## CONTENTS

### INTRODUCTION
- Overview 7

### Code Engineering Sets
- Code Engineering Sets 8

### Generating Code
- Code Generation for Set 11
- Code Generation for Model Element 12

### Reverse
- Reverse 12
  - Rules of the association or attribute creation on reverse 14

### Global options for Code Engineering
- Global options for Code Engineering 17
  - Code engineering options for all sets in your project 17
  - Java Documentation Properties dialog box 21
  - Round Trip 22
  - Type Mapping Table 22

### Files of Properties
- Files of Properties 23

### JAVA CODE ENGINEERING
- Java Code Engineering 24

#### Introduction
- Java Code Engineering Introduction 24
  - Abbreviations 24
  - References 24
  - Java support in MagicDraw 24

#### Java Mapping to UML
- Java Mapping to UML 25
  - Java Profile 25
  - Java referenced types 26
  - Mapping to UML rules 27

#### Java CE Properties
- Java CE Properties 52
  - Java Reverse Properties 52
  - Java Language Options 53

#### Method Implementation Reverse
- Method Implementation Reverse 56
  - Sequence Diagram from Java Source Wizard 56

### C++ CODE ENGINEERING
- C++ Code Engineering 61

#### Abbreviations
- C++ Code Engineering Abbreviations 61

#### References
- C++ Code Engineering References 61

#### C++ ANSI Profile
- C++ ANSI Profile 61
  - Data Types 61
  - Stereotypes 64

#### Mapping
- Mapping 72
  - Class 72
    - Base Class Definition 72
    - Class Member Variable 73
    - Class Member Function 74
    - Class Constructor/Destructor 74
  - Variable 75
    - Variable modifiers 76
    - Variable extern 77
    - Variable default value 78
    - Const Volatile qualified type 79
  - Function 79
    - Function variable-length parameter list 80
void parameter  80
Register parameter  81
Function modifiers  82
Function pointer  82
Function operator  83
Exception  83
Visibility  84
Static members  84
Friend declaration  85
Struct  86
Union  86
Enumeration  86
Typedef  87
Namespace  88
Global functions and variables  88
Class definition  89
Class Template Definition  89
Function Template Definition  90
Default template parameter  91
Template instantiation  92
Partial template instantiation  95
Template specialization  95
Forward class declaration  96
Include declaration  97
Conversion from old project version  100
Translation Activity Diagram  100
Language properties  104
Type Modifiers  123
Stereotypes  126
Tag Value  135
Constructor and Destructor name  137
Data type  139
DSL customization  145
Operation and Constructor  145
Attribute  146
Generalization  147
Enumeration literal  147
Namespace  147
Template parameter  148
Profile constraints  148
Operation  148
Constructor  148
Destructor  149
Global  149
Typedef  149
Friend  149
New in MagicDraw 12.1  150
CG Properties Editor  150
Roundtrip on #include statement and forward class declaration  150
Project Option and Code Generation Options  154

Copyright © 1998-2015 No Magic, Inc.
## CONTENTS

New in MagicDraw 14.0  157  
  Support C++ dialects  157  
  CG Properties Editor  158  

New in MagicDraw 16.8  159  
  Doxygen-after-member documentation  159  
  @see on documentation  160  
  Symbian macro  160  

Tutorial  160  
  Type Modifier  160  
  Global Member  161  
  Typedef  161  
  Function Pointer  163  
  Friend  165  
  How to specify component to generate code to  167  
  @see support for import-code-only mode  168  
  Navigable short cut from model link in documentation  169  

Project constraint  170  
  Working with QT  174  

Microsoft C++ Profiles  175  
  Microsoft Visual C++ Profile  175  
  C++/CLI Profile  183  
  C++ Managed Profile  192  
  Modeling with Microsoft Specific Profiles  194  

C# CODE ENGINEERING  229  

C# 2.0 Description  229  
  Generics  229  
  Anonymous Methods  250  
  Partial Types  251  
  Nullable Types  254  
  Accessor Declarations  255  
  Static Class  256  
  Extern Alias Directive  257  
 Pragma Directives  259  
  Fix Size Buffer  261  

C# 3.0 Description  264  
  Extension Methods  264  
  Lambda Expression Conversion  265  

C# Profile  267  
  Stereotype  267  
  Data Type  272  

Conversion from old project version  272  
  Translation Activity Diagram  272  

Mapping  277  
  Language Properties Mapping  277  
  C# Properties Customization  281  
  Using Directive Mapping  282  

Constraints  286  
  Mapping Constraints  286  
  UML Constraints  286  

Copyright © 1998-2015 No Magic, Inc.
CONTENTS

Translation Constraints 289

CORBA IDL MAPPING TO UML 290
  CORBA Interface Implementation 291

WSDL 294
  WSDL Mapping to UML elements 295
    Defined stereotypes 295
    Definitions 295
    Import, namespace 296
    Messages 296
    Types 298
    Port types 299
    Bindings 300
    Services 302
    Ports 304
INTRODUCTION

Overview

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

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

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

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

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

MagicDraw code engineering supports Java, C++, CORBA IDL, DDL, XML Schema, WSDL, and C# languages. 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:

<table>
<thead>
<tr>
<th>Language</th>
<th>Professional Edition</th>
<th>Enterprise Edition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Java</td>
<td>Java</td>
<td>+</td>
</tr>
<tr>
<td>Java Bytecode</td>
<td>Java</td>
<td>+</td>
</tr>
<tr>
<td>C++</td>
<td>C++</td>
<td>+</td>
</tr>
<tr>
<td>CORBA IDL</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td>DDL/Database</td>
<td>With Cameo Data Modeler plugin (separately purchaseable)</td>
<td>With Cameo Data Modeler plugin (free of charge)</td>
</tr>
<tr>
<td>engineering</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CIL</td>
<td>C#</td>
<td>+</td>
</tr>
<tr>
<td>CIL Disassembler</td>
<td>C#</td>
<td>+</td>
</tr>
<tr>
<td>XML Schema</td>
<td>With Cameo Data Modeler plugin (separately purchaseable)</td>
<td>With Cameo Data Modeler plugin (free of charge)</td>
</tr>
<tr>
<td>WSDL</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td>C#</td>
<td>C#</td>
<td>+</td>
</tr>
</tbody>
</table>
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), XML Schema, and WSDL). The new set is created.

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.
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.

![Round Trip Set dialog box](image)

**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.

<table>
<thead>
<tr>
<th>Element name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>All files</td>
<td>Helps you find directories with the source files for the set.</td>
</tr>
<tr>
<td>Files of type</td>
<td>Contains possible file name extensions for the chosen language.</td>
</tr>
</tbody>
</table>
The **Add Data from Model** tab helps you manage elements located in the UML model.

![Round Trip Set dialog box. Add data from model tab](image)

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:

<table>
<thead>
<tr>
<th>Button</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Add</strong></td>
<td>The selected file in the <strong>All Files</strong> or <strong>All Data</strong> list is added to the set.</td>
</tr>
<tr>
<td><strong>Add All</strong></td>
<td>All files in the opened or selected directory are added to the set.</td>
</tr>
<tr>
<td><strong>Add Recursively</strong></td>
<td>All files in the opened directory and its subdirectories are added to the set.</td>
</tr>
<tr>
<td><strong>Remove</strong></td>
<td>Removes the selected entity from the set.</td>
</tr>
<tr>
<td><strong>Remove All</strong></td>
<td>Removes all entities from the set.</td>
</tr>
</tbody>
</table>

**Generating Code**

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

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

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

The Code Generation Options dialog box appears.

![Code Generation Options dialog box](image)

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

<table>
<thead>
<tr>
<th>Box name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output Directory</td>
<td>Type the directory where the generated files will be saved.</td>
</tr>
<tr>
<td>‘...’</td>
<td>The Set Output Directory dialog box appears. Select the directory for the generated files.</td>
</tr>
<tr>
<td>Set as Working Directory</td>
<td>The output directory is set as a working directory and files are saved to the working directory.</td>
</tr>
<tr>
<td>Reverse before generation</td>
<td>Changes your model according to changes in the existing code.</td>
</tr>
</tbody>
</table>

**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.
INTRODUCTION

Reverse

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

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

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

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

<table>
<thead>
<tr>
<th>Box name</th>
<th>Function</th>
</tr>
</thead>
</table>
| If element deleted from model     | To influence the structure of generated code, click one of the following option buttons:  
|                                  | • 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                                                  |
INTRODUCTION
Reverse

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

Models can be reversed without creating a set.

To reverse a model without creating a set

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

![Reverse Options dialog box](image)

**Figure 6 -- Reverse options dialog box**

<table>
<thead>
<tr>
<th>Element name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CREATE CLASS FIELDS AS</strong></td>
<td></td>
</tr>
<tr>
<td>Attributes</td>
<td>Class fields are represented in model as attributes.</td>
</tr>
<tr>
<td>Associations</td>
<td>Class fields are represented in model as association ends.</td>
</tr>
<tr>
<td>According to rules</td>
<td>Association or Attribute creation on reverse is ability to enter rules that help to decide if an association or attribute must be created on reverse. For more information, see “Rules of the association or attribute creation on reverse” on page 14.</td>
</tr>
<tr>
<td>Resolve collection generics</td>
<td>Reverse engineering is able to create associations when one class has collection of other classes and uses Java generics (for example, List&lt;String&gt;). If selected, types of collection will be resolved (property type will be not collection, but real type). Predefined container types in Java language properties will be appended by all the same containers in form: java.util.List&lt;$$type$$&gt; where $$type$$ replaced to value of &quot;type&quot; property when code is generated.</td>
</tr>
</tbody>
</table>

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

**Rules of the association or attribute creation on reverse**

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

Creating association or attribute creation rules on reverse

1. In the MagicDraw Browser, select the code engineering set for reverse.
2. From the set shortcut menu, choose the Reverse command. The Reverse Options dialog box is opened.
3. In the Create class fields as group, select the According to rules radio button. Then press the “…” button. The Class Field Creation Rules dialog box opens (see Figure 7 on page 15).
Managing association or attribute creation on reverse rules

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

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

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

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

Defining rules

You can define rules in the **Rules** dialog box (see Figure 8 on page 16). To invoke this dialog box, in the **Class Field Creation** dialog box, press the **Add** or **Edit** button.

The **Rules** dialog box description:

- In the **Create** drop down list select to create **Attribute** or **Association** on reverse if property matches defined critters.
- The **Qualified Name** expression field supports simple search patterns - supports ‘*’ and ‘?’
The Qualified Name expression pattern can contain qualified name of the target element, which contains separator of UML separator style (::) and qualified name is counted to the first package with <<modelLibrary>> stereotype.

The Element Type drop down list contains subset of UML Type names: Any, Class, Interface, DataType, Primitive, Enumeration, Stereotype values are displayed.

If in the Element Type drop down list Any value is selected, the element type will be matched by qualified name expression.

Sample of the association creation on reverse

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

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

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

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

- Create “Association”, if type name “match” path “myClient::*” with “all” meta type.
- Create “Association”, if type name “match” path “myServer::*” with “all” meta type.
- If rules do not match, create class fields as attributes.
Global options for Code Engineering

Code engineering options for all sets in your project

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

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\(^1\), 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 10).

**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 12).

### Tab name | Element name | Function
--- | --- | ---
**Java Language Options** | 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.

