generate/reverse XSD files.

Code Engineering Project of this language has such Language Properties:

- Default Target XSD File Name – the name of default xsd file.

This CE set has one RT Component by default. Selected by user classes will be added into this component. Component is mapped to xsd file. RT Component does not have inner RT Components.

TODO in Magicdraw

add encoding property to project

WSDL

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
		Component	extension - string name - string targetNamespace - string	
Definition	WSDLdefinitions	Class		
Message	WSDLmessage	Interface		
Port Type	WSDLporrtype	Class		
Binding	WSDLbinding	Instance	extension - string	
Port	WSDLport	Specification		
Service	WSDLservice	Component	extension - string	
Type	WSDLtypes	Component	extension - string	
	WSDLimport	ElementImport, PackageImport		
	xmns	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	

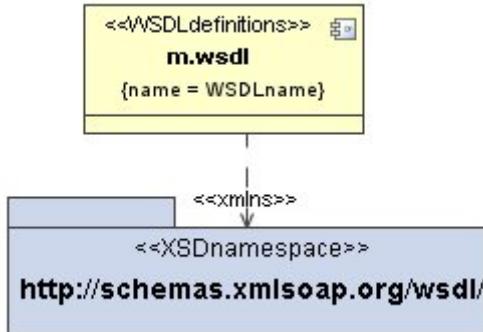
Definitions

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:

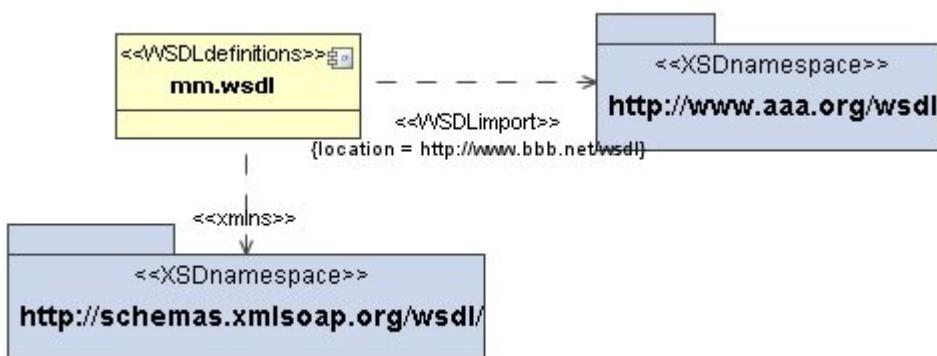


Import, namespace

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/">

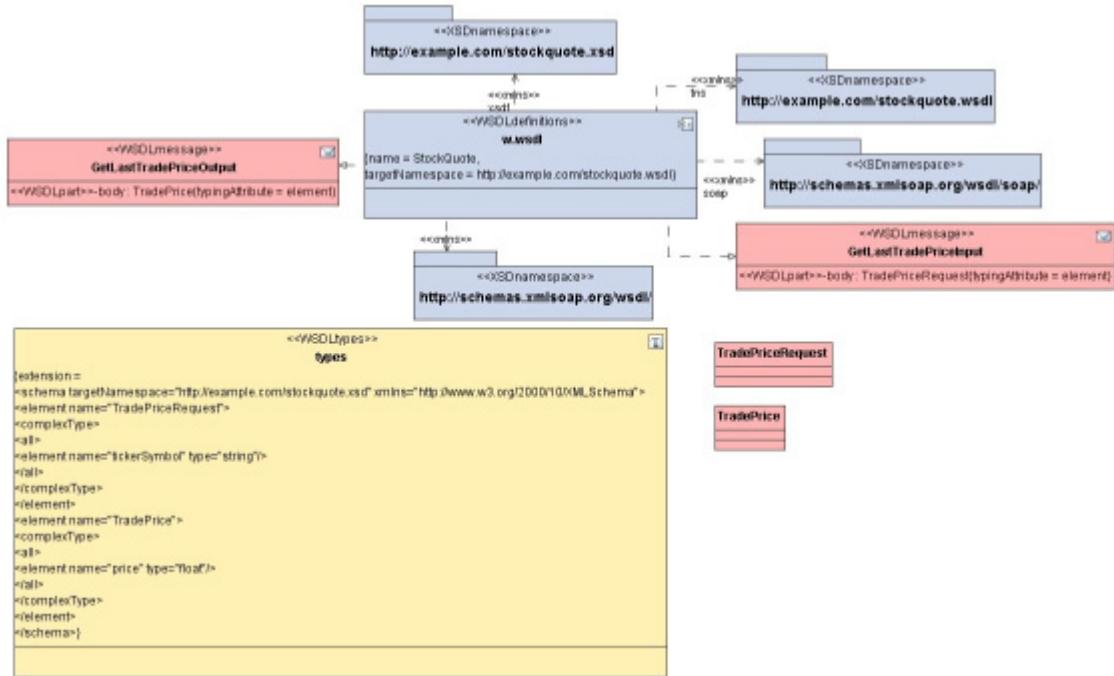
  <types>
    <schema targetNamespace="http://example.com/stockquote.xsd"
      xmlns="http://www.w3.org/2000/10/XMLSchema">
      <element name="TradePriceRequest">
        <complexType>
          <all>
            <element name="tickerSymbol" type="string"/>
          </all>
        </complexType>
      </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">

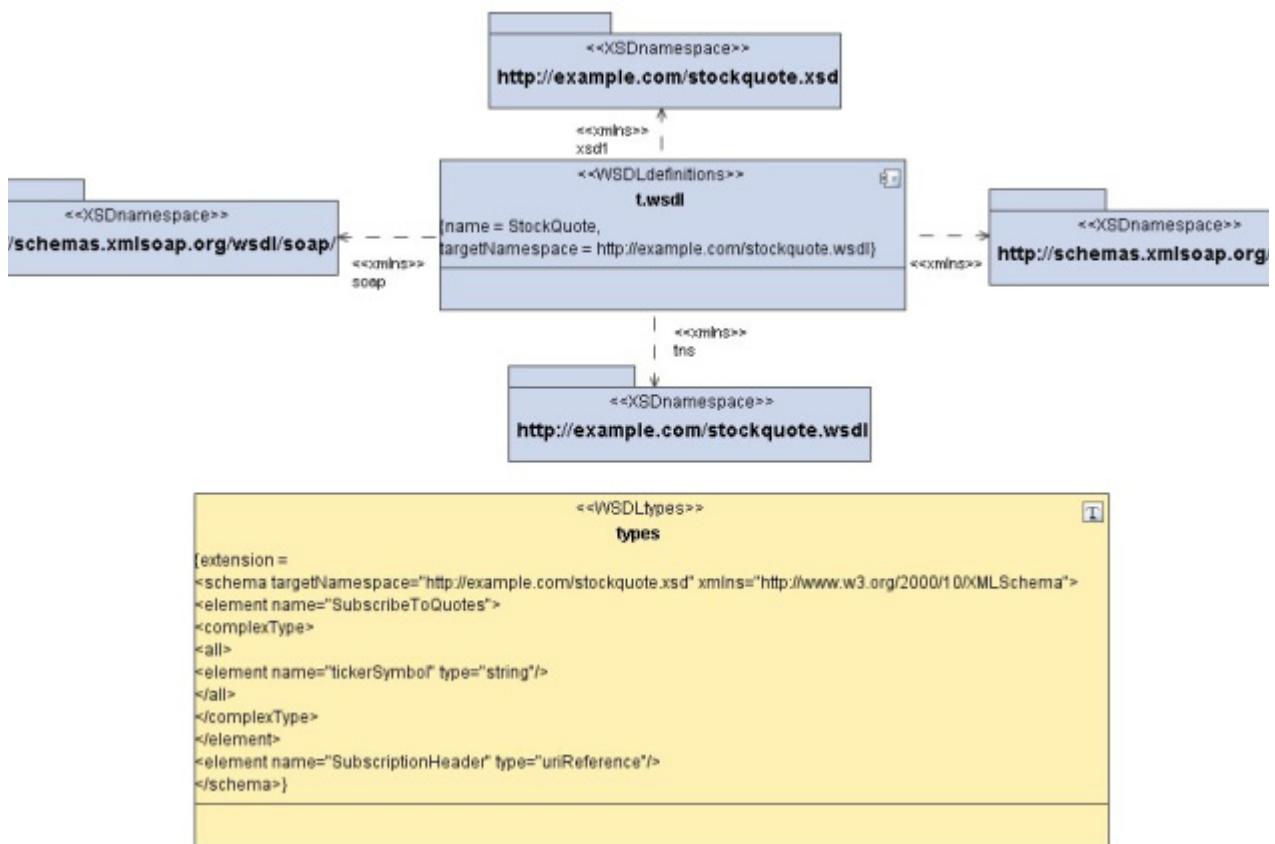
```

```

<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:xsd1="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="xsd1:SubscribeToQuotes"/>
        <part name="subscribeheader" element="xsd1: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"/>
            </input>
        </operation>
    </binding>
</definitions>
```

```

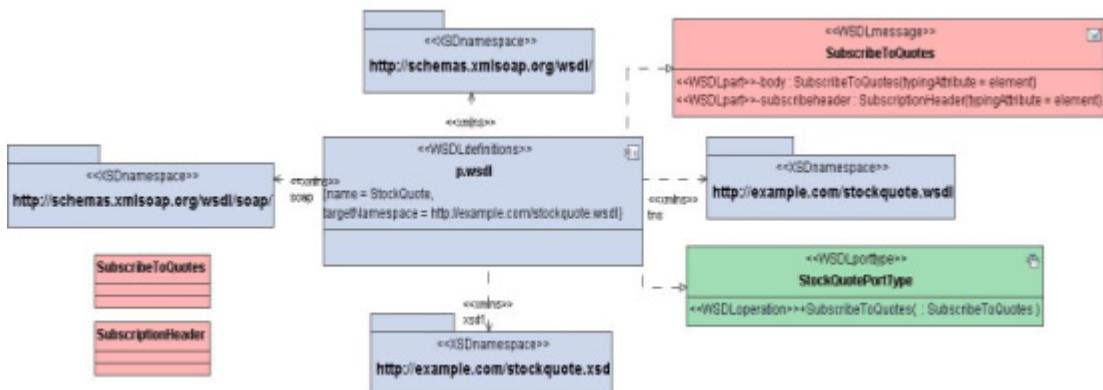
<soap:header message="tns:SubscribeToQuotes"
part="subscribeheader" use="literal"/>
</input>
</operation>
</binding>

<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>

    <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"
```

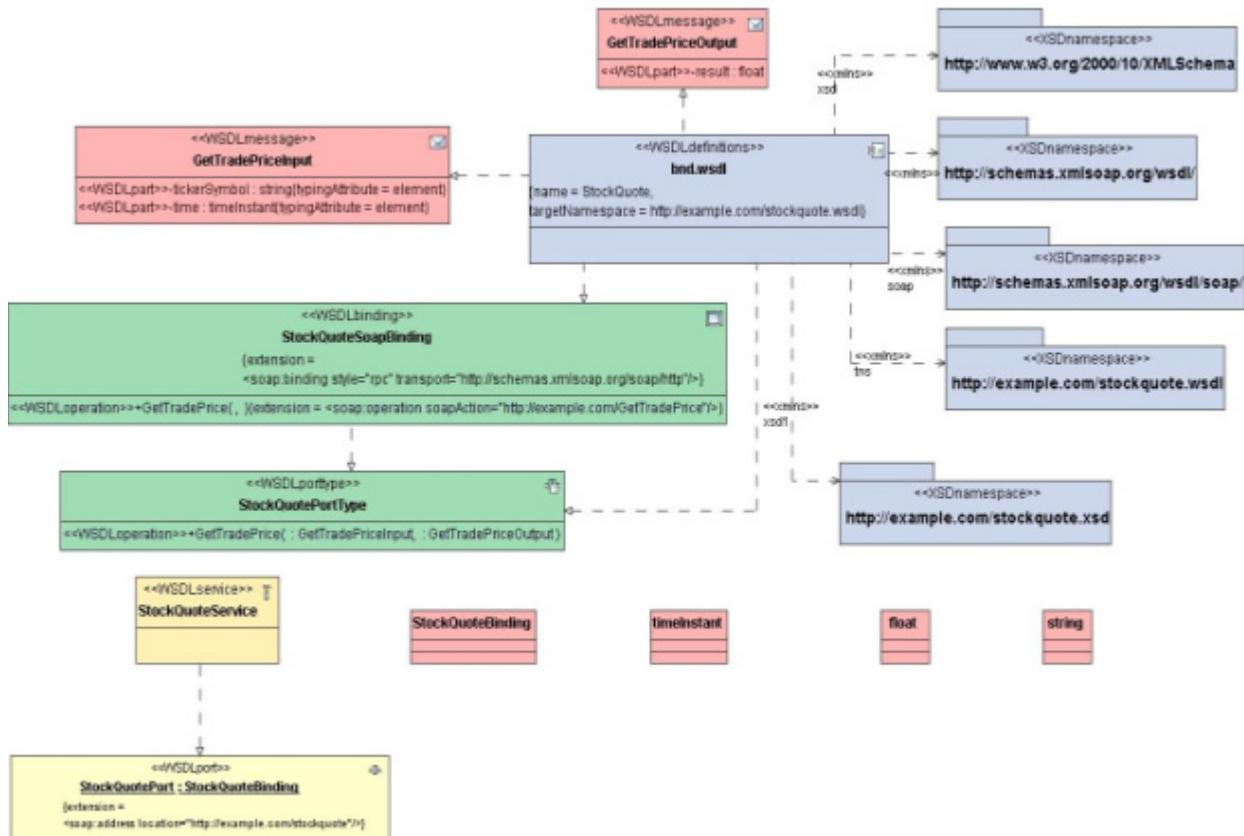
```

        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>

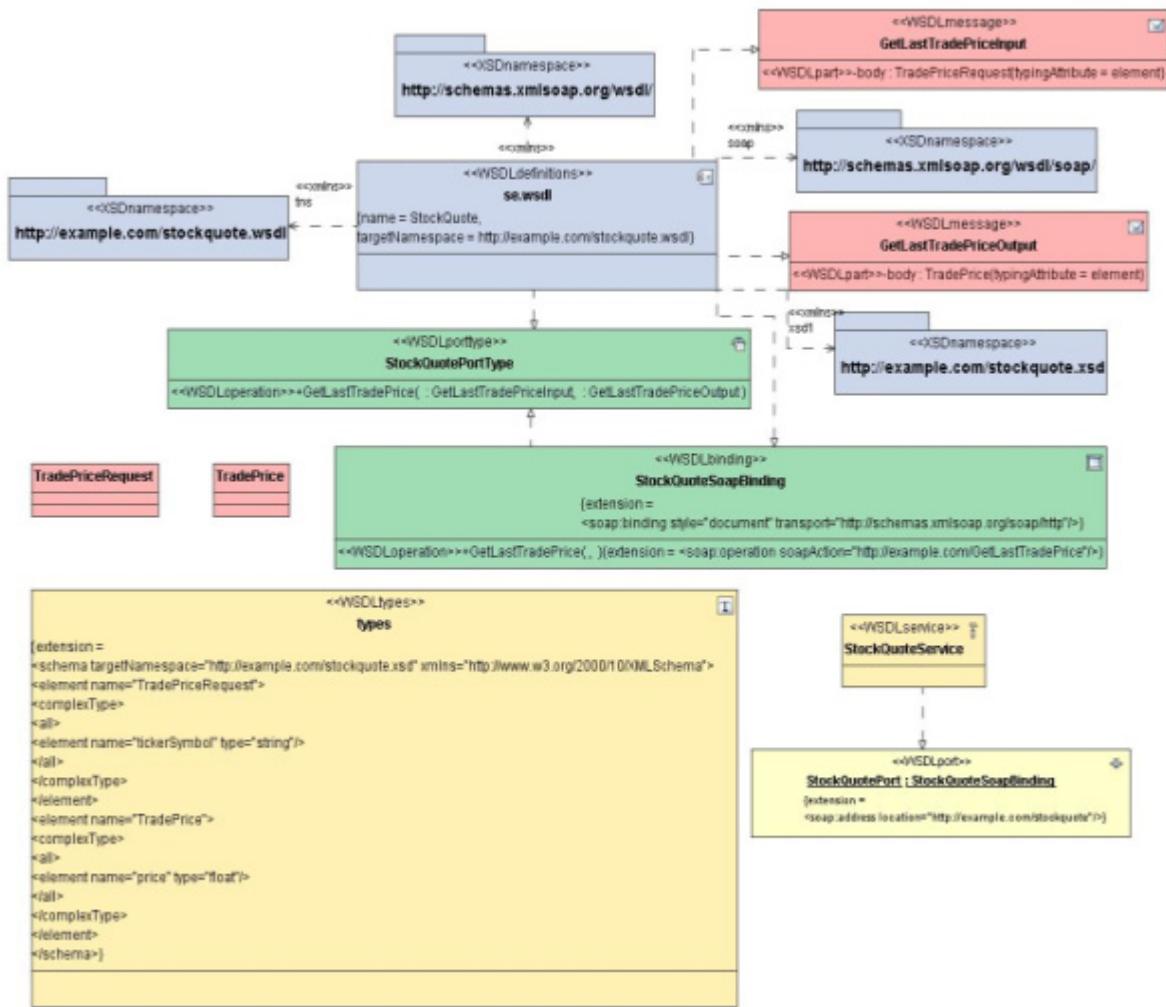
<portType name="StockQuotePortType">
  <operation name="GetLastTradePrice">
    <input message="tns:GetLastTradePriceInput"/>
    <output message="tns:GetLastTradePriceOutput"/>
  </operation>
</portType>

<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>

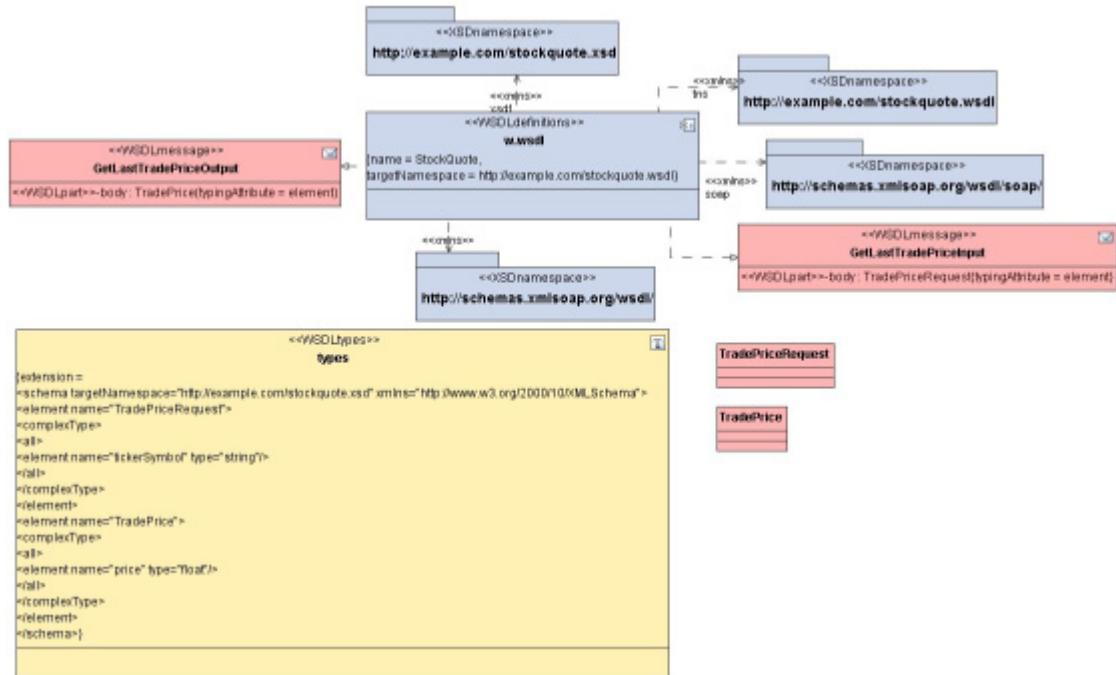
    <service name="Hello_Service">
        <documentation>WSDL File for HelloService</documentation>
```