---

\(^1\) Starting from version 18.1, referred as “used project” in MagicDraw UI. This user guide mentions the old keyword, which will be replaced in the documentation of the next MagicDraw version.
<table>
<thead>
<tr>
<th>Tab name</th>
<th>Element name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use CLASSPATH</td>
<td>The ‘...’ button is activated. Search a classpath for importing sentences generation in the Set classpath dialog box.</td>
<td></td>
</tr>
</tbody>
</table>

![Figure 10 -- Set classpath dialog](image)

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.

<table>
<thead>
<tr>
<th>Java Source</th>
<th>Available choices 1.4, 5.0, or 7.0.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Header</td>
<td>Add the specific header to all your code files. Click the button and enter header text in the <strong>Header</strong> screen. You may also define $DATE, $AUTHOR, and $TIME in the header.</td>
</tr>
</tbody>
</table>

**C++ Language Options**

Set the generated code style for C++ programming languages.

<table>
<thead>
<tr>
<th>Element name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Generate opening bracket in new line</td>
<td>Opens a bracket in the newly generated line.</td>
</tr>
<tr>
<td>Generate spaces</td>
<td>Spaces inside an assignment and other operators are generated.</td>
</tr>
<tr>
<td>Generate empty documentation</td>
<td>Comment brackets are placed in your code, unless class in the model has no documentation.</td>
</tr>
<tr>
<td>Generate methods body into class</td>
<td>Select check box to generate methods body into class.</td>
</tr>
<tr>
<td>Documentation Style</td>
<td>Two styles are available for documentation.</td>
</tr>
</tbody>
</table>

**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.
<table>
<thead>
<tr>
<th>Tab name</th>
<th>Element name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDL Language Options</td>
<td>Generate opening bracket in new line</td>
<td>Opens a bracket in the new line generating.</td>
</tr>
<tr>
<td></td>
<td>Generate spaces</td>
<td>Spaces inside an assignment and other operators are generated.</td>
</tr>
<tr>
<td></td>
<td>Generate empty documentation</td>
<td>Comment brackets are placed in your code, unless class in the model has no documentation.</td>
</tr>
<tr>
<td></td>
<td>Generate imports</td>
<td>Generation of &quot;import&quot; statements for classes that are referenced in the generated class.</td>
</tr>
<tr>
<td></td>
<td>Generate preprocessor directives</td>
<td>Generates pre-processors directives.</td>
</tr>
<tr>
<td>Documentation Style</td>
<td></td>
<td>Three styles are available for documentation.</td>
</tr>
<tr>
<td></td>
<td>Header &quot;...&quot;</td>
<td>Add the specific header to all your code files. Click the &quot;...&quot; button and enter header text in the Header dialog. You may also define $DATE, $AUTHOR, and $TIME in the header. Since MagicDraw version 17.0, the variable $DOCUMENTATION can be used for exporting the documentation of the file component (from the File View package) as a header of the IDL file. <strong>IMPORTANT!</strong> The variable $DOCUMENTATION is available with MagicDraw 17.0 Service Pack 4 and later versions.</td>
</tr>
<tr>
<td>Set Include Path</td>
<td></td>
<td>Specify the path for the &quot;includes&quot;. Click the &quot;...&quot; button to open the Select Folder dialog box.</td>
</tr>
<tr>
<td>C# Language Options</td>
<td>Generate opening bracket in new line</td>
<td>Opens a bracket in the newly generated line.</td>
</tr>
<tr>
<td></td>
<td>Generate spaces</td>
<td>Generates spaces inside an assignment and other operators.</td>
</tr>
<tr>
<td></td>
<td>Generate empty documentation</td>
<td>Comment brackets are placed in your code, unless class in the model has no documentation.</td>
</tr>
<tr>
<td></td>
<td>Header</td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td>Generate required &quot;using&quot; directives</td>
<td>Automatic generation of &quot;using&quot; directives. This option facilitates the usage of namespaces and types defined in other namespaces.</td>
</tr>
<tr>
<td></td>
<td>Concatenate namespace names</td>
<td>If not selected namespace names are separated into several lines. e.g.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>namespace A</td>
</tr>
<tr>
<td></td>
<td></td>
<td>{</td>
</tr>
<tr>
<td></td>
<td></td>
<td>namespace B</td>
</tr>
<tr>
<td></td>
<td></td>
<td>{</td>
</tr>
</tbody>
</table>
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.

![Documentation Properties dialog box](image)

Figure 11 -- Documentation Properties dialog box

<table>
<thead>
<tr>
<th>Tab name</th>
<th>Element name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Documentation:</td>
<td>Processor</td>
<td>Use C# XMI processor then generates c# xmi documentation for commenting the code.</td>
</tr>
<tr>
<td></td>
<td>Style</td>
<td>Select one of the listed comment styles.</td>
</tr>
<tr>
<td></td>
<td>Header</td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td>Conditional Symbols</td>
<td>Add the conditional symbols, which cannot be recognized and should be skipped during reverse. Click the ‘...’ button and add conditional symbols in the Define Conditional Symbols dialog box.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Box Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tag Name</td>
<td>Type a tag name.</td>
</tr>
<tr>
<td>Value</td>
<td>Type the value of the tag.</td>
</tr>
<tr>
<td>Generate</td>
<td>The selected tag will be placed in the generated code as a comment before classifier (class or interface), operation or attribute.</td>
</tr>
<tr>
<td>Up</td>
<td>Moves the selected item up the list.</td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>Supported language</th>
<th>File of Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>JAVA</td>
<td>java.prop</td>
</tr>
<tr>
<td>C++</td>
<td>C++.prop</td>
</tr>
<tr>
<td>CORBA IDL</td>
<td>idl.prop</td>
</tr>
<tr>
<td>JAVA Bytecode</td>
<td>javabytecode.prop</td>
</tr>
<tr>
<td>DDL</td>
<td>ddl.prop</td>
</tr>
<tr>
<td>CIL</td>
<td>cil.prop</td>
</tr>
<tr>
<td>CIL Disassembler</td>
<td>cil disassembler.prop</td>
</tr>
<tr>
<td>C#</td>
<td>c#.prop</td>
</tr>
<tr>
<td>IDL</td>
<td>idl.prop</td>
</tr>
<tr>
<td>XML Schema</td>
<td>xmlschema.prop</td>
</tr>
<tr>
<td>WSDL</td>
<td>wsdl.prop</td>
</tr>
</tbody>
</table>

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.
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 25 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 52 introduces specific Java options.

Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UML</td>
<td>Unified Modeling Language</td>
</tr>
<tr>
<td>JLS</td>
<td>Java Language Specification</td>
</tr>
<tr>
<td>CE</td>
<td>Code Engineering</td>
</tr>
<tr>
<td>CES</td>
<td>Code Engineering Set</td>
</tr>
<tr>
<td>JVM</td>
<td>Java Virtual Machine</td>
</tr>
</tbody>
</table>

References


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.
Java referenced types

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

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

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

MagicDraw resolves referenced classes from the specified class path (by default class path is boot class path taken from the JVM on which MagicDraw is started) and creates in appropriate package structure (for more
information see "Sorting reversed classes according to the classpath", on page 27).

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

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

Sorting reversed classes according to the classpath

References are sorted to packages according to the classpath entry.

See a sample in Figure 13 on page 27, there reverse result is represented of the following java code:

```java
go public class CustomFrame extends JFrame
{    String mTitle;
}
```

![Figure 13 -- Sample: The String, JFrame and CustomFrame classes location after reverse](image)

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.

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

<table>
<thead>
<tr>
<th>Java element</th>
<th>MagicDraw-UML element</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class declaration</td>
<td>UML Class with stereotype «JavaClass» (optional)</td>
</tr>
<tr>
<td>Class name</td>
<td>UML Class name</td>
</tr>
<tr>
<td>Class documentation</td>
<td>UML Class Documentation</td>
</tr>
<tr>
<td>Class extends clause</td>
<td>Is mapped to the UML Generalization relationship, where supplier is extended class and client is mapped class</td>
</tr>
<tr>
<td>Class implements clause</td>
<td>Is mapped to the UML Interface Realization relationship, where supplier is extended class and client is mapped class.</td>
</tr>
<tr>
<td>Visibility modifier</td>
<td>UML Class “Visibility” property</td>
</tr>
<tr>
<td>Abstract modifier</td>
<td>UML Class “Is Abstract” property</td>
</tr>
<tr>
<td>Java element</td>
<td>MagicDraw-UML element</td>
</tr>
<tr>
<td>--------------------</td>
<td>------------------------------------------------------------</td>
</tr>
<tr>
<td>Final modifier</td>
<td>UML Class “Is Final Specialization” property</td>
</tr>
<tr>
<td>Static modifier</td>
<td>Java Language property “Static modifier”</td>
</tr>
<tr>
<td>Strictfp modifier</td>
<td>Java Language property “Strictfp modifier”</td>
</tr>
</tbody>
</table>

**Example**

**Java Source Code**

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

**MagicDraw UML Model**

![Class diagram]

*Figure 15 -- Class diagram*
Figure 16 -- UML Class specification dialog
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

<table>
<thead>
<tr>
<th>Java element</th>
<th>MagicDraw-UML element</th>
</tr>
</thead>
<tbody>
<tr>
<td>Field declaration</td>
<td>UML Property, owned by UML Class, with stereotype «JavaProperty» (optional)</td>
</tr>
<tr>
<td>Field name</td>
<td>UML Property Name</td>
</tr>
<tr>
<td>Field documentation</td>
<td>UML Field Documentation</td>
</tr>
<tr>
<td>Field type</td>
<td>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</td>
</tr>
<tr>
<td>Field type modifiers</td>
<td>Is mapped to the MagicDraw specified property “Type Modifier”</td>
</tr>
<tr>
<td>Visibility modifier</td>
<td>UML Property “Visibility” property</td>
</tr>
<tr>
<td>Final modifier</td>
<td>UML Property “Is Read Only” property</td>
</tr>
<tr>
<td>Modifier</td>
<td>Description</td>
</tr>
<tr>
<td>----------------</td>
<td>------------------------------------</td>
</tr>
<tr>
<td>Static modifier</td>
<td>UML Property &quot;Is Static&quot; property</td>
</tr>
<tr>
<td>Transient modifier</td>
<td>Java Language property &quot;Transient modifier&quot;</td>
</tr>
<tr>
<td>Volatile modifier</td>
<td>Java Language property &quot;Volatile modifier&quot;</td>
</tr>
</tbody>
</table>

**Example**

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

**MagicDraw UML Model**

*Figure 18 -- Class with property*
Figure 19 -- UML Property specification dialog
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

<table>
<thead>
<tr>
<th>Java element</th>
<th>MagicDraw-UML element</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operation declaration</td>
<td>UML Operation, owned by UML Class, with stereotype «JavaOperation» (optional)</td>
</tr>
<tr>
<td>Operation name</td>
<td>UML Operation Name</td>
</tr>
<tr>
<td>Operation documentation</td>
<td>UML Operation Documentation</td>
</tr>
<tr>
<td>------------------------</td>
<td>----------------------------</td>
</tr>
<tr>
<td>Parameters list</td>
<td>UML Operation “Parameters” list</td>
</tr>
<tr>
<td>Return type</td>
<td>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</td>
</tr>
<tr>
<td>Return type modifiers</td>
<td>Is mapped to the MagicDraw specified property “Type Modifier”</td>
</tr>
<tr>
<td>Visibility modifier</td>
<td>UML Operation “Visibility” property</td>
</tr>
<tr>
<td>Final modifier</td>
<td>UML Operation “Is Leaf” property</td>
</tr>
<tr>
<td>Abstract modifier</td>
<td>UML Operation “Is Abstract” property</td>
</tr>
<tr>
<td>Static modifier</td>
<td>UML Operation “Is Static” property</td>
</tr>
<tr>
<td>Synchronized modifier</td>
<td>UML Operation “Concurrency” kind “guarded”</td>
</tr>
<tr>
<td>Throws list</td>
<td>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</td>
</tr>
<tr>
<td>Native modifier</td>
<td>Java Language property “Native modifier”</td>
</tr>
<tr>
<td>Strictfp modifier</td>
<td>Java Language property “Strictfp modifier”</td>
</tr>
</tbody>
</table>

**Parameter mapping table**

<table>
<thead>
<tr>
<th>Java element</th>
<th>MagicDraw-UML element</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameter declaration</td>
<td>UML Parameter, owned by UML Operation, with stereotype «JavaParameter» (optional)</td>
</tr>
<tr>
<td>Parameter name</td>
<td>UML Parameter Name</td>
</tr>
<tr>
<td>Parameter documentation</td>
<td>When is used JavaDoc preprocessing, it is mapped to UML Parameter Documentation, else it is part of UML Operation Documentation.</td>
</tr>
<tr>
<td>Return type</td>
<td>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.</td>
</tr>
<tr>
<td>Return type modifiers</td>
<td>Is mapped to the MagicDraw specified property “Type Modifier”</td>
</tr>
<tr>
<td>Final modifier</td>
<td>Direction kind “in” of UML Parameter</td>
</tr>
</tbody>
</table>
Example

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

MagicDraw UML Model

Figure 21 -- Class with operation

Figure 22 -- UML Operation specification dialog
Figure 23 -- UML Operation language properties
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 28.