```

<port binding="tns:Hello_Binding" name="Hello_Port">
  <soap:address
    location="http://localhost:8080/soap/servlet/rpcrouter"/>
</port>
</service>
</definitions>

```

Reversed UML model example:



TRANSFORMATIONS

There are several kinds of transformations, presented by MagicDraw UML:

- UML to DDL
- DDL to UML
- Generic DDL to Oracle DDL
- UML to XML Schema
- XML Schema to UML

Model Transformations Wizard is the main component covering all functionality and logics of model transformations. The wizard is used for creation of new transformations.

To open **Model Transformations Wizard**

- From the **Tools** main menu, choose **Model Transformations**.
- Select the package(s). From the shortcut menu, choose **Tools** and then choose

Transform.

NOTE: More information about this wizard, see **Model Transformation Wizard** subsection in the *MagicDraw User Manual, Tools* section.

When transforming the model, there is a default type mapping applied to each of this transformation. This mapping is used to map data types from one domain into appropriate data types in another domain.

To change default mapping rules

1. From the appropriate **Type Map** profile, choose **Modules** and then **Open Module as Project**.
2. Make changes in the default type mapping rules (add/delete/change dependencies of data types).
3. Save changes.
4. In the previous project, from the appropriate **Type Map** profile, choose **Modules** and then **Reload Module**. Changed type mapping will be applied on every next that kind transformation.

NOTE:

More information how to create your own transformation rules or change mapping behavior, see **Model Transformation Mapping** paragraph in the *MagicDraw User Manual, Tools* section, *Model Transformations Wizard* subsection

-or-

watch the the model transformation viewlet at www.magicdraw.com/viewlets for information about how to set up the mapping.

UML to DDL transformation

There are two kinds of UML to DDL transformation:

- UML to Generic DDL
- UML to Oracle DDL

According to selected transformation kind, the Class diagrams will be transformed to the Generic DDL or Oracle DDL diagram.

Type mapping

If there are types specified in the UML model for some elements, after transformation UML types should be converted to DDL types. Because of that, there is a type mapping from UML types to DDL types.

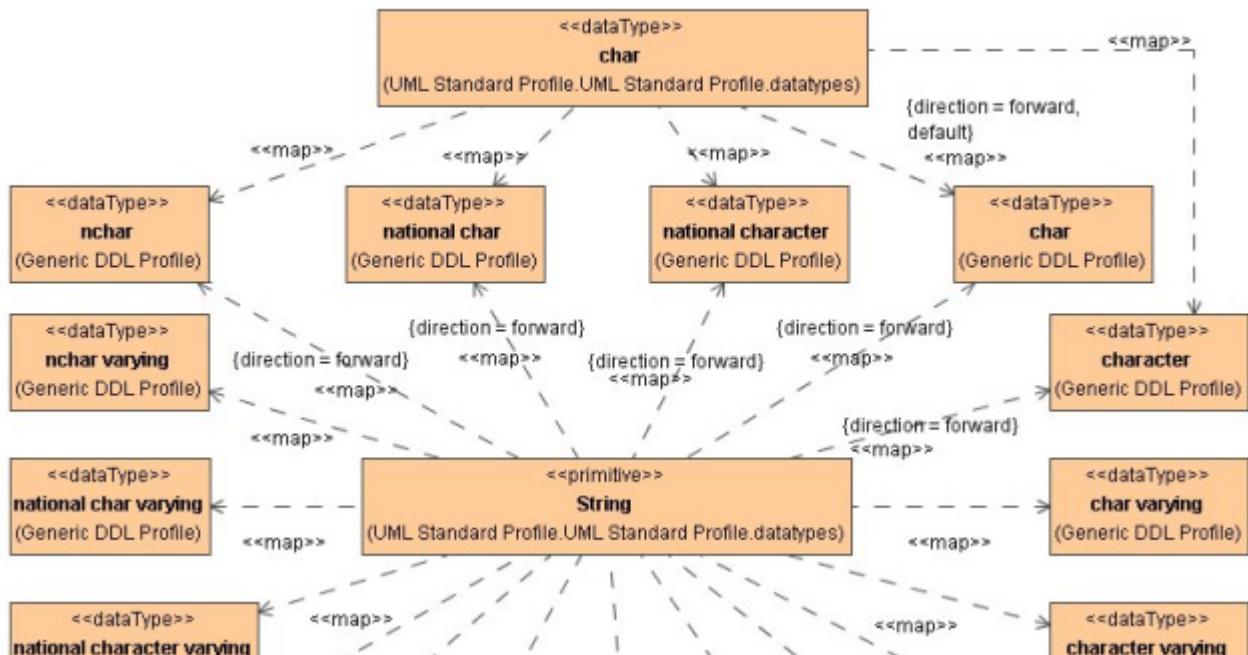
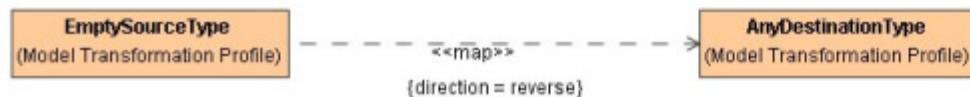
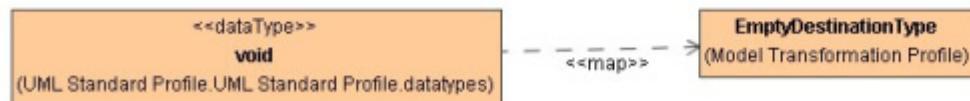
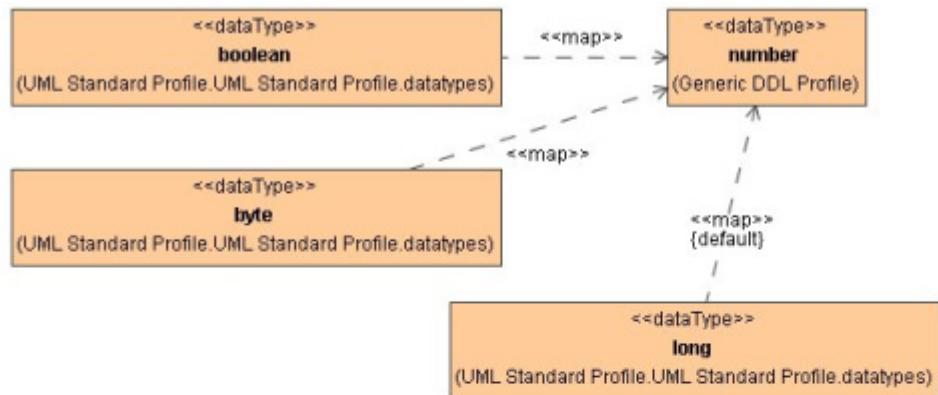
Mapping rules are based on dependencies, which contains the **UML to Generic DDL Type Map** profile (or **UML to Oracle DDL Type Map** profile).

UML to Generic DDL type map

Double-click the *UML to Generic DDL Type Map Diagram* (stored in UML to Generic DDL Type Map profile) to open it and see default type mapping rules, applied for this transformation.

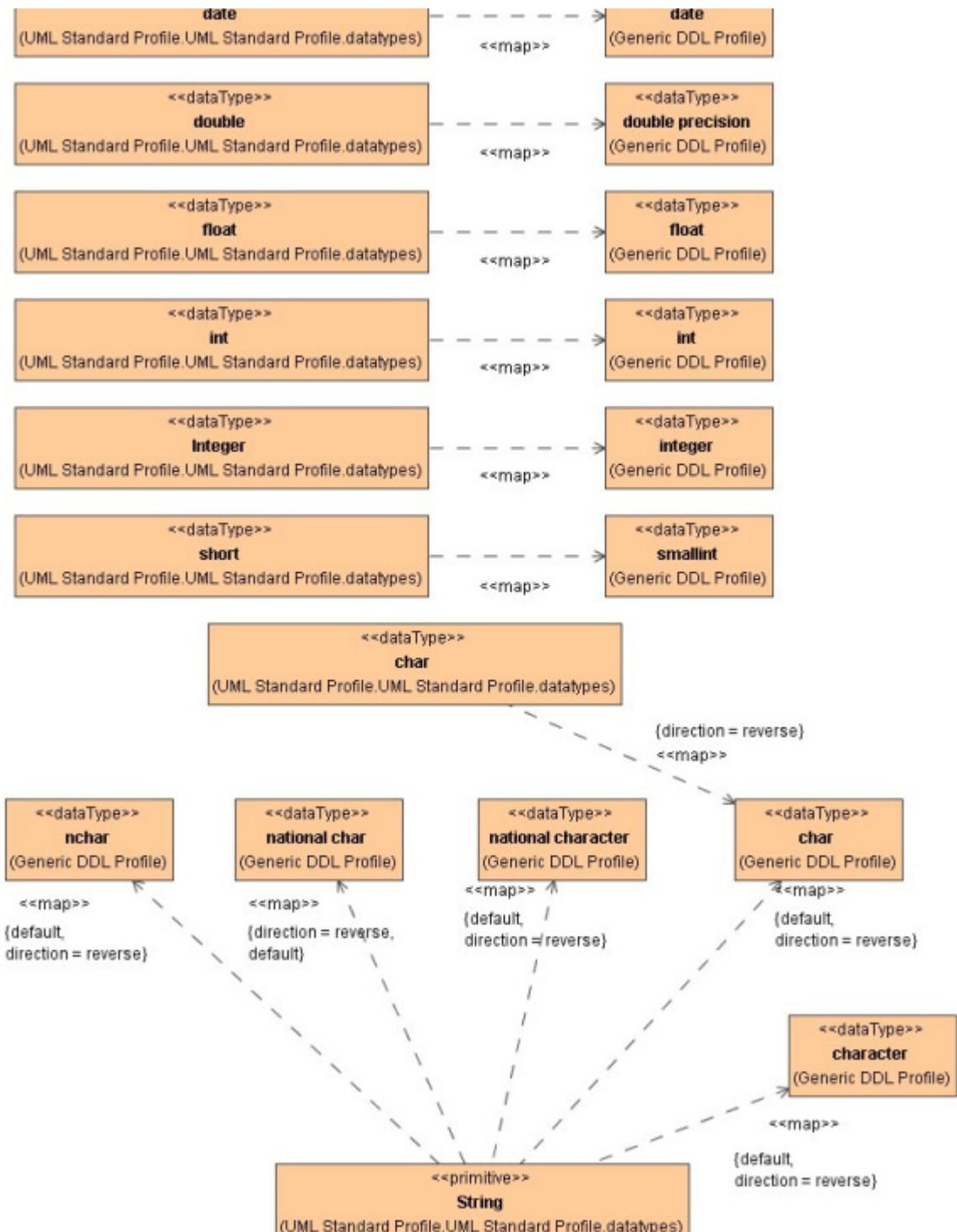
TRANSFORMATIONS

UML to DDL transformation



TRANSFORMATIONS

UML to DDL transformation



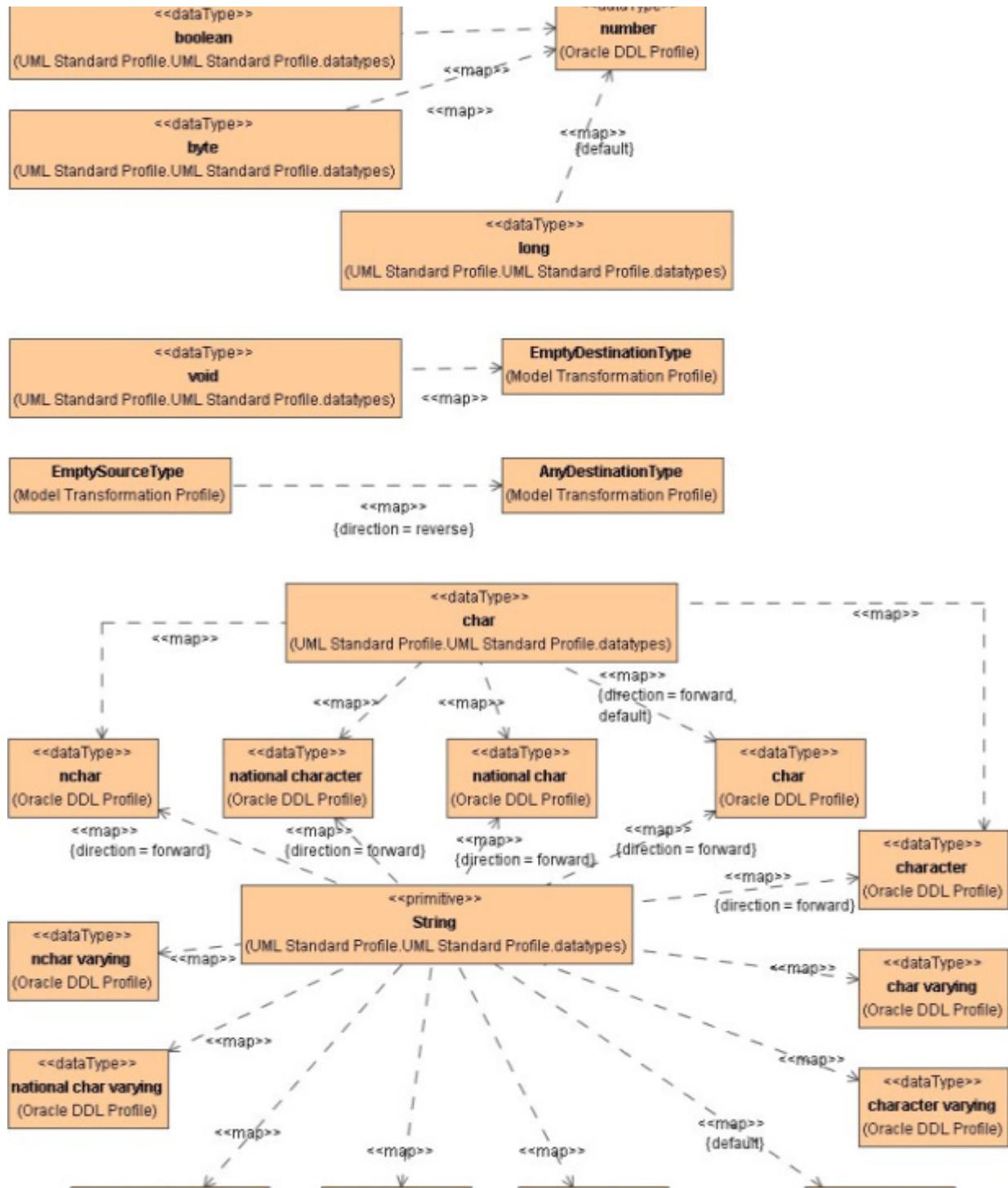
UML to Oracle DDL type map

Double-click the *UML to Oracle DDL Type Map Diagram* (stored in UML to Oracle DDL Type Map profile) to

open it and see default type mapping rules, applied for this transformation.

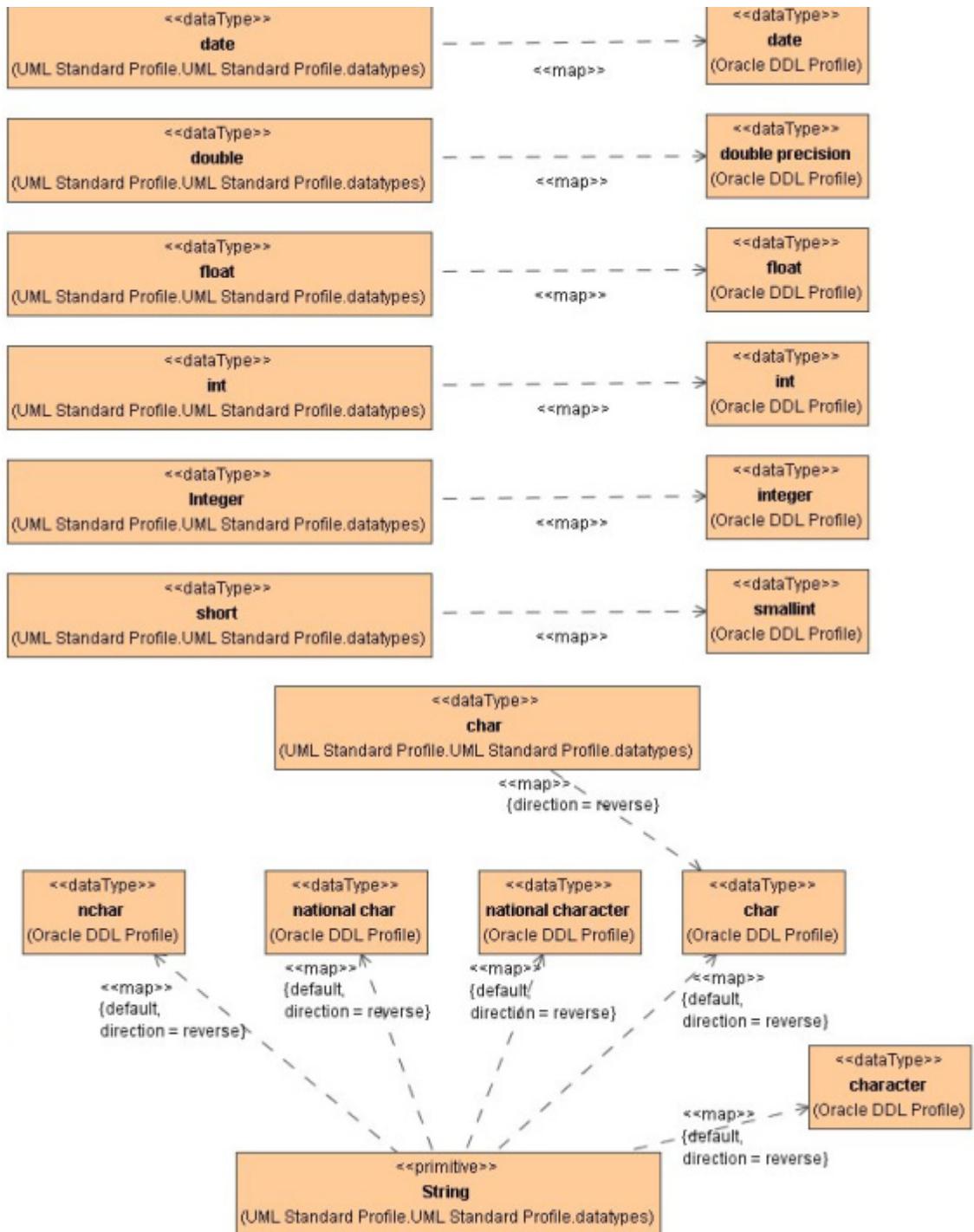
TRANSFORMATIONS

UML to DDL transformation



TRANSFORMATIONS

UML to DDL transformation



Transformation results

All the relationships except associations and dependencies are discarded.

Methods of the classes are discarded.

For each class the <<table>> stereotype is applied.

There are additional properties to choose for UML to DDL transformation in the **Model Transformation Wizard** (for more information about Model Transformation Wizard, see *MagicDraw User Manual*, *Tools* section, *Model Transformations Wizard* subsection.)

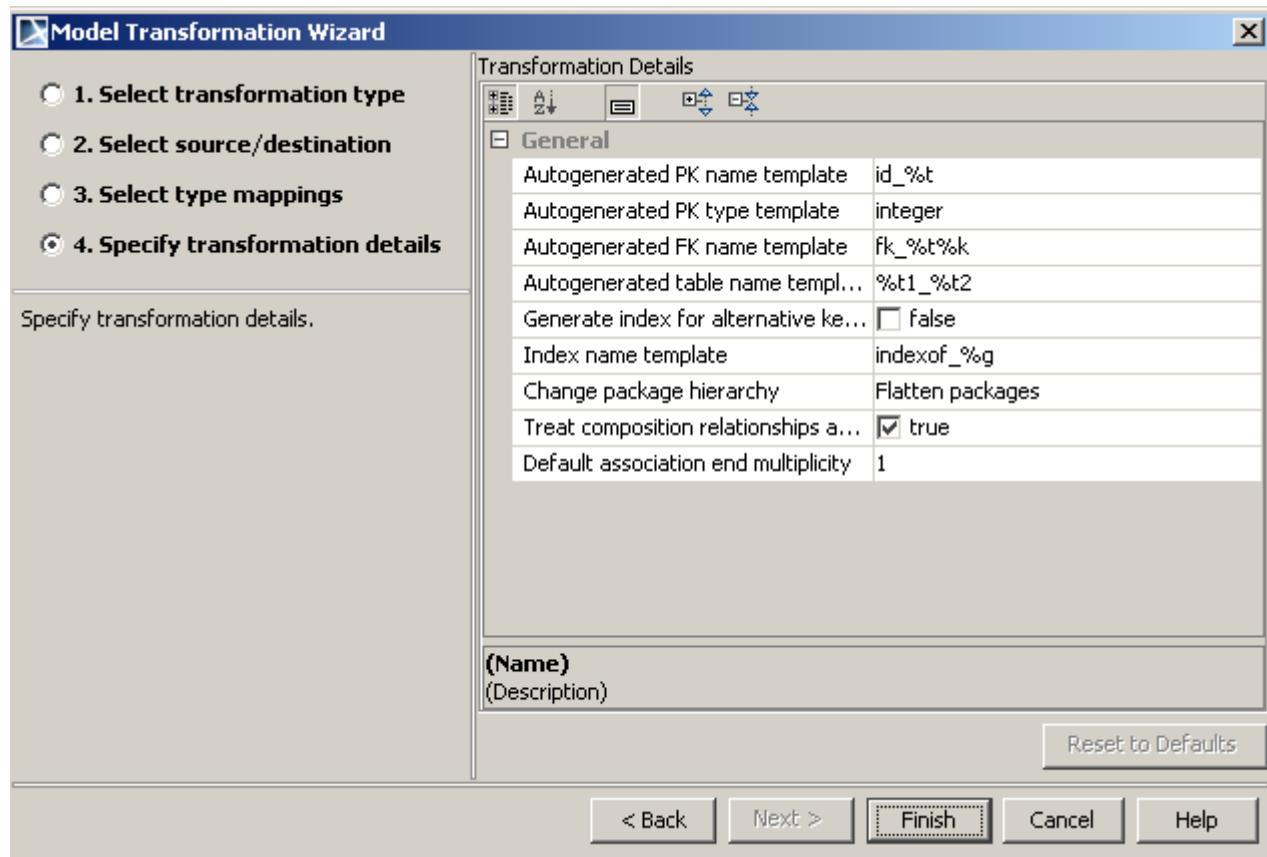


Figure 1 -- Model Transformation Wizard for DDL to UML transformation, Specify Transformation Details

Tab name	Box	Function
Autogenerated PK name template	Text box	If the class has no PK column in the ER model, this transformation parameter for autogenerated column name will generate the PK. You may specify the pattern for the PK name. Default "id_%t", where %t is replaced by the name of the table.
Autogenerated PK type template	Combo box	Specifies the type of the autogenerated PKs. Default: integer (from Generic DDL profile).
Autogenerated FK name template	Text Box	The foreign keys are automatically generated to implement the relationships between classes. This transformation parameter autogenerates FK name. You may specify the pattern for the name. Default: "fk_%t%k", where %t is replaced by the name of the table, the foreign key is pointing. The %k is replaced by the key name, to which this foreign key points. The %r is replaced by the name of the relationship, which is realized with this foreign key.
Autogenerated table name template	Text box	This transformation parameter autogenerates table name. You may specify the pattern for the name. Default "%t1_%t2", where %t1 is replaced by the name of the first table, %t2 - second table. The %r pattern (name of relationship) is also supported.
Generated index for alternative keys	Check box	If "true", generates index for <<AK>>. Default: false
Index name template	Text box	If the above option is set to "true", you may choose the template for the index name. Template may contain %g pattern, which will be replaced with AK group name. Default: indexof_%g
Change package hierarchy	Combo box	Choose option for packages from transformation source: to strip all the package hierarchy, or flatten the package hierarchy down to the first level where each package is transformed into the schema. Default: Flatten packages

Tab name	Box	Function
Treat composition relationship as identifying	<input type="checkbox"/>	If this option is set to "true", the composition associations are treated as if the <>identifying>> stereotype were applied to them. Default: true
Default association end multiplicity	<input type="button" value="Combo box"/>	If multiplicity was not specified in model, defined multiplicity will be set after transformation. Default: 1