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 28.
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

<table>
<thead>
<tr>
<th>Java element</th>
<th>MagicDraw-UML element</th>
</tr>
</thead>
<tbody>
<tr>
<td>Literal declaration</td>
<td>UML Parameter with stereotype «JavaEnumerationLiteral», owned by UML Class with stereotype «JavaEnumerationLiteral»</td>
</tr>
<tr>
<td>Literal name</td>
<td>UML Property Name</td>
</tr>
<tr>
<td>Literal documentation</td>
<td>When is used JavaDoc preprocessing, it is mapped to UML Parameter Documentation, else it is part of UML Operation Documentation.</td>
</tr>
</tbody>
</table>

Example

Java Source Code

```java
enum MyInterface {
}
```

MagicDraw UML Model

Figure 25 -- UML Enumeration

Figure 26 -- 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**

<table>
<thead>
<tr>
<th>Java element</th>
<th>MagicDraw-UML element</th>
</tr>
</thead>
<tbody>
<tr>
<td>Annotation declaration</td>
<td>UML Interface with stereotype «JavaAnnotation» (optional)</td>
</tr>
<tr>
<td>Annotation name</td>
<td>UML Interface name</td>
</tr>
<tr>
<td>Annotation documentation</td>
<td>UML Interface Documentation</td>
</tr>
<tr>
<td>Visibility modifier</td>
<td>UML Interface “Visibility” property</td>
</tr>
</tbody>
</table>

**Example**

**Java Source Code**

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

**MagicDraw UML Model**

![UML Interface representing Java annotation type](image)

**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**

<table>
<thead>
<tr>
<th>Java element</th>
<th>MagicDraw-UML element</th>
</tr>
</thead>
<tbody>
<tr>
<td>Annotation member declaration</td>
<td>UML Operation with stereotype «JavaAnnotationMember» (Optional), owned by UML Interface with Stereotype «JavaAnnotation».</td>
</tr>
<tr>
<td>Annotation member name</td>
<td>UML Operation Name</td>
</tr>
</tbody>
</table>
**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**

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

**MagicDraw UML Model**

![Figure 28 -- UML Interface representing Java annotation type with Java annotation members](image-url)

*Figure 28 -- UML Interface representing Java annotation type with Java annotation members*
Figure 29 -- UML Operation, representing Java annotation member, specification dialog
Annotations Usage

Java element can be annotated. Such annotation is mapped to 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**

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

**MagicDraw UML Model**

*Figure 31 -- Java annotation usage*

*Figure 32 -- 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

<table>
<thead>
<tr>
<th>Java element</th>
<th>MagicDraw-UML element</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type variable declaration</td>
<td>Is mapped to the UML Template Parameter. This UML Template Parameter is of the Class type from the UML Metamodel. UML Template Parameter has property “Parametered Element” of the UML Template Parameter. Metamodel type “Class” is taken from the “UML Standard profile/UML 2 Metamodel”</td>
</tr>
<tr>
<td>Type variable name</td>
<td>Is mapped to the UML Class name. This UML Class is “Parametered Element”</td>
</tr>
<tr>
<td>Type bounds</td>
<td>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.</td>
</tr>
</tbody>
</table>

Example

Java Source Code

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

MagicDraw UML Model

![Figure 33 -- UML Class with template parameter E, representing Java type variable](image-url)
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 35 -- Template parameters in UML Class specification dialog
**Parameterized Type**

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

**Parameterized type mapping table**

<table>
<thead>
<tr>
<th>Java element</th>
<th>MagicDraw-UML element</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameterized type</td>
<td>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.</td>
</tr>
<tr>
<td>Parameter for type</td>
<td>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</td>
</tr>
<tr>
<td>Type modifiers of the parameter</td>
<td>It is mapped to the MagicDraw property “Type modifiers” of the UML Template Parameter Substitution</td>
</tr>
<tr>
<td>Wildcard</td>
<td>UML Class with a name “?” from the Java Profile is used as “Actual” value in UML Template Parameter Substitution.</td>
</tr>
<tr>
<td>Wildcard with bounds</td>
<td>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.</td>
</tr>
</tbody>
</table>
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 36 -- UML Interface representing “java.util.List” with template parameter](image)

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 37 -- UML Template Binding between template class and bounded element](image)

You need to open UML Template Binding specification dialog and create UML Template Parameter Substitution for appropriate UML Template Parameter.
Figure 38 -- 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>

![Image of parameterized type creation]

Figure 39 -- Created parameterized type for “java.util.List<String>”

**Imports**

<table>
<thead>
<tr>
<th>Java element</th>
<th>MagicDraw-UML element</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type import</td>
<td>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.</td>
</tr>
<tr>
<td>Type</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------------</td>
<td>------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Package import</td>
<td>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.</td>
</tr>
<tr>
<td>Static import for all static members</td>
<td>Mapped to the UML Element Import with stereotype «JavaStaticImport». <code>{JavaImportAll}</code> 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.</td>
</tr>
<tr>
<td>Static import for single static members</td>
<td>Mapped to the UML Element Import with stereotype «JavaStaticImport». <code>{JavaImportAll}</code> tag of «JavaStaticImport» must have “false” value and tag <code>{JavaImportedMember}</code> 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, <code>{JavaImportedMember}</code> can have listed all of them.</td>
</tr>
</tbody>
</table>
Example

Java Source Code

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

public class Test {
    
}
```

MagicDraw UML Model

Figure 40 -- 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 41 on page 52. Other options are common for all languages and are described in “Reverse”, on page 12.

![Reverse Options - Untitled1](image)

Figure 41 -- 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

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

MagicDraw UML Model

![Test](image)

Figure 42 -- Attribute type, retrieved from collection
Figure 43 -- 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.
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.