Transformation splits many-to-many relationship into two relationships with intermediate table.

DDL to UML transformation

The DDL diagrams will be transformed into the class diagrams. DDL to UML transformation can be applied to both the Generic DDL models and Oracle DDL models.

Type mapping

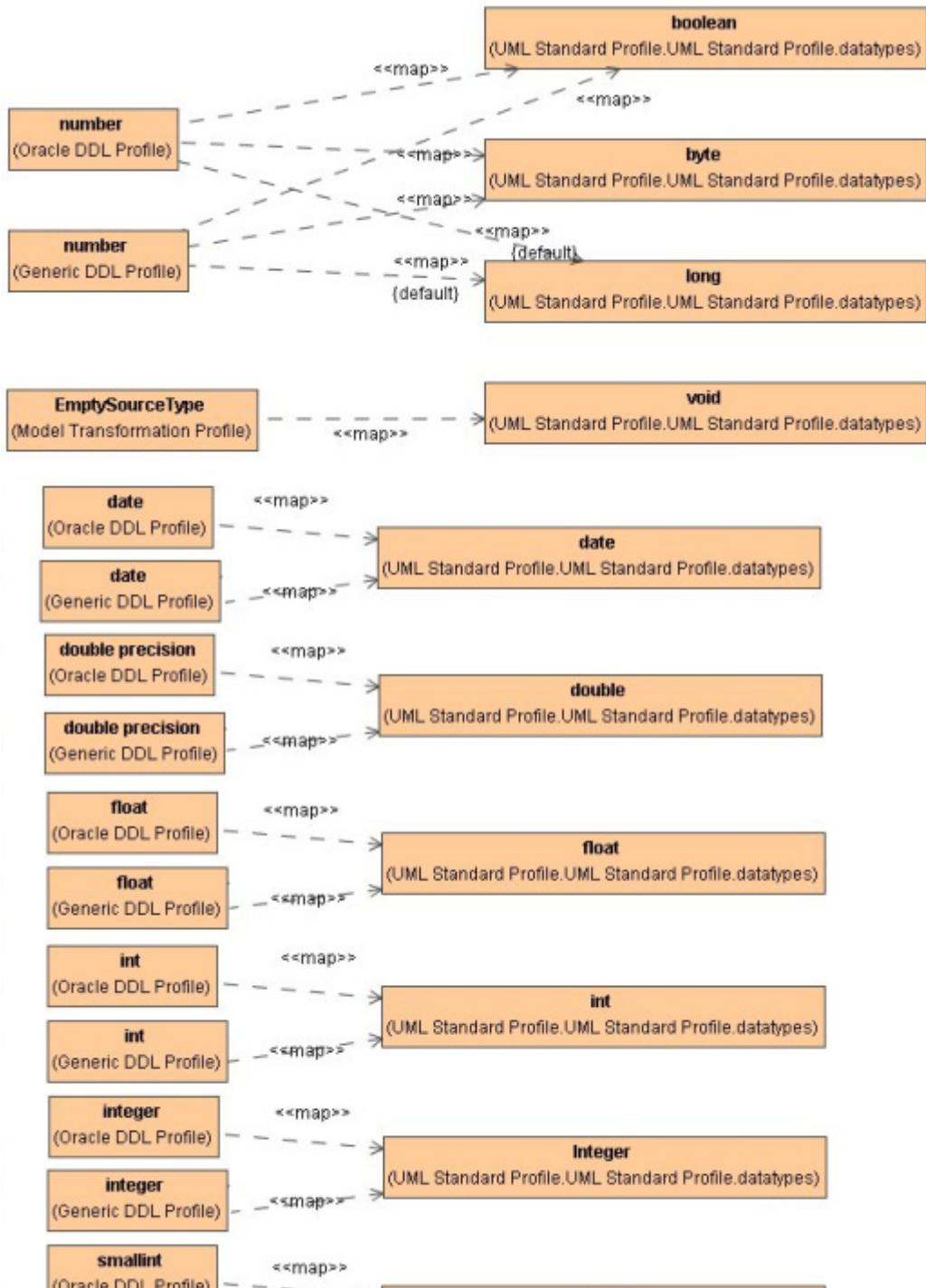
If there are types specified in the DDL model for elements, after transformation DDL types should be converted to UML types. Because of that, there is a type mapping from DDL Generic or Oracle types to UML types.

Mapping rules are based on dependencies, which contains the **DDL to UML Type Map** profile.

Double-click the *DDL to UML Type Map Diagram* (stored in DDL to UML Type Map profile) to open it and see default type mapping rules, applied for this transformation.

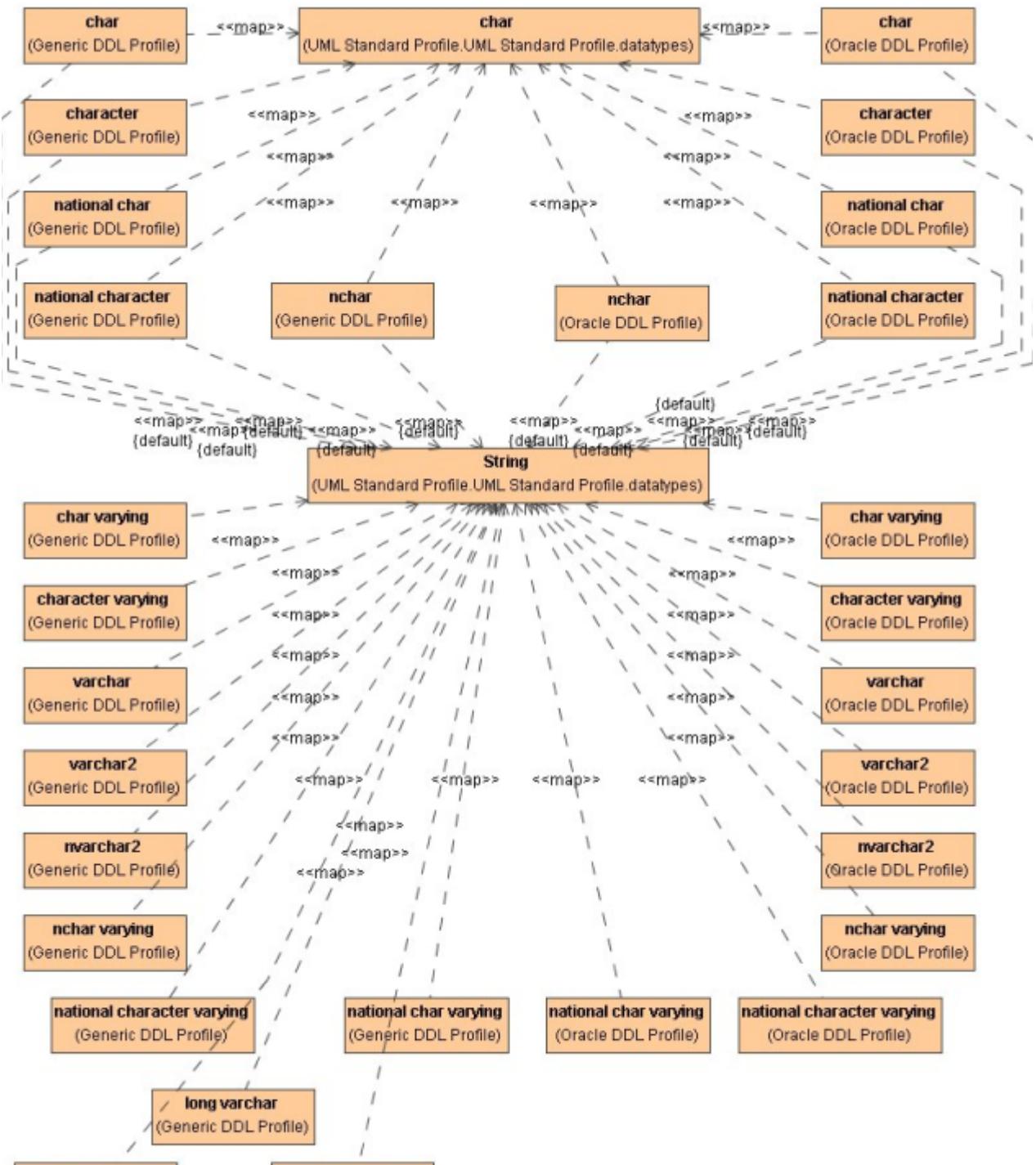
TRANSFORMATIONS

DDL to UML transformation



TRANSFORMATIONS

DDL to UML transformation



Transformation results

The DDL stereotypes are discarded from tables, views, fields, associations (except the PK stereotype).

Views are discarded in transformed class diagram.

There are additional properties to choose for DDL to UML transformation in the **Model Transformation Wizard** (for more information about Model Transformation Wizard, see *MagicDraw User Manual*, *Tools* section, *Model Transformations Wizard* subsection.)

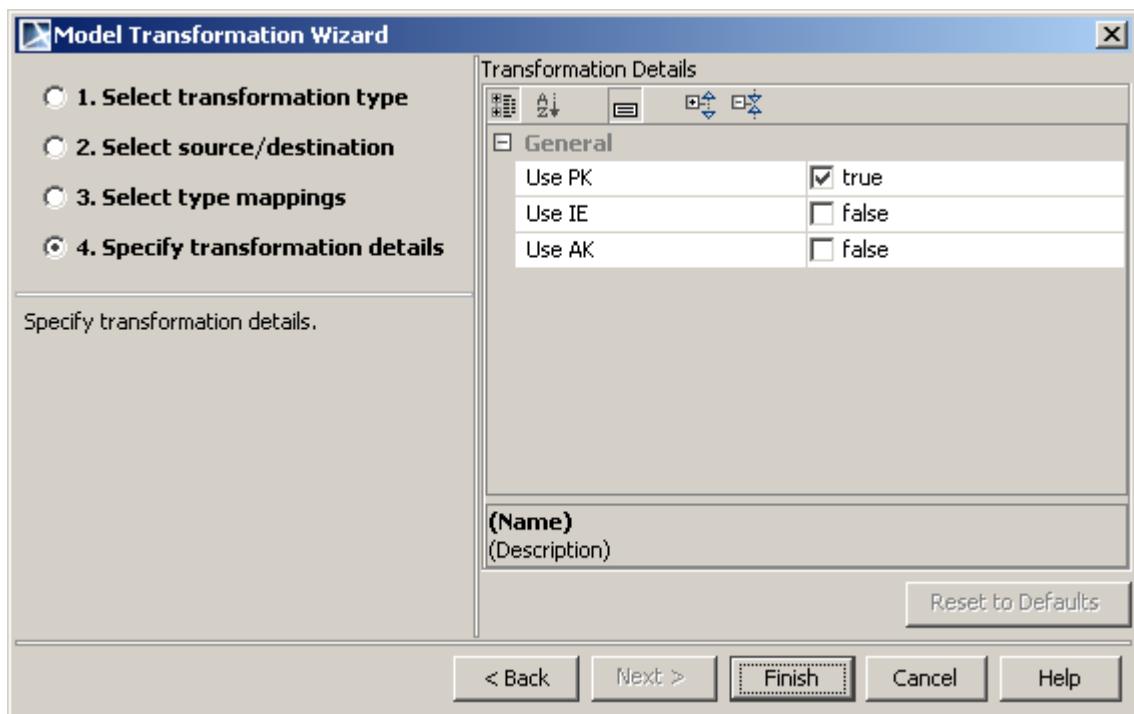


Figure 2 -- Model Transformation Wizard for UML to DDL transformation, Specify Transformation Details

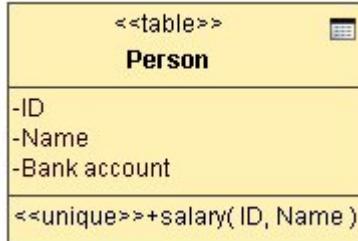
Tab name	Box	Function
Use PK	Check box	If set to “true”, appropriate columns with primary key stereotype is marked after transformation.
Use IE	Check box	If set to “true”, indexed columns with inverted entity stereotype is marked after transformation.
Use AK	Check Box	If set to “true”, unique columns with alternative key stereotype is marked after transformation.

The <<IE>> stereotype are applied to the columns in UML model from *indexes* in the DDL.

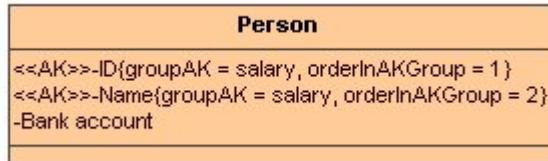
The <<AK>> stereotypes are applied to the columns in UML model from *unique* constraints in the DDL.

If the *unique* or *index* of the DDL contains more than one column, the "group" tag is created on the corresponding columns. The value of the tag is name of the *unique/index*.

If the PK, *unique* constraint or *index* of the DDL contains more than one column, the "orderInXXGroup" tag is created on the corresponding columns. The value of the tag is the place number of the column in the PK, unique constraint or index (first column gets tag value=1, second column - 2, etc). Example:



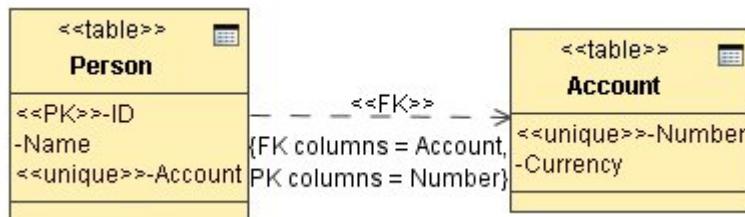
After transformation:



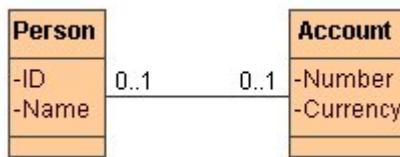
There are some foreign key cases, when after transformation, association with multiplicities are created in class diagram:

To transform foreign key, when <<unique>> stereotype is set

Before transformation:

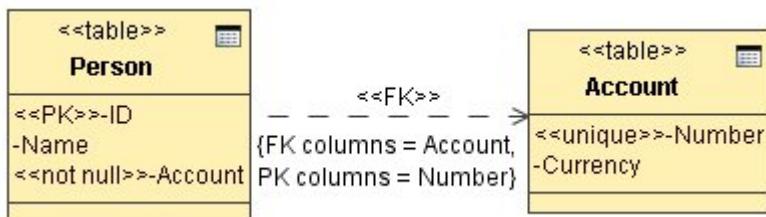


After transformation:

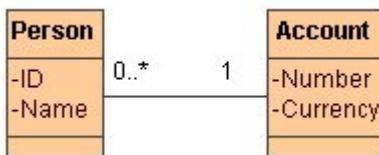