```java
/**
 *
 */
```
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 49).

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 49.

Class Count To Generate Import On Demand

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

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

Documentation

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

Style 1
    /**
    *
    */

Style 2
    /**
    *
    */

Processor is responsible for analyzing documentation context. There are two types <none> and Java Doc. <none> options does nothing with documentation and just set it as is on element.

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

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

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

Classpath

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

This options is used by Java reverse engineering. If referenced element is not found in model for some reasons, that it is searched in this defined path. And if class is matched by name, this class is added into model.
By default, MagicDraw imports boot classpath of the JVM, on which is running.

**Java Source**

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

**Header**

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

Template strings

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

**Method Implementation Reverse**

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

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

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

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

**Sequence Diagram from Java Source Wizard**

Sequence Diagram from Java Source Wizard is the primary tool for reversing s sequence diagram from Java method. It contains four steps that are described below.
STEP 1 Specify Name and Package.

In this step, type the name of the newly created sequence diagram. By 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 the 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.

*Figure 45 -- Sequence Diagram from Java Source Wizard*
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

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

Figure 46 -- Sequence Diagram from Sequence Wizard. Specify Symbols Properties
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


C++ ANSI Profile

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

Data Types

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

Each fundamental type with a modifier (signed, unsigned, short, long) can be declared in different order, but it is the same type. For example, short int or int short. Only versions defined as “type” by ANSI simple-type-specifiers are created as datatype in the profile. (See ANSI spec 7.1.5.2) If an UML synonym 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 only Integer, Boolean, String, and UnlimitedNatural are defined and they are not defined as datatypes, but as primitives. In the current version of MD UnlimitedNatural is not defined in any profile and Integer and boolean (b is lowercase) are datatypes.
Figure 47 -- C++ datatype

**char**

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

**signed char**

Signed char does not have a synonym.

**unsigned char**

Unsigned char does not have a synonym.

**int**

Synonyms for int are

- signed
- signed int
- int signed

**unsigned int**

Synonyms for unsigned int are

- unsigned
- int unsigned
unsigned short int

Synonyms for unsigned short int are
- unsigned short
- short unsigned
- unsigned int short
- short unsigned int
- short int unsigned
- int short unsigned
- int unsigned short

unsigned long int

Synonyms for unsigned long int are
- unsigned long
- long unsigned
- unsigned int long
- long unsigned int
- long int unsigned
- int long unsigned
- int unsigned long

long int

Synonyms for long int are
- long
- int long
- signed long
- long signed
- signed long int
- signed int long
- long signed int
- long int signed
- int signed long
- int long signed

short int

Synonyms for short int are
- short
- int short
- signed short
- short signed
- signed short int
• signed int short
• short signed int
• short int signed
• int signed short
• int short signed

double
double does not have a synonym.

long double
long double does not have a synonym

float
float does not have a synonym.

void
void does not have a synonym.

bool
bool does not have a synonym.

wchar_t
wchar_t does not have a synonym.

Stereotypes
All C++ stereotypes are based on «C++Element» stereotypes.

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

«C++Class», «C++Operation», «C++Parameter», «C++Attribute», «C++LiteralValue», «C++Include», «C++Generalization», and «C++TemplateParameter» are invisible stereotypes and are used only to store C++ language properties. These stereotypes and their tag definitions are used by the DSL framework.
Figure 48 -- C++ Stereotype tag definitions
C++Operation

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Meta class</th>
<th>Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>C++Operation</td>
<td>Operation</td>
<td>Const function</td>
</tr>
<tr>
<td></td>
<td></td>
<td>void f() const;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Constraint: Only valid for member function</td>
</tr>
<tr>
<td></td>
<td></td>
<td>if isQuery then</td>
</tr>
<tr>
<td></td>
<td></td>
<td>stereotype-</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&gt;select(name='C++Global') -&gt;isEmpty()</td>
</tr>
</tbody>
</table>

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

C++Operator

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Meta class</th>
<th>Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>C++Operator</td>
<td>Operation</td>
<td>operator function</td>
</tr>
<tr>
<td></td>
<td></td>
<td>T&amp; operator+(T&amp; a);</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Constraint: name start with “operator”</td>
</tr>
</tbody>
</table>
### C++Parameter

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

<table>
<thead>
<tr>
<th>name</th>
<th>Meta class</th>
<th>Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>C++Parameter</td>
<td>Parameter</td>
<td></td>
</tr>
<tr>
<td>Tag</td>
<td>Type</td>
<td>Description</td>
</tr>
</tbody>
</table>
| 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.

<table>
<thead>
<tr>
<th>name</th>
<th>Meta class</th>
<th>Constraints</th>
</tr>
</thead>
</table>
| 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.
int x(5);
Constraint
owner.stereotype->exists(name='C++Global') |
| bit field       | String[0..1] | Bit field declaration
int x:2;
Constraint: Only valid for member function
stereotype->select(name='C++Global') ->isEmpty() |
| c type declaration | boolean[1]=false | Declare attribute’s type in C style
C style:
enum Day x;
C++ style:
Day x; |
| container       | String[0..1] | container of the attribute. $ character is replaced by the attribute type.
vector<$$> x; |
C++ LiteralValue

«C++LiteralValue» is an invisible stereotype used to include language properties for any C++ enum field. See Enumeration for more info.

<table>
<thead>
<tr>
<th>name</th>
<th>Meta class</th>
<th>Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>C++LiteralValue</td>
<td>EnumerationLiteral</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Tag</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>String[0..1]</td>
<td>Value definition of an enum field. (A valid C++ expression)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>enum Day {Mon = 2};</td>
</tr>
</tbody>
</table>

C++ Friend

«C++Friend» stereotype is used to define C++ friend relationship. See Friend declaration for more info.

<table>
<thead>
<tr>
<th>Name</th>
<th>Meta class</th>
<th>Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>C++Friend</td>
<td>Dependency</td>
<td>Client is Class or Operation and supplier is Class</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(client.oclIsTypeOf(Class) or client.oclIsTypeOf(Operation)) and</td>
</tr>
<tr>
<td></td>
<td></td>
<td>supplier.oclIsTypeOf(Class)</td>
</tr>
</tbody>
</table>

C++ Struct

«C++Struct» stereotype is used to define C++ struct. See Struct for more info.

<table>
<thead>
<tr>
<th>name</th>
<th>Meta class</th>
<th>Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>C++Struct</td>
<td>Class</td>
<td></td>
</tr>
</tbody>
</table>

C++ Typedef

«C++Typedef» stereotype is used to define C++ typedef. See Typedef for more info.

<table>
<thead>
<tr>
<th>name</th>
<th>Meta class</th>
<th>Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>C++Typedef</td>
<td>Class</td>
<td>A typedef does not contain operation and attribute</td>
</tr>
<tr>
<td></td>
<td></td>
<td>feature-&gt;isEmpty()</td>
</tr>
<tr>
<td></td>
<td></td>
<td>A «C++BaseType» dependency is defined</td>
</tr>
</tbody>
</table>
C++ Union

«C++ Union» stereotype is used to define C++ union. See Union for more info.

<table>
<thead>
<tr>
<th>name</th>
<th>Meta class</th>
<th>Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>C++Union</td>
<td>Class</td>
<td></td>
</tr>
</tbody>
</table>

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 88 for more info.

<table>
<thead>
<tr>
<th>name</th>
<th>Meta class</th>
<th>Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>C++Global</td>
<td>Class</td>
<td></td>
</tr>
</tbody>
</table>

C++ Namespace

«C++ Namespace» stereotype is used to define C++ namespace. See Namespace for more info.

<table>
<thead>
<tr>
<th>name</th>
<th>Meta class</th>
<th>Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>C++Namespace</td>
<td>Package</td>
<td></td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Name</th>
<th>Meta class</th>
<th>Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>C++Constructor</td>
<td>Operation</td>
<td>name = owner.name</td>
</tr>
</tbody>
</table>

Tag Type Description

- explicit
  - boolean[1]=false
  - Explicit constructor
    - explicit a();

- initialization list
  - String[0..1]
  - Constructor initialization
    - a() : x(1) {}
### C++Extern

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Meta class</th>
<th>Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>C++Extern</td>
<td>Operation, Property</td>
<td>owner.stereotype-&gt;exists(name='C++Global')</td>
</tr>
</tbody>
</table>

#### Tag Description

- **linkage**
  - **Type**: String[0..1]
  - **Description**: Linkage specification:
    - `extern "C"`

### C++FunctionPtr

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Meta class</th>
<th>Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>C++FunctionPtr</td>
<td>Parameter, Property</td>
<td></td>
</tr>
</tbody>
</table>

#### Tag Description

- **signature**
  - **Type**: Operation
  - **Description**: The signature of the function (C++ function pointer definition without the operation name)

- **member class**
  - **Type**: Class
  - **Description**: 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.

<table>
<thead>
<tr>
<th>Name</th>
<th>Meta class</th>
<th>Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>C++FunctionSignature</td>
<td>Class</td>
<td>The class cannot have properties. properties-&gt;isEmpty()</td>
</tr>
</tbody>
</table>

### C++Class

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Meta class</th>
<th>Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>C++Class</td>
<td>Class</td>
<td></td>
</tr>
</tbody>
</table>

### C++BaseType

«C++BaseType» stereotype is used to link base type of a typedef.

<table>
<thead>
<tr>
<th>Name</th>
<th>Meta class</th>
<th>Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>C++BaseType</td>
<td>Dependency</td>
<td>Client is type of Class with «C++Typedef» stereotype.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Tag</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
C++ CODE ENGINEERING
C++ ANSI Profile

<table>
<thead>
<tr>
<th>type modifiers</th>
<th>String[0..1]</th>
<th>Type modifiers of the typedef or function pointer.</th>
</tr>
</thead>
<tbody>
<tr>
<td>member class</td>
<td>Class[0..1]</td>
<td>Memberclass of typedef or function pointer.</td>
</tr>
<tr>
<td>array</td>
<td>String[0..1]</td>
<td>Array definition.</td>
</tr>
</tbody>
</table>

**C++Include**

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Meta class</th>
<th>Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>C++Include</td>
<td>Association, Dependency, Generalization, Parameter, Property, TemplateBinding, TemplateParameter</td>
<td>Client is type of Component</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Tag</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>header include</td>
<td>String</td>
<td>The value of tag is one of the following</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• None</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• User Include</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• System Include</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Class Forward</td>
</tr>
<tr>
<td>implementation include</td>
<td>String</td>
<td>The value of tag is one of the following</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• None</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• User Include</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• System Include</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Class Forward</td>
</tr>
</tbody>
</table>

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

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

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

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

**C++TemplateParameter**

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Meta class</th>
<th>Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>C++TemplateParameter</td>
<td>TemplateParameter</td>
<td></td>
</tr>
<tr>
<td>Tag</td>
<td>Type</td>
<td>Description</td>
</tr>
<tr>
<td>type keyword</td>
<td>C++TemplateTypeKeyword=class</td>
<td>The value of tag is one of the following</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• class</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• typename</td>
</tr>
</tbody>
</table>
C++ Generalization

«C++ Generalization» is used for information related to generalization and interface realization.

<table>
<thead>
<tr>
<th>Name</th>
<th>Meta class</th>
<th>Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>C++ Generalization</td>
<td>Generalization, InterfaceRealization</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Tag</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>inheritance visibility</td>
<td>C++ Generalization Visibility[1]=none</td>
<td>The value of tag is one of the following:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• None</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• public</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• protected</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• private</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Example: class A : private B {};</td>
</tr>
<tr>
<td>virtual inheritance</td>
<td>boolean[1]=false</td>
<td>Virtual inheritance</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Example: class A : virtual B {};</td>
</tr>
</tbody>
</table>

Mapping

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

Class

C++ class map to a UML class

<table>
<thead>
<tr>
<th>Code</th>
<th>MD-UML</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>class A {};</td>
<td></td>
<td><img src="A.png" alt="A" /></td>
</tr>
</tbody>
</table>

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.
**Class Member Variable**

Class member variables are mapped to UML attributes. See **Variable** for more info.

<table>
<thead>
<tr>
<th>Code</th>
<th>MD-UML</th>
</tr>
</thead>
<tbody>
<tr>
<td>class baseClass {}; class OtherBaseClass {}; class SuperClass :</td>
<td><strong>Class</strong> - <strong>SuperClass</strong></td>
</tr>
<tr>
<td>public BaseClass, protected virtual OtherBaseClass {};</td>
<td>![Diagram of SuperClass Generalization relationship]</td>
</tr>
<tr>
<td></td>
<td><strong>Generalization</strong> - <strong>&lt;&gt;</strong></td>
</tr>
<tr>
<td></td>
<td>![Diagram of Generalization language properties]</td>
</tr>
</tbody>
</table>
Class Member Function

Class member functions are mapped to UML operations. See Function for more info.

<table>
<thead>
<tr>
<th>Code</th>
<th>MD-UML</th>
</tr>
</thead>
<tbody>
<tr>
<td>class ClassFunction {</td>
<td></td>
</tr>
<tr>
<td>public:</td>
<td></td>
</tr>
<tr>
<td>void simpleFunc();</td>
<td></td>
</tr>
<tr>
<td>float paramFunc(int x, char y);</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Code</th>
<th>MD-UML</th>
</tr>
</thead>
<tbody>
<tr>
<td>class ConstructClass {</td>
<td></td>
</tr>
<tr>
<td>public:</td>
<td></td>
</tr>
<tr>
<td>ConstructClass();</td>
<td></td>
</tr>
<tr>
<td>~ConstructClass();</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
</tbody>
</table>
Variable

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

<table>
<thead>
<tr>
<th>Code</th>
<th>MD-UML</th>
</tr>
</thead>
<tbody>
<tr>
<td>int age;</td>
<td></td>
</tr>
</tbody>
</table>

Figure 51 -- Variable property age

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

<table>
<thead>
<tr>
<th>Code</th>
<th>MD-UML</th>
</tr>
</thead>
<tbody>
<tr>
<td>int* ptrVar;</td>
<td></td>
</tr>
<tr>
<td>int&amp; refVar</td>
<td></td>
</tr>
</tbody>
</table>

Figure 52 -- Variable property ptrVar
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..*]”

<table>
<thead>
<tr>
<th>Code</th>
<th>MD-UML</th>
</tr>
</thead>
<tbody>
<tr>
<td>int arrayVar[5];</td>
<td></td>
</tr>
</tbody>
</table>

### Variable modifiers

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

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

<table>
<thead>
<tr>
<th>Code</th>
<th>MD-UML</th>
</tr>
</thead>
<tbody>
<tr>
<td>mutable int x;</td>
<td></td>
</tr>
</tbody>
</table>
Bit field is mapped to *Bit field* tag value.

<table>
<thead>
<tr>
<th>Code</th>
<th>MD-UML</th>
</tr>
</thead>
</table>
| struct BitStruct {  
  int a:2;  
}; | <<C++Struct>>  
  BitStruct  
  +a : int |

**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.

<table>
<thead>
<tr>
<th>Code</th>
<th>MD-UML</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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 cannot be initialized using function style method.

<table>
<thead>
<tr>
<th>Code</th>
<th>MD-UML</th>
</tr>
</thead>
<tbody>
<tr>
<td>extern int externVar;</td>
<td></td>
</tr>
<tr>
<td>int var = 5;</td>
<td></td>
</tr>
<tr>
<td>int var2(10);</td>
<td></td>
</tr>
</tbody>
</table>
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)

<table>
<thead>
<tr>
<th>Code</th>
<th>MD-UML</th>
</tr>
</thead>
</table>
| class CVModifiers { const int* const constAttribute; } | CVModifiers
|                                           | -constAttribute : int{(readOnly)}           |

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.

<table>
<thead>
<tr>
<th>Code</th>
<th>MD-UML</th>
</tr>
</thead>
<tbody>
<tr>
<td>float paramFunc(int x, char x);</td>
<td></td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>Code</th>
<th>MD-UML</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class FunctionClass {</td>
<td></td>
</tr>
<tr>
<td>public:</td>
<td></td>
</tr>
<tr>
<td>EllipsisFunc(...);</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
</tbody>
</table>

void parameter

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

<table>
<thead>
<tr>
<th>Code</th>
<th>MD-UML</th>
</tr>
</thead>
</table>
**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).

```cpp
class RegisterParamClass {
    void registerParam(register int x);
};
```
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.

<table>
<thead>
<tr>
<th>Code</th>
<th>MD-UML</th>
</tr>
</thead>
</table>
| ```cpp
class FuncModifierClass {
    explicit FuncModifierClass();
};
``` | ![Operation - FuncModifierClass](image) |

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.

<table>
<thead>
<tr>
<th>Code</th>
<th>MD-UML</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cpp</code></td>
<td><img src="image" alt="Code MD-UML" /></td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>Code</th>
<th>MD-UML</th>
</tr>
</thead>
<tbody>
<tr>
<td>float (A*funcPtr)(int);</td>
<td></td>
</tr>
<tr>
<td>Class Op {</td>
<td></td>
</tr>
<tr>
<td>Public:</td>
<td></td>
</tr>
<tr>
<td>Op operator+(Op x);</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Code</th>
<th>MD-UML</th>
</tr>
</thead>
<tbody>
<tr>
<td>void throwFunc() throw (int,char);</td>
<td></td>
</tr>
</tbody>
</table>
**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.

<table>
<thead>
<tr>
<th>Code</th>
<th>MD-UML</th>
</tr>
</thead>
<tbody>
<tr>
<td>class ClassVisibility {</td>
<td></td>
</tr>
<tr>
<td>int privateVar;</td>
<td></td>
</tr>
<tr>
<td>protected:</td>
<td></td>
</tr>
<tr>
<td>int protectedVar;</td>
<td></td>
</tr>
<tr>
<td>public:</td>
<td></td>
</tr>
<tr>
<td>int publicVar;</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td>ClassVisibility</td>
</tr>
<tr>
<td></td>
<td>-privateVar: int</td>
</tr>
<tr>
<td></td>
<td>#protectedVar: int</td>
</tr>
<tr>
<td></td>
<td>+publicVar: int</td>
</tr>
</tbody>
</table>

**Static members**

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

<table>
<thead>
<tr>
<th>Code</th>
<th>MD-UML</th>
</tr>
</thead>
<tbody>
<tr>
<td>class StaticClass {</td>
<td></td>
</tr>
<tr>
<td>static int staticVar;</td>
<td></td>
</tr>
<tr>
<td>static void staticFunc();</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td>StaticClass</td>
</tr>
<tr>
<td></td>
<td>-staticVar: int</td>
</tr>
<tr>
<td></td>
<td>-staticFunc()</td>
</tr>
</tbody>
</table>

**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 cannot be abstract.

<table>
<thead>
<tr>
<th>Code</th>
<th>MD-UMl</th>
</tr>
</thead>
<tbody>
<tr>
<td>class AbstractClass {</td>
<td></td>
</tr>
<tr>
<td>virtual abstractOperation()=0;</td>
<td>![AbstractClass]</td>
</tr>
<tr>
<td>}</td>
<td>![abstractOperation]</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Code</th>
<th>MD-UMl</th>
</tr>
</thead>
<tbody>
<tr>
<td>class ClassB {</td>
<td></td>
</tr>
<tr>
<td>public:</td>
<td>![ClassB]</td>
</tr>
<tr>
<td>friend void friendFunc();</td>
<td>![friendFunc]</td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
<tr>
<td>void friendFunc();</td>
<td></td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Code</th>
<th>MD-UMl</th>
</tr>
</thead>
<tbody>
<tr>
<td>class ClassD {</td>
<td>![ClassD]</td>
</tr>
<tr>
<td>void func(ClassC c);</td>
<td>![ClassD::func]</td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
<tr>
<td>class ClassC {</td>
<td>![ClassC]</td>
</tr>
<tr>
<td>friend void ClassD::func(ClassC c);</td>
<td>![friend::ClassD::func]</td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>Code</th>
<th>MD-UML</th>
</tr>
</thead>
<tbody>
<tr>
<td>class FriendClass { public: friend class ClassA; }; class ClassA { };</td>
<td><img src="image" alt="UML Diagram" /></td>
</tr>
</tbody>
</table>

### 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”

<table>
<thead>
<tr>
<th>Code</th>
<th>MD-UML</th>
</tr>
</thead>
<tbody>
<tr>
<td>struct MyStruc { };</td>
<td><img src="image" alt="UML Diagram" /></td>
</tr>
</tbody>
</table>

### 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”

<table>
<thead>
<tr>
<th>Code</th>
<th>MD-UML</th>
</tr>
</thead>
<tbody>
<tr>
<td>union MyUnion { };</td>
<td><img src="image" alt="UML Diagram" /></td>
</tr>
</tbody>
</table>

### 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.

<table>
<thead>
<tr>
<th>Code</th>
<th>MD-UML</th>
</tr>
</thead>
<tbody>
<tr>
<td>enum Day { Mon, Tue=2 };</td>
<td><img src="image" alt="UML Diagram" /></td>
</tr>
</tbody>
</table>
**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).

<table>
<thead>
<tr>
<th>Code</th>
<th>MD-UML</th>
</tr>
</thead>
<tbody>
<tr>
<td>namespace n { namespace m { } }</td>
<td>![Diagram]</td>
</tr>
</tbody>
</table>

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.

1.
### 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.

<table>
<thead>
<tr>
<th>Code</th>
<th>MD-UML</th>
</tr>
</thead>
<tbody>
<tr>
<td>int var;</td>
<td><img src="image1" alt="UML Diagram" /></td>
</tr>
<tr>
<td>int func(int x);</td>
<td><img src="image2" alt="UML Diagram" /></td>
</tr>
<tr>
<td>namespace std {</td>
<td><img src="image3" alt="UML Diagram" /></td>
</tr>
<tr>
<td>int err;</td>
<td><img src="image4" alt="UML Diagram" /></td>
</tr>
<tr>
<td>void printf();</td>
<td><img src="image5" alt="UML Diagram" /></td>
</tr>
<tr>
<td>}</td>
<td><img src="image6" alt="UML Diagram" /></td>
</tr>
</tbody>
</table>

### 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*. 

<table>
<thead>
<tr>
<th>Code</th>
<th>MD-UML</th>
</tr>
</thead>
<tbody>
<tr>
<td>class VarInitClass {</td>
<td><img src="image7" alt="UML Diagram" /></td>
</tr>
<tr>
<td>} c, d;</td>
<td><img src="image8" alt="UML Diagram" /></td>
</tr>
<tr>
<td>class OuterVarInit {</td>
<td><img src="image9" alt="UML Diagram" /></td>
</tr>
<tr>
<td>class InnerVarInit {</td>
<td><img src="image10" alt="UML Diagram" /></td>
</tr>
<tr>
<td>} e;</td>
<td><img src="image11" alt="UML Diagram" /></td>
</tr>
<tr>
<td>}</td>
<td><img src="image12" alt="UML Diagram" /></td>
</tr>
</tbody>
</table>
### 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
```cpp
template <class T>
void simpleFunc(T x);
// overload old style
void simpleFunc(int x);
// overload new style
template<>
void simpleFunc<char>(char x);
```

### MD-UML
```
<<C++Global>>
+<T : Class | simpleFunc(x : T) +simpleFunc(x : int) +simpleFunc(x : char)>
```

### 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
```cpp
template <class T=int>
class defaultTemplate {
}
```

### MD-UML
```
<T : Class = int
defaultTemplate
defaultTemplateInstance
```
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.

<table>
<thead>
<tr>
<th>Code</th>
<th>MD-UML</th>
</tr>
</thead>
<tbody>
<tr>
<td>template &lt;class T&gt;</td>
<td></td>
</tr>
<tr>
<td>class simpleTemplate {</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
<tr>
<td>simpleTemplate&lt;int&gt;</td>
<td></td>
</tr>
<tr>
<td>simpleTemplateInstance;</td>
<td></td>
</tr>
</tbody>
</table>
For template argument using template instantiation as argument, an intermediate class is created with the specific binding.

```
Code

template <class T>
class T1Class {
};
template <class T>
class T2Class {
};
T1Class<T2Class<int>> ...
```

```
MD-UML

T1Class
\|-- T2Class
\|  \|-- T1Class<T2Class<T->int>>
    \|-- T2Class<T->int>
```

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

template <class T>
class b {
    template <class T>
class c {
    }
};
b<int>::c<char> ...
```

```
MD-UML

b
\|-- T:int
\|-- c
\|  \|-- b<T->int>
    \|-- c<T->char>
```
Example of complex template instantiation. Containment relationship 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.

```c++
// Example code

#include <iostream>

using namespace std;

// Class A
class A {
public:
    template <class T1>
    class A {
    }

    template <class T2>
    class B {
    }

    template <class T3>
    class C {
    }

    template <class T4>
    class D {
    }

    template <class T>
    class ScopeTest {
    private:
        T<T1, B<T1, D<T4, float>>>::var;
    }

    // Containment tree
    
    // Diagram:

    // Containment relationship
    
};
```
Partial template instantiation

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

```
template <class T, class U, class V>
class PT {};
template <class A, class B>
class PT<B, int, A> {};
```

Template specialization

C++ Template specialization uses the same mapping as Template Instantiation.
Forward class declaration

The example code is declared in A.h file. The file component A.h has the «use» association applied by «C++Include» stereotype with "Class Forward" tag value.
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.
#include "B.h"
#include <E.h>
class A {
private:
    B* b;
    E* e;
};

### Specification for `#include "B.h"`

### Specification for `#include <E.h>`
The example code is declared in D.cpp file.

```cpp
#include <E.h>
class B;
class D {
  private: B* b;
  E* e;
}
```

Specification for `#include <E.h>`

```
#include <E.h>
```

Specification for class B;

```
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 54 -- Open local project Activity Diagram
Open teamwork project

- Open teamwork project
- Project is opened completely

[version > 11.6]

[Version <= 11.5]

[not has permission to edit]
- Show error dialog about cannot translate project and C++ code engineering set may not work correctly

[has permission to edit]
- Show warning dialog about this project needs to be translated
- [cannot lock all elements] -> Show error dialog about cannot lock all elements and this must be done manually by invoking menu item
- [can lock all elements] -> Translate

- Show the list of elements that need to be unlock
- Project is opened without translation

- Project is translated completely

Figure 55 -- Open teamwork project Activity Diagram
Import MagicDraw project

Figure 56 -- Import MagicDraw project Activity Diagram
Use module

![Activity Diagram](Image)

**Figure 57 -- Use module 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 59 -- Class Language Properties*
Class - Class key

<table>
<thead>
<tr>
<th>Old value</th>
<th>Translation</th>
</tr>
</thead>
<tbody>
<tr>
<td>class</td>
<td>no change</td>
</tr>
<tr>
<td>struct</td>
<td>Apply the «C++Struct» stereotype.</td>
</tr>
</tbody>
</table>

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 60 -- Operation Example in Class Diagram](image1)

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

![Figure 61 -- Translated Operation in Class Diagram](image2)

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

![Figure 62 -- Operation Language Properties (Operation)](image3)

There are Initialization list and Explicit modifier in Operation Language Properties that need to be translated and apply the «C++Constructor» stereotype.
Figure 63 -- 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 64 -- Operation Initialization list

<table>
<thead>
<tr>
<th>Old value</th>
<th>Translation</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;empty&gt;</td>
<td>no change.</td>
</tr>
<tr>
<td>myAttribute(n)</td>
<td>Apply the «C++Constructor» stereotype and set initial</td>
</tr>
<tr>
<td></td>
<td>ization list tag value to myAttribute(n).</td>
</tr>
</tbody>
</table>
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`.