To transform foreign key, when <<not null>> stereotype is set

Before transformation:

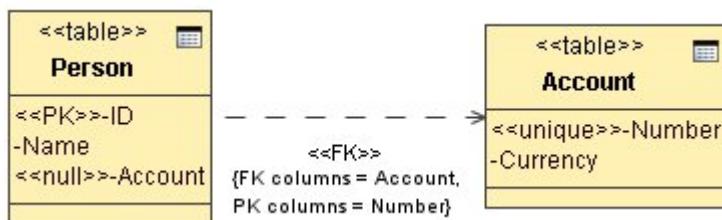


After transformation:

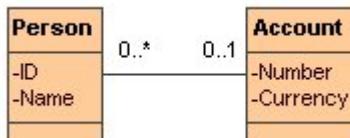


To transform foreign key, when <<null>> stereotype is set

Before transformation:

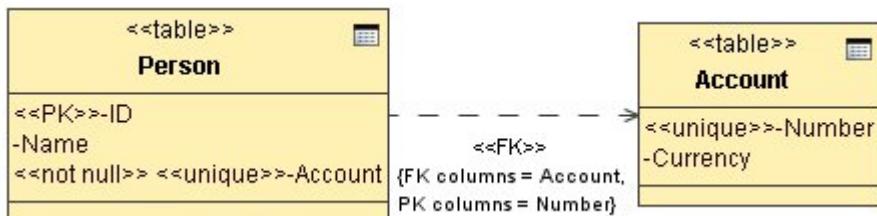


After transformation:

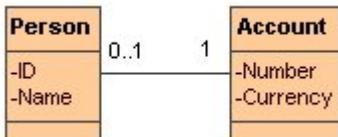


To transform foreign key, when <<unique>> and <<not null>> stereotypes are set

Before transformation:



After transformation:



Generic DDL to Oracle DDL transformation

The Generic DDL to Oracle DDL transformation is used for legacy purposes. Transformation replaces stereotypes from Generic DDL profile with corresponding stereotypes from Oracle DDL profile. MagicDraw has a separate profile for Oracle DDL modeling. Oracle scripts are generated and reverse-engineered using this profile. Users, who previously used Generic DDL profile to model Oracle databases, should now migrate to new profile. The Generic DDL to Oracle DDL transformation provides an easy solution for migration: just apply this transformation to your Generic DDL model (choose to transform in-place in the second step of the wizard).

Generic DDL to Oracle DDL type mapping

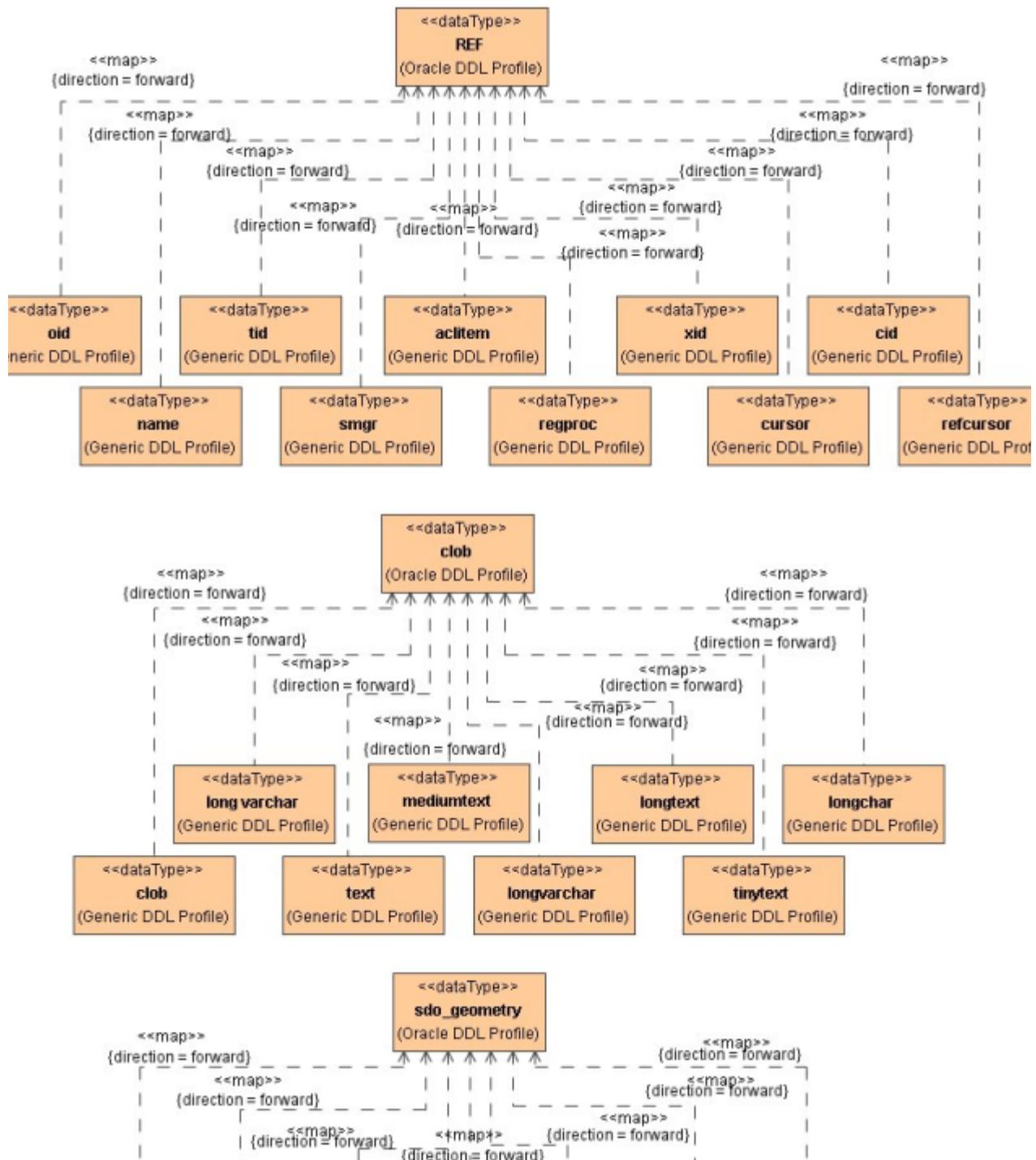
If there are types specified in the Generic DDL model for elements, after transformation DDL types should be converted to Oracle DDL types. Because of that, there is a type mapping from DDL Generic to Oracle types.

Mapping rules are based on dependencies, which contains the **Generic To Oracle DDL Type Map** profile.

Double-click the class diagram (stored in Generic To Oracle DDL Type Map profile) to open it and see default type mapping rules, applied for this transformation.

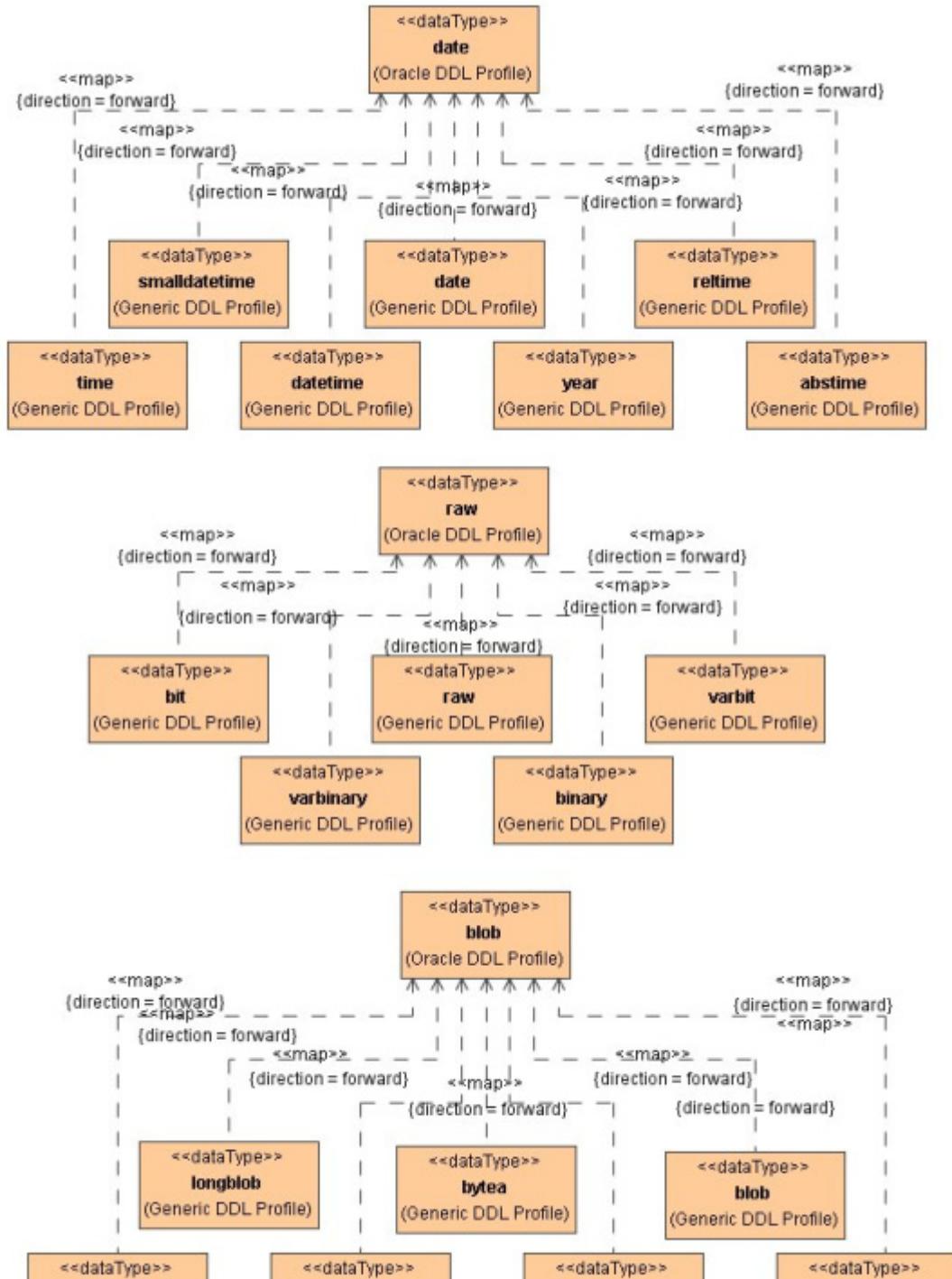
TRANSFORMATIONS

Generic DDL to Oracle DDL transformation



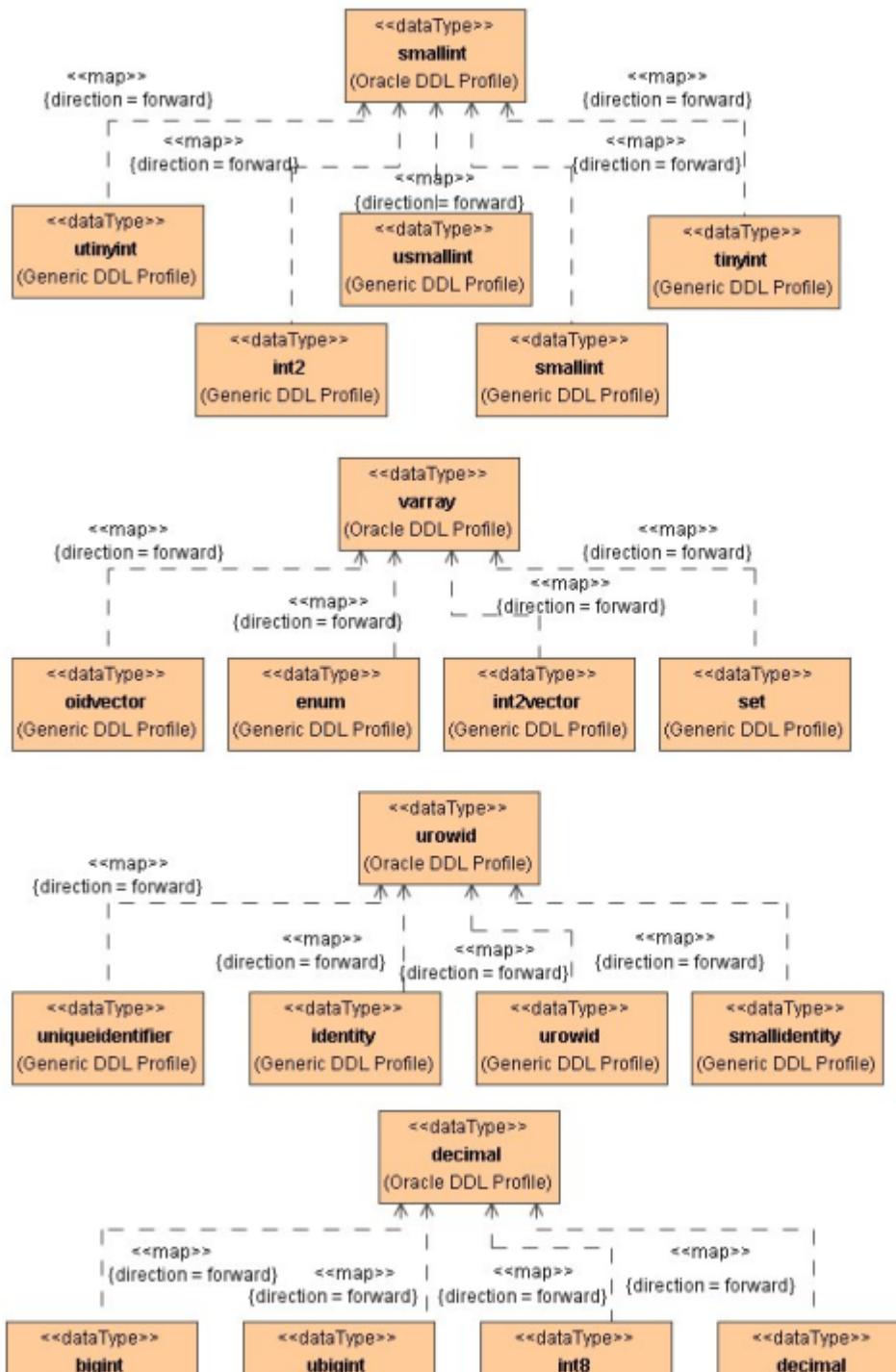
TRANSFORMATIONS

Generic DDL to Oracle DDL transformation



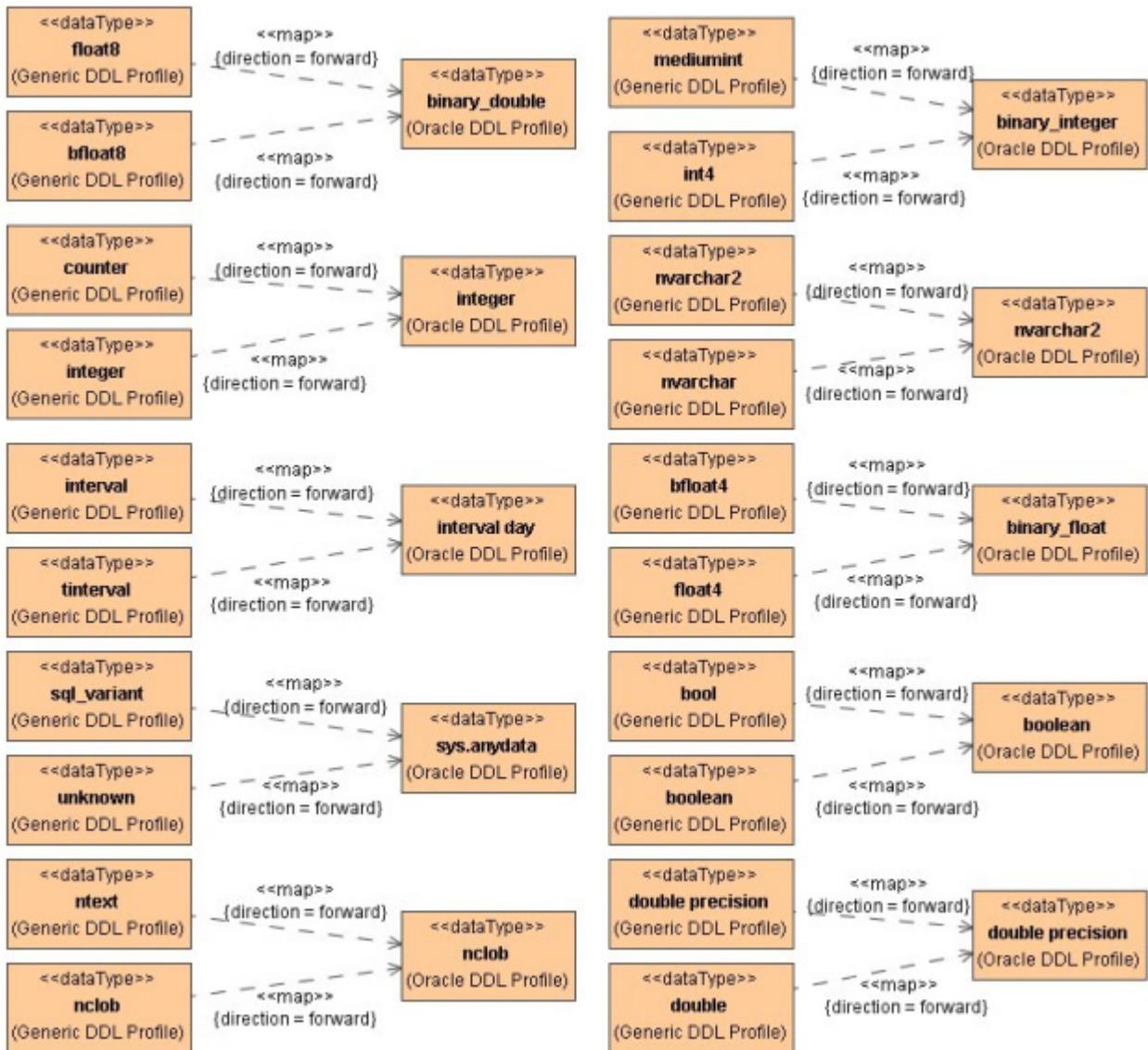
TRANSFORMATIONS

Generic DDL to Oracle DDL transformation



TRANSFORMATIONS

Generic DDL to Oracle DDL transformation



TRANSFORMATIONS

Generic DDL to Oracle DDL transformation



UML to XML Schema transformation

The UML to XML Schema transformation helps to create the equivalent XML schema model from the given UML model.

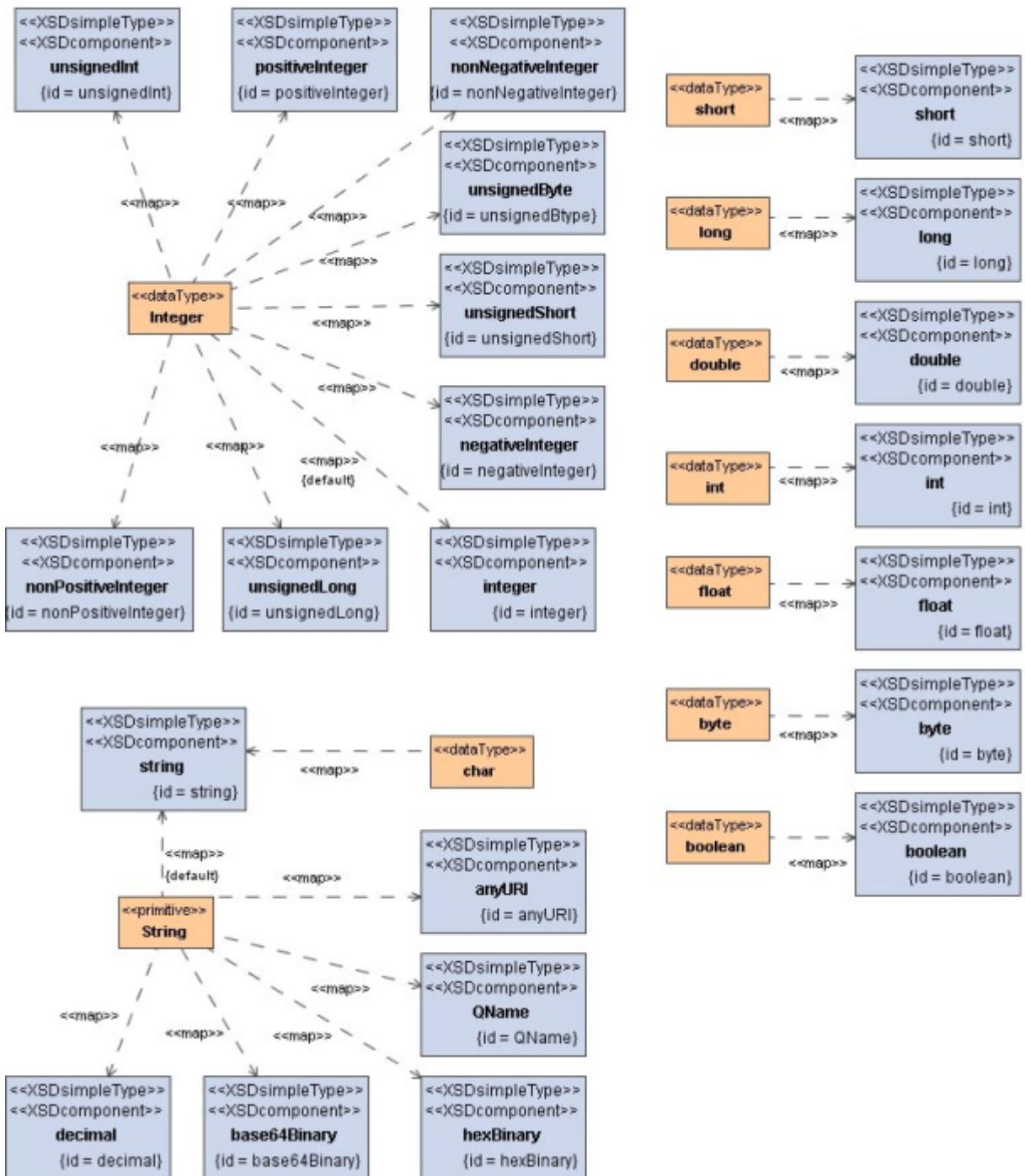
Basically this transformation is copying of source UML model, and then applying the necessary stereotypes according to the XML schema modeling rules.

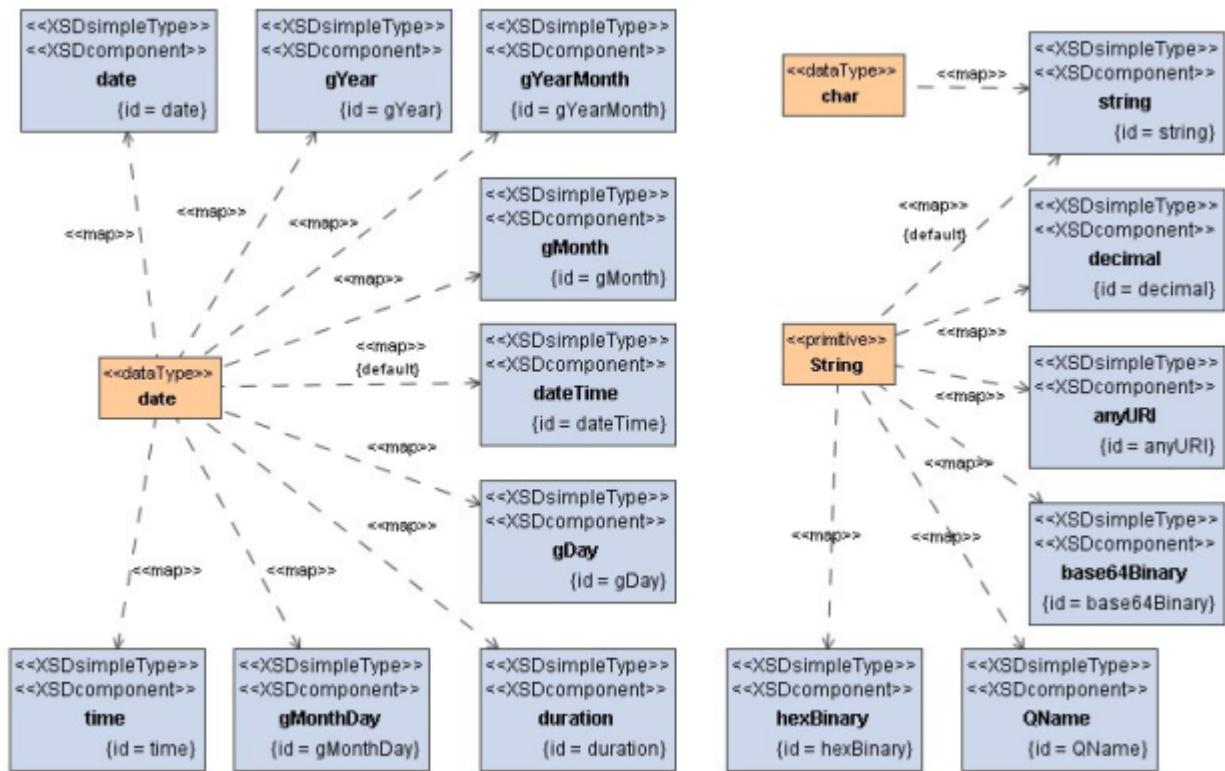
Type mapping

This type map stores mapping between primitive UML data types and primitive XML Schema data types.

TRANSFORMATIONS

UML to XML Schema transformation





Transformation results

For each class in the transformation destination set, the `<<XSDcomplexType>>` stereotype is applied, unless this class is derived from the simple XML type (i.e. one of the basic types, or type, stereotyped with `XSDsimpleType`). In that case `<<XSDsimpleType>>` stereotype is applied.