Figure 65 -- Operation Inline modifier
Operation - Virtual Modifier

![Image of the operation settings interface]

Figure 66 -- Operation Virtual modifier

<table>
<thead>
<tr>
<th>Old value</th>
<th>Translation</th>
</tr>
</thead>
<tbody>
<tr>
<td>not virtual</td>
<td>Apply the «C++Operation» stereotype and set virtual tag value to false.</td>
</tr>
<tr>
<td>virtual</td>
<td>Apply the «C++Operation» stereotype and set virtual tag value to true.</td>
</tr>
</tbody>
</table>
Operation - Explicit Modifier

Figure 67 -- Operation Explicit modifier

<table>
<thead>
<tr>
<th>Old value</th>
<th>Translation</th>
</tr>
</thead>
<tbody>
<tr>
<td>not explicit</td>
<td>Apply the «C++Constructor» stereotype and set explicit tag value to false.</td>
</tr>
<tr>
<td>explicit</td>
<td>Apply the «C++Constructor» stereotype and set explicit tag value to true.</td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>ReturnValueChangeabilityClass</th>
</tr>
</thead>
<tbody>
<tr>
<td>+noParamOperation()</td>
</tr>
<tr>
<td>+noReturnParamOperation( myParam : int[] )</td>
</tr>
<tr>
<td>+returnParamOperation() : void</td>
</tr>
</tbody>
</table>

**Figure 68 -- Return value changeability Example in Class Diagram**

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

<table>
<thead>
<tr>
<th>ReturnValueChangeabilityClass</th>
</tr>
</thead>
<tbody>
<tr>
<td>+noParamOperation()</td>
</tr>
<tr>
<td>+noReturnParamOperation( myParam : int[0..] )</td>
</tr>
<tr>
<td>+returnParamOperation() : void&quot;const &quot;$</td>
</tr>
</tbody>
</table>

**Figure 69 -- Translated Return value changeability in Class Diagram**

The Return value changeability Language Properties is being shown in the figure below.
Figure 70 -- Operation Return value changeability

<table>
<thead>
<tr>
<th>Old value</th>
<th>Translation</th>
</tr>
</thead>
<tbody>
<tr>
<td>none</td>
<td>no change.</td>
</tr>
<tr>
<td><code>const</code>, operation doesn't have parameter.</td>
<td>Create one return type parameter and set <strong>Type Modifier</strong> to <strong>const</strong>.</td>
</tr>
</tbody>
</table>

**const**, operation has parameter but does not have return type parameter. Create one return type parameter and set **Type Modifier** to **const**.
**Attribute**

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

const, operation has return type parameter.  
Set **Type Modifier** in return type parameter to **const**.

The Model that is being shown in the figure below is translation.
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 72 -- Translated Attribute in Class Diagram**

**Figure 73 -- Attribute Language Properties**

**Attribute - Mutable**

**Figure 74 -- Attribute Mutable**
**Attribute - Volatile**

<table>
<thead>
<tr>
<th>Old value</th>
<th>Translation</th>
</tr>
</thead>
<tbody>
<tr>
<td>false</td>
<td>Apply the «C++Attribute» stereotype and set mutable tag value to false.</td>
</tr>
<tr>
<td>true</td>
<td>Apply the «C++Attribute» stereotype and set mutable tag value to true.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Old value</th>
<th>Translation</th>
</tr>
</thead>
<tbody>
<tr>
<td>false</td>
<td>no change.</td>
</tr>
<tr>
<td>true</td>
<td>Set Type Modifier to volatile $volatile$.</td>
</tr>
</tbody>
</table>
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

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 78 -- Attribute Pointer to function

<table>
<thead>
<tr>
<th>Old value</th>
<th>Translation</th>
</tr>
</thead>
<tbody>
<tr>
<td>false</td>
<td>no change.</td>
</tr>
<tr>
<td>true</td>
<td>no change.</td>
</tr>
</tbody>
</table>
Figure 79 -- Attribute Container

<table>
<thead>
<tr>
<th>Old value</th>
<th>Translation</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;empty&gt;</td>
<td>no change.</td>
</tr>
<tr>
<td>vector&lt;T&gt;</td>
<td>Apply the «C++Attribute» stereotype and set container tag value to vector&lt;T&gt;.</td>
</tr>
<tr>
<td>list&lt;T&gt;</td>
<td>Apply the «C++Attribute» stereotype and set container tag value to list&lt;T&gt;.</td>
</tr>
<tr>
<td>map&lt;Key, T, Compare&gt;</td>
<td>Apply the «C++Attribute» stereotype and set container tag value to map&lt;Key, T, Compare&gt;.</td>
</tr>
<tr>
<td>stack&lt;T&gt;</td>
<td>Apply the «C++Attribute» stereotype and set container tag value to stack&lt;T&gt;.</td>
</tr>
<tr>
<td>multimap&lt;Key, T, Compare&gt;</td>
<td>Apply the «C++Attribute» stereotype and set container tag value to multimap&lt;Key, T, Compare&gt;.</td>
</tr>
<tr>
<td>set&lt;Key, Compare&gt;</td>
<td>Apply the «C++Attribute» stereotype and set container tag value to set&lt;Key, Compare&gt;.</td>
</tr>
</tbody>
</table>
Generalization

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

![Figure 80 -- Generalization Example in Class Diagram](image)

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

![Figure 81 -- Generalization Language Properties](image)
### Generalization - Inheritance type

**Figure 82 -- Generalization Inheritance type**

<table>
<thead>
<tr>
<th>Old value</th>
<th>Translation</th>
</tr>
</thead>
<tbody>
<tr>
<td>public</td>
<td>Apply the «C++Generalization» stereotype and set inheritance visibility tag value to <strong>public</strong>.</td>
</tr>
<tr>
<td>protected</td>
<td>Apply the «C++Generalization» stereotype and set inheritance visibility tag value to <strong>protected</strong>.</td>
</tr>
<tr>
<td>private</td>
<td>Apply the «C++Generalization» stereotype and set inheritance visibility tag value to <strong>private</strong>.</td>
</tr>
</tbody>
</table>
Generalization - Virtual modifier

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

Array

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

---

**Type Modifiers**

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

**Array**

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

<table>
<thead>
<tr>
<th>Old value</th>
<th>Translation</th>
</tr>
</thead>
<tbody>
<tr>
<td>![AttributeClass -myAttribute : int&quot;[15]&quot;&quot;]</td>
<td>![AttributeClass -myAttribute : int [0..*] ]</td>
</tr>
</tbody>
</table>

Apply the «C++Attribute» stereotype and set array tag value to [15].
Set Multiplicity field to 0..*.
Remove [15] from type modifier field.
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.
The «C++EnumerationLiteral» stereotype is being shown in the figure below.

Figure 87 -- «C++EnumerationLiteral» stereotype

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

Figure 88 -- Enumeration Literal Tag Value

<table>
<thead>
<tr>
<th>Old value</th>
<th>Translation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enumeration Literal applies the «C++EnumerationLiteral» stereotype and sets C++Initializer tag value to 0.</td>
<td>Apply the «C++LiteralValue» and set value tag value to 0. Remove the «C++EnumerationLiteral» stereotype and C++Initializer tag value from Enumeration Literal.</td>
</tr>
</tbody>
</table>
The Model that is being shown in the figure below is a translation.

```c++
<<enumeration>>
EnumClass
literal
```

*Figure 89 -- 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.

```c++
<<C++Namespace>>
MyPackage
{unique namespace name = "myNamespace"}
```

*Figure 90 -- «C++Namespace» stereotype Example in Class Diagram*
The «C++Namespace» stereotype is being shown in the figure below.

![Figure 91 -- «C++Namespace» stereotype](image)

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

![Figure 92 -- unique namespace name tag value](image)

<table>
<thead>
<tr>
<th>Old value</th>
<th>Translation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Package applies the «C++Namespace» stereotype in old profile (C++ Profile) and sets <strong>unique namespace name</strong> tag value to <strong>myNamespace</strong>.</td>
<td>Apply the «C++Namespace» stereotype in new profile (c++ ANSI profile) and set <strong>unique namespace name</strong> tag value to <strong>myNamespace</strong>. Remove the «C++Namespace» stereotype (C++ Profile) and <strong>unique namespace name</strong> tag value.</td>
</tr>
</tbody>
</table>
The «C++Typename» stereotype can apply to Template Parameters. Therefore, Elements that have template parameters could apply this stereotype.
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 93 -- Elements that can have template parameters

Figure 94 -- `<<C++Typename>>` stereotype
The «C++TemplateParameter» stereotype is being shown in the figure below.

**Figure 95 -- «C++TemplateParameter» stereotype**

<table>
<thead>
<tr>
<th>Old value</th>
<th>Translation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Template Parameter does not apply the «C++Typename» stereotype.</td>
<td>Apply the «C++TemplateParameter» stereotype and set type keyword tag value to class.</td>
</tr>
<tr>
<td>Template Parameter applies the «C++Typename» stereotype.</td>
<td>Apply the «C++TemplateParameter» stereotype and set type keyword tag value to typename.</td>
</tr>
<tr>
<td></td>
<td>Remove the «C++Typename» stereotype.</td>
</tr>
</tbody>
</table>
**«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 96 -- «constructor» and «destructor» stereotype Example in Class Diagram](image)

**«constructor»**

The «constructor» stereotype is being shown in the figure below.

![Figure 97 -- «constructor» stereotype](image)
### Old value

Operation applies the «constructor» stereotype in UML Standard Profile.

### Translation

- Apply the «C++Constructor» stereotype (c++ ANSI profile).
- Remove the «constructor» stereotype (UML Standard Profile).

---

The «destroyer» stereotype is being shown in the figure below.

---

*Figure 98 -- «destroyer» stereotype*
The «C++Operation» stereotype can apply to an operation.

```cpp
class Operation {
    int myAttribute;

public:
    Operation() {
        throw C++ThrownExceptions()
    }

    void myOperation() {
        C++ThrownExceptions = none;
    }
};
```

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.
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.

![Class - myClass](image)

*Figure 101 -- Constructor and Destructor name Example in Class Diagram*

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

<table>
<thead>
<tr>
<th>Old value</th>
<th>Translation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operation name is the same as the owner's name.</td>
<td>Apply the <code>«C++Constructor»</code> stereotype.</td>
</tr>
</tbody>
</table>
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 103 -- bool data type Example in Class Diagram](image)

**Attribute type**

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

![Figure 104 -- old bool data type in Attribute type](image)

<table>
<thead>
<tr>
<th>Old value</th>
<th>Translation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Old bool data type in old profile (C++ Profile).</td>
<td>Change to the new bool data type in new profile</td>
</tr>
<tr>
<td></td>
<td>(c++ ANSI profile).</td>
</tr>
</tbody>
</table>
Parameter type

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

**Figure 105 -- old bool data type in Parameter type**

<table>
<thead>
<tr>
<th>Old value</th>
<th>Translation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Old bool data type in old profile (C++ Profile).</td>
<td>Change to the new bool data type in new profile</td>
</tr>
<tr>
<td></td>
<td>(c++ ANSI profile).</td>
</tr>
</tbody>
</table>
Return type

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

**Figure 106 -- Figure 57 -- old bool type in Return type**

<table>
<thead>
<tr>
<th>Old value</th>
<th>Translation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Old bool data type in old profile (C++ Profile).</td>
<td>Change to the new bool data type in new profile (c++ ANSI profile).</td>
</tr>
</tbody>
</table>
Actual in Template Parameter Substitution

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

<table>
<thead>
<tr>
<th>Old value</th>
<th>Translation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Old bool data type in old profile (C++ Profile).</td>
<td>Change to the new bool data type in new profile (c++ ANSI profile).</td>
</tr>
</tbody>
</table>
Default in Classifier Template Parameter

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

*Figure 108 -- old bool data type in Default in Classifier Template Parameter*
## Conversion from old project version

<table>
<thead>
<tr>
<th>Old value</th>
<th>Translation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Old bool data type in old profile (C++ Profile).</td>
<td>Change to the new bool data type in new profile (c++ ANSI profile).</td>
</tr>
</tbody>
</table>
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 109 -- tag values in Operation](image1)

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

![Figure 110 -- tag values in Constructor](image2)
There is **C++ThrownExceptions** tag value that translate to Operation tag value.

![Figure 111 -- Throw exception tag value in Operation](image1)

### 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 112 -- tag values in Attribute](image2)
Generalization

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

![Figure 113 -- tag values in Generalization](image1)

Enumeration literal

There is value from «C++EnumerationLiteral» stereotype that translate to Enumeration literal tag value.

![Figure 114 -- tag value in Enumeration literal](image2)

Namespace

There is value from «C++Namespace» stereotype that translate to Namespace tag value.
Template parameter

There is value from «C++Typename» stereotype that translate to Template parameter tag value.

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.
Destroyer

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 cannot 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

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.

```cpp
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**

<table>
<thead>
<tr>
<th>Model</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1" alt="Model Diagram" /></td>
<td><code>#include &quot;B.h&quot;</code>&lt;br&gt;class A{&lt;br&gt;   B i;&lt;br&gt;   C* j;&lt;br&gt;};`</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Model</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image2" alt="Model Diagram" /></td>
<td><code>#include &quot;B.h&quot;</code>&lt;br&gt;class A:B{&lt;br&gt;};`</td>
</tr>
</tbody>
</table>
## Parameter

<table>
<thead>
<tr>
<th>Model</th>
<th>Code</th>
</tr>
</thead>
</table>
| ![Parameter Model Diagram](image1) | ```
#include "B.h"

class A{
  public:
    void func(B i);
};
``` |

## Template Parameter Substitution

<table>
<thead>
<tr>
<th>Model</th>
<th>Code</th>
</tr>
</thead>
</table>
| ![Template Parameter Substitution Model Diagram](image2) | ```
#include "B.h"

template <class Tclass>
class TP{};

class A{
  TP<B> i;
};
``` |
Template Parameter Default Value

<table>
<thead>
<tr>
<th>Model</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1.png" alt="Model Diagram" /></td>
<td><code>#include &quot;B.h&quot;</code>&lt;br&gt;template &lt;class TClass=B&gt;&lt;br&gt;class A{};</td>
</tr>
</tbody>
</table>

Usage

<table>
<thead>
<tr>
<th>Model</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image2.png" alt="Model Diagram" /></td>
<td><code>#include &quot;B.h&quot;</code>&lt;br&gt;class A{&lt;br&gt;}</td>
</tr>
</tbody>
</table>
Project Option and Code Generation Options

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

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.
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.

<table>
<thead>
<tr>
<th>Automatic remove unnecessary usage relationship</th>
<th>Model</th>
<th>Code</th>
</tr>
</thead>
</table>
| **Selected**                                  | ![Model Diagram](image1.png) | //X.h
#include "Y.h"
class X{}; |
| **Not Selected**                              | ![Model Diagram](image2.png) | //X.h
class X{}; |
New in MagicDraw 14.0

Support C++ dialects

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

- ANSI – conform to ISO/IEC 14882 specification for C++ programming language
- 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.

![New dialects in C++ CE](image)

*Figure 120 -- New dialects in C++ CE*
# CG Properties Editor

![CG Properties Editor](image)

*Figure 121 -- New properties*
### New in MagicDraw 16.8

**Doxygen-after-member documentation**

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

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

<table>
<thead>
<tr>
<th>Property name</th>
<th>Description</th>
</tr>
</thead>
</table>
| **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:  
  ```cpp
  class A {
    B::X i;
  };
  ```  
The parser does not have enough information to decide whether B is class or namespace. |
| **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`. |

**NOTE** This feature only works for import-code-only mode.
@see on documentation

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

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

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

Symbian macro

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

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

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

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

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

Tutorial

Type Modifier

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

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

```c++
class A
{
    const int* const i;
};
```
type modifier must be set to “const $* const” as shown in the figure below.

![Type Modifier Example](image)

**Figure 123 -- 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»

```
<<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.

```
<<C++Global>>
+i: int
+(char): void
```

See "Global functions and variables", on page 88 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.

```c++
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.

```cpp
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.

```cpp
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.

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

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

2. Apply stereotype «C++Friend» to the dependency.

See Friend declaration for mapping example.
How to specify component to generate code to

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

However, MagicDraw allows user to specify the component to generate the model into by using Realization. See the figure below for better understanding.
When drawing the Realization from component A.h to class B. Class B will be added to code engineering set as you can see in the red circle, which means that class B will be generated in A.h.

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

@see support for import-code-only mode

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

- `@see`
- `@see`
- `@sa`
- `\sa`
Example:

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

Navigable short cut from model link in documentation

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

- **Ctrl + Alt + click**
  MagicDraw will navigate to the documentation of the model that the link point to.
This chapter describes how to set include paths for include system file.

```cpp
// TestWinRoundtrip.cpp - Defines the entry point for the console application.

#include "stdafx.h"
#include <iostream>

class A {
public:
    void print() {
        std::cout << "Hello World" << std::endl;
    }