If the class is derived from other class, which is stereotyped as `<<XSDcomplexType>>`, additionally apply `<<XSDcomplexContent>>` stereotype is applied on this class and `<<XSDextension>>` on the corresponding generalization relationship.

If the class is derived from other class, which is stereotyped as `<<XSDsimpleType>>`, additionally `<<XSDrestriction>>` stereotype is applied on the corresponding generalization relationship.

If the class is not derived from anything, and has attributes with `XSElement` tag, `<<XSDcomplexContent>>` stereotype is applied on this class.

If the class is not derived from anything, and has no attributes with `XSDelement` tag, no `<<XXXXContent>>` stereotype is applied on this class - the class has an empty content.

The UML datatypes in the transformation source set are transformed into the classes with the `<<XSDsimpleType>>` stereotype - unless after the type map this class appears to be derived from class with `<<XSDcomplexType>>` stereotype. Then the `<<XSDcomplexType>>` stereotype is used.

For each attribute of the class, which is NOT of the simple XML type (i.e. one of the basic types, or type, stereotyped with `<<XSDsimpleType>>`) or has multiplicity > 1, the `<<XSDelement>>` stereotype is applied.

For each composition association, linking 2 classes stereotyped as XML schema types, stereotype on the association end is applied, the same as the rules for attributes.

Enumerations in the UML model are transformed into the enumerations in the XML Schema model (classes with `<<XSDsimpleType>>` stereotype derived by restriction from XML string type, where all the elements of the original enumeration are converted into the attributes with `<<XSDenumeration>>` stereotype).

For each package in the transformation set, `<<XSDnamespace>>` stereotype is applied.

In each package, one additional class for the XML schema is created. The name of the schema class is constructed by taking the name of the package and then appending the ".xsd" to it (e.g. if the package in the source model set is named "user", then name the schema class "user.xsd" in the destination package).

The targetNamespace value is added to the schema class, with the name of its parent (e.g. if the schema is placed in the "http://magicdraw.com/User" package, the targetNamespace=" http://magicdraw.com/User" is set on the schema class).

Schema class and the namespaces <http://www.w3c.org/2001/XMLSchema> [XML Schema profile] and its target namespace are linked using the xmlns relationships. The names of these links are: the same as target namespace, for the link to target namespace; "xs" for the XML Schema namespace.

Class diagrams are transformed into XML Schema diagrams.

The model elements, which have no meaning in the XML schemas, are discarded. This includes (without limitation) behavioral features of classes, interfaces, actors, use cases, states, activities, objects, messages, stereotype and tag definitions.

There are additional properties to choose for UML to XML Schema transformation in the **Model Transformation Wizard** (for more information about Model Transformation Wizard, see *MagicDraw User Manual*, *Tools* section, *Model Transformations Wizard* subsection.)

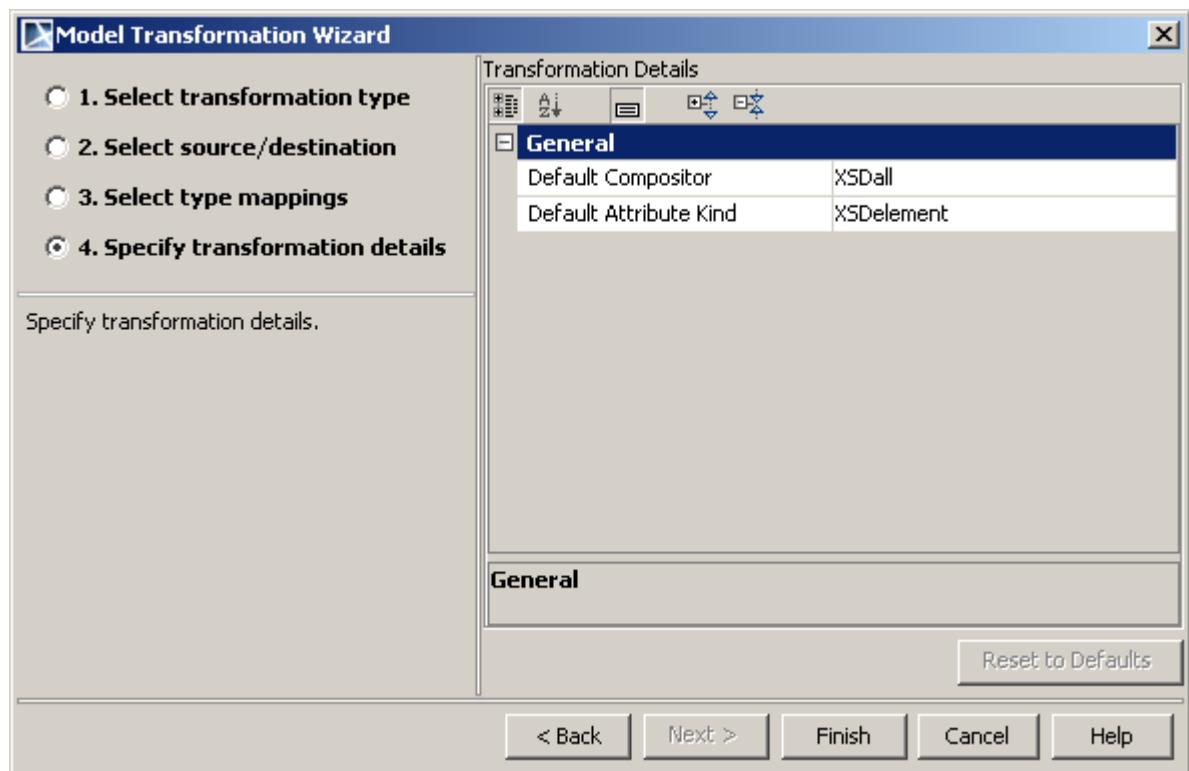


Figure 3 -- Model Transformation Wizard for UML to XML Schema transformation, Specify Transformation Details

Tab name	Box	Function
Default Compositor	Combo box	Possible choices: XSDall, XSDchoice, XSDsequence Determines element grouping in complex types of XML Schema. Default: XSDall
Default Attribute Kind	Combo box	Determines to what attribute kind, XSDelement or XSDattribute UML attribute will be mapped. Default: XSDelement

XML Schema to UML transformation

The XML Schema to UML transformation helps to extract the abstract UML model from the XML schema model.

Type mapping

Type map stores mapping between primitive UML data types and primitive XML Schema data types, the same applied for UML to XML Schema Transformation just in reversed order. For XML Schema to UML element type mapping diagram, see “Type mapping” on page 579.

Transformation Results

The XML Schema diagrams are transformed to the Class diagrams.

Unnecessary stereotypes (XSDxxxx) are discarded from the classes.

Attributes of the classes are gathered if they were spread into several different classes.

Attributes of the classes may be realized as associations. In this case the main class gathers all the associations of the members.

The same principle is applied when elements are in a group, shared by two or more classes. Elements (attributes) are copied into both destination classes.

The attributes with <<XSDgroupRef>> stereotype are treated as if the group relationship has been drawn and transformed accordingly - discarded in the UML model, and the group content (elements/attributes) placed in their place.

Simple XML schema types (classes with <<XSDsimpleType>> stereotype), which after copying and type remap happen to be derived from any data type (UML DataType) or not derived from anything, are transformed into the UML data types.

Simple XML schema types, which are derived by restriction from string and are restricted by enumerating string values, are converted into enumerations in the UML diagrams.

The classes with <<XSDschema>> stereotype are not copied into the destination model.

The <<XDSkey>>, <<XSDkeyref>>, <<XSDunique>> stereotyped attributes are not copied into destination model.

The <<XDSany>>, <<XSDanyAttribute>> stereotyped attributes are not copied into destination model.

The <<XDSnotation>> stereotyped attributes are not copied into destination model.

The <<XDSlength>>, <<XDSminLength>>, <<XDSmaxLength>>, <<XSDpattern>>, <<XSDfractionDigits>>, <<XSDtotalDigits>>, <<XDSmaxExclusive>>, <<XDSmaxInclusive>>, <<XDSminExclusive>>, <<XDSminInclusive>> stereotyped attributes are not copied into destination model.

The XML schemas (classes with XSDschema stereotype) should not be transformed, but they may contain inner classes (anonymous types of schema elements). These inner classes are transformed using usual rules for UML type transformation - as if they were not inner classes but normal XML schema types.