    int main(int argc, char* argv[]) {
        A* myA = new A();
        myA->print();
        return 0;
    }
};
```

Figure 124 -- Include system file example

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

To set include path

1. From the Options menu choose Project.
2. Expand Code Engineering, choose C++ Language Options. Select the Include path check-box. 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.
Then select the code-engineering set and then select Properties.
Finally set the “Parse includes” property to “true”.

Note

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

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

MagicDraw has already defined some default skipped macros for QT.
Microsoft C++ Profiles

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

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 \_	exttt{int8}, \_	exttt{int16}, and \_	exttt{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.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure128}
\caption{Microsoft Visual C++ Data Type}
\end{figure}
Stereotype

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

Figure 129 -- Microsoft Visual C++ Stereotypes
VC++Class

«VC++Class» inherits from «C++Class» and «VC++StorageClass»

<table>
<thead>
<tr>
<th>Name</th>
<th>Meta class</th>
<th>Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>VC++Class</td>
<td>Class</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Tag</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>inheritance</td>
<td>inheritanceType[0..1] = single (Enumeration) See inheritanceType</td>
<td>Represent the utilization of VC++ keywords, __single_inheritance, __multiple_inheritance and __virtual_inheritance.</td>
</tr>
<tr>
<td>abstract</td>
<td>boolean[1] = false</td>
<td>Represent the utilization of keyword __abstract or abstract, depending on C++ dialects.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Inherited tag</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>declspecModifier</td>
<td>String</td>
<td>For keeping extended declaration modifier of __declspec See Extended storage-class attributes with __declspec</td>
</tr>
<tr>
<td>C++Attributes</td>
<td>String</td>
<td>For keeping C++ Attributes</td>
</tr>
</tbody>
</table>

VC++Struct

«VC++Struct» inherits from «C++Struct» and «VC++StorageClass»

<table>
<thead>
<tr>
<th>Name</th>
<th>Meta class</th>
<th>Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>VC++Struct</td>
<td>Class</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Inherited tag</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>declspecModifier</td>
<td>String</td>
<td>For keeping extended declaration modifier of __declspec See Extended storage-class attributes with __declspec</td>
</tr>
<tr>
<td>C++Attributes</td>
<td>String</td>
<td>For keeping C++ Attributes</td>
</tr>
</tbody>
</table>
### VC++Union

<table>
<thead>
<tr>
<th>Name</th>
<th>Meta class</th>
<th>Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>VC++Union</td>
<td>Class</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Inherited tag</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>declspecModifier «VC++StorageClass»</td>
<td>String</td>
<td>For keeping extended declaration modifier of __declspec See Extended storage-class attributes with __declspec</td>
</tr>
<tr>
<td>C++Attributes «VC++Element»</td>
<td>String</td>
<td>For keeping C++ Attributes</td>
</tr>
</tbody>
</table>

### VC++Enumeration

<table>
<thead>
<tr>
<th>Name</th>
<th>Meta class</th>
<th>Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>VC++Enumeration</td>
<td>Enumeration</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Tag</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>Classifier[0..1]</td>
<td>Represent the type of enumeration literal</td>
</tr>
</tbody>
</table>

### VC++Typedef

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Meta class</th>
<th>Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>VC++Typedef</td>
<td>Class</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Inherited tag</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>C++Attributes «VC++Element»</td>
<td>String</td>
<td>For keeping C++ Attributes</td>
</tr>
</tbody>
</table>
### VC++Attribute

<table>
<thead>
<tr>
<th>Name</th>
<th>Meta class</th>
<th>Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>VC++Attribute</td>
<td>Property</td>
<td></td>
</tr>
</tbody>
</table>

**Inherited tag**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>C++Attributes</td>
<td>String</td>
<td>For keeping C++ Attributes</td>
</tr>
<tr>
<td>«VC++Element»</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### VC++Operation

<table>
<thead>
<tr>
<th>Name</th>
<th>Meta class</th>
<th>Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>VC++Operation</td>
<td>Operation</td>
<td></td>
</tr>
</tbody>
</table>

**Tag**

<table>
<thead>
<tr>
<th>Tag</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
</table>
| callingConvention | callingConvention [0..1] (Enumeration) | Represent the utilization of keywords.  
|                 |                       | • __cdecl  
|                 |                       | • __clrcall  
|                 |                       | • __stdcall  
|                 |                       | • __fastcall  
|                 |                       | • __thiscall  |

**Inherited tag**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
</table>
| declspecModifier | String      | For keeping extended declaration modifier of __declspec  
| «VC++StorageClass» |             | See Extended storage-class attributes with __declspec |
| C++Attributes    | String      | For keeping C++ Attributes |
| «VC++Element»    |             |              |
### VC++Parameter

<table>
<thead>
<tr>
<th>Name</th>
<th>Meta class</th>
<th>Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>VC++Parameter</td>
<td>Parameter</td>
<td></td>
</tr>
</tbody>
</table>

### VC++StorageClass

«VC++StorageClass» is an abstract stereotype corresponding to extended declaration modifier of `__declspec`.

<table>
<thead>
<tr>
<th>Name</th>
<th>Meta class</th>
<th>Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>VC++StorageClass</td>
<td>Element</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Tag</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>declspecModifier</td>
<td>String</td>
<td>For keeping extended declaration modifier of <code>__declspec</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td>See <a href="#">Extended storage-class attributes with __declspec</a></td>
</tr>
</tbody>
</table>

### VC++Element

<table>
<thead>
<tr>
<th>Name</th>
<th>Meta class</th>
<th>Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>VC++Element</td>
<td>Element</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Tag</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>C++Attributes</td>
<td>String</td>
<td>For keeping C++ Attributes</td>
</tr>
</tbody>
</table>
### VC++Interface

<table>
<thead>
<tr>
<th>Name</th>
<th>Meta class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VC++Interface</td>
<td>Interface</td>
<td>Represent interface declaration with <code>__interface</code> keyword.</td>
</tr>
</tbody>
</table>

### VC++Event

<table>
<thead>
<tr>
<th>Name</th>
<th>Meta class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VC++Event</td>
<td>Operation, Property</td>
<td>Represent interface, method or member declaration with <code>__event</code> keyword.</td>
</tr>
</tbody>
</table>

#### Tag

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

### Enumeration

**inheritanceType**

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

<table>
<thead>
<tr>
<th>Literal</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>single</td>
<td>Representation of <code>__single_inheritance</code> keyword</td>
</tr>
<tr>
<td>multiple</td>
<td>Representation of <code>__multiple_inheritance</code> keyword</td>
</tr>
<tr>
<td>Virtual</td>
<td>Representation of <code>__virtual_inheritance</code> keyword</td>
</tr>
</tbody>
</table>

### callingConvention

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

<table>
<thead>
<tr>
<th>Literal</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>__cdecl</td>
<td>Represent the utilization of <code>__cdecl</code> modifier.</td>
</tr>
<tr>
<td>__clrcall</td>
<td>Represent the utilization of <code>__clrcall</code> modifier.</td>
</tr>
<tr>
<td>__stdcall</td>
<td>Represent the utilization of <code>__stdcall</code> modifier.</td>
</tr>
<tr>
<td>__fastcall</td>
<td>Represent the utilization of <code>__fastcall</code> modifier.</td>
</tr>
<tr>
<td>__thiscall</td>
<td>Represent the utilization of <code>__thiscall</code> modifier.</td>
</tr>
</tbody>
</table>
C++/CLI Profile

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

C++CLIClass

<table>
<thead>
<tr>
<th>Name</th>
<th>Meta class</th>
<th>Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>C++CLIClass</td>
<td>Class</td>
<td></td>
</tr>
</tbody>
</table>

Tag

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLI Type</td>
<td>Represent the utilization of ref class or value class keywords for defining CLR class.</td>
</tr>
</tbody>
</table>
### Inherited tag

<table>
<thead>
<tr>
<th>Inherited tag</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>inheritance</td>
<td>inheritanceType[0..1] = single</td>
<td>Represent the utilization of VC++ keywords, __single_inheritance, __multiple_inheritance and</td>
</tr>
<tr>
<td>«VC++Class»</td>
<td>(Enumeration)</td>
<td>_virtual_inheritance.</td>
</tr>
<tr>
<td>decspecModifier</td>
<td>String</td>
<td>For keeping extended declaration modifier of __declspec</td>
</tr>
<tr>
<td>«VC++StorageClass»</td>
<td></td>
<td>See Extended storage-class attributes with __declspec</td>
</tr>
<tr>
<td>C++Attributes</td>
<td>String</td>
<td>For keeping C++ Attributes</td>
</tr>
<tr>
<td>«VC++Element»</td>
<td></td>
<td></td>
</tr>
<tr>
<td>«VC++Class»</td>
<td></td>
<td></td>
</tr>
<tr>
<td>accessSpecifier</td>
<td>CLIvisibilityKind[0..1] (Enumeration)</td>
<td>Represent the usage of visibility kind introduced in C++/CLI including</td>
</tr>
<tr>
<td>«C++CLIClassMember»</td>
<td></td>
<td>• internal</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• protected public</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• public protected</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• private protected</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• protected private</td>
</tr>
</tbody>
</table>

### C++CLIStruct

<table>
<thead>
<tr>
<th>Name</th>
<th>Meta class</th>
<th>Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>C++CLIStruct</td>
<td>Class</td>
<td></td>
</tr>
</tbody>
</table>

### Inherited tag

<table>
<thead>
<tr>
<th>Inherited tag</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>managedType</td>
<td>CLI Type[1] = ref</td>
<td>Represent the utilization of ref class or value class keywords for defining CLR class.</td>
</tr>
<tr>
<td>«C++CLIClass»</td>
<td>(Enumeration)</td>
<td>See CLI Type</td>
</tr>
<tr>
<td>«C++CLIClass»</td>
<td></td>
<td></td>
</tr>
<tr>
<td>inheritance</td>
<td>inheritanceType[0..1] = single</td>
<td>Represent the utilization of VC++ keywords, __single_inheritance, __multiple_inheritance and</td>
</tr>
<tr>
<td>«VC++Class»</td>
<td>(Enumeration)</td>
<td>_virtual_inheritance.</td>
</tr>
<tr>
<td>decspecModifier</td>
<td>String</td>
<td>For keeping extended declaration modifier of __declspec</td>
</tr>
<tr>
<td>«VC++StorageClass»</td>
<td></td>
<td>See Extended storage-class attributes with __declspec</td>
</tr>
</tbody>
</table>
### Inherited tag

<table>
<thead>
<tr>
<th>Name</th>
<th>Meta class</th>
<th>Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>VC++Attributes</td>
<td>String</td>
<td>For keeping VC++ Attributes</td>
</tr>
<tr>
<td>«VC++Element»</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### C++CLIEnumeration

<table>
<thead>
<tr>
<th>Name</th>
<th>Meta class</th>
<th>Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>C++CLIEnumeration</td>
<td>Enumeration</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Tag</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>declarationKeyword</td>
<td>classKey[1] = class (Enumeration) See classKey</td>
<td>Represent the utilization of enum class or enum struct. keywords. See enum class, enum struct</td>
</tr>
</tbody>
</table>

### Inherited tag

<table>
<thead>
<tr>
<th>Name</th>
<th>Meta class</th>
<th>Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>type</td>
<td>Classifier[0..1]</td>
<td>Represent the type of enumeration literal</td>
</tr>
<tr>
<td>«VC++Enumeration»</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### C++CLIInterface

<table>
<thead>
<tr>
<th>Name</th>
<th>Meta class</th>
<th>Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>C++CLIIterface</td>
<td>Interface</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Tag</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>declarationKeyword</td>
<td>classKey[1] = class (Enumeration) See classKey</td>
<td>Represent the utilization of interface class or interface struct. keywords. See ref class, ref struct, value class, value struct, interface class, interface struct</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Inherited tag</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>declspecModifier</td>
<td>String</td>
<td>For keeping extended declaration modifier of __declspec See Extended storage-class attributes with __declspec</td>
</tr>
<tr>
<td>«VC++StorageClass»</td>
<td></td>
<td></td>
</tr>
<tr>
<td>++Attributes</td>
<td>String</td>
<td>For keeping C++ Attributes</td>
</tr>
<tr>
<td>«VC++Element»</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### C++CLIClassMember

<table>
<thead>
<tr>
<th>Name</th>
<th>Meta class</th>
<th>Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>C++CLIClassMember</td>
<td>Operation, Property</td>
<td>This stereotype is an abstract stereotype for keeping accessSpecifier tag definition.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Tag</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>accessSpecifier</td>
<td>CLIvisibilityKind[0..1] (Enumeration)</td>
<td>Represent the usage of visibility kind introduce in C++/CLI including</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• internal</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• protected public</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• public protected</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• private protected</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• protected private</td>
</tr>
</tbody>
</table>

### C++CLIAttribute

<table>
<thead>
<tr>
<th>Name</th>
<th>Meta class</th>
<th>Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>C++CLIAttribute</td>
<td>Property</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Tag</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>field</td>
<td>attributeField[0..1]</td>
<td>Represent the usage of initonly or literal keywords.</td>
</tr>
<tr>
<td></td>
<td>(Enumeration)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>See attributeField</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Inherited tag</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>accessSpecifier</td>
<td>CLIvisibilityKind[0..1] (Enumeration)</td>
<td>Represent the usage of visibility kind introduce in C++/CLI including</td>
</tr>
<tr>
<td>«C++CLIClassMember»</td>
<td></td>
<td>• internal</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• protected public</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• public protected</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• private protected</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• protected private</td>
</tr>
<tr>
<td>declspecModifier</td>
<td>String</td>
<td>For keeping extended declaration modifier of declspec</td>
</tr>
<tr>
<td>«VC++StorageClass»</td>
<td></td>
<td>See Extended storage-class attributes with declspec</td>
</tr>
<tr>
<td>C++Attributes</td>
<td>String</td>
<td>For keeping C++ Attributes</td>
</tr>
<tr>
<td>«VC++Element»</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### C++CLIOperation

<table>
<thead>
<tr>
<th>Name</th>
<th>Meta class</th>
<th>Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>C++CLIOperation</td>
<td>Operation</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Tag</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>overrideSpecifier</td>
<td>operationOverrideSpecifier[0..1] (Enumeration)</td>
<td>Represent the utilization of C++/CLI operation override specifiers, <code>new</code> and <code>override</code>.</td>
</tr>
<tr>
<td>explicitOverride</td>
<td>Operation[0..*]</td>
<td>Represent the usage of explicit override feature.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Inherited tag</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
</table>
| accessSpecifier «C++CLIClassMember» | CLIVisibilityKind[0..1] (Enumeration) | Represent the usage of visibility kind introduce in C++/CLI including
• internal
• protected public
• public protected
• private protected
• protected private |
| callingConvention «VC++Operation» | callingConvention[0..1] | Represent the utilization of keywords.
• __cdecl
• __clrcall
• __stdcall
• __fastcall
• __thiscall |
| C++Attributes «VC++Element» | String | For keeping C++ Attributes |
C++CLIProperty

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Meta class</th>
<th>Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>C++CLIProperty</td>
<td>Property</td>
<td>The stereotype must not be applied to the same attribute as «C++Attribute» or other stereotypes derived from «C++Attribute»</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Tag</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>propertyIndexType</td>
<td>String[0..*]</td>
<td>index type keeps the list of property index types.</td>
</tr>
<tr>
<td>isVirtualProperty</td>
<td>boolean[1] = false</td>
<td>Specify whether the property is virtual or not.</td>
</tr>
<tr>
<td>propertyType</td>
<td>PropertyType[1] = property data member</td>
<td>Specify property type between: • property data member • property block with get • property block with set • property block with get and set</td>
</tr>
</tbody>
</table>

Inherited tag

<table>
<thead>
<tr>
<th>Tag</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>accessSpecifier</td>
<td>CLIVisibilityKind [0..1] (Enumeration)</td>
<td>Represent the usage of visibility kind introduce in C++/CLI including • internal • protected public • public protected • private protected • protected private</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Name</th>
<th>Meta class</th>
<th>Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>C++CLIDelegate</td>
<td>Class</td>
<td>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.</td>
</tr>
</tbody>
</table>
### C++CLIEvent

<table>
<thead>
<tr>
<th>Name</th>
<th>Meta class</th>
<th>Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>C++CLIEvent</td>
<td>Property</td>
<td>«C++CLIEvent» can be applied only when the attribute type is «C++CLIDelegate». Represent the utilization of event keyword.</td>
</tr>
</tbody>
</table>

#### Tag

<table>
<thead>
<tr>
<th>Tag</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>isCLIVirtualEvent</td>
<td>boolean[1] = false</td>
<td>Specify whether the event is virtual or not.</td>
</tr>
</tbody>
</table>

#### Inherited tag

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>accessSpecifier</td>
<td>CLIVisibilityKind[0..1] (Enumeration)</td>
<td>Represent the usage of visibility kind introduce in C++/CLI including</td>
</tr>
<tr>
<td>«C++CLIClassMember»</td>
<td></td>
<td>• internal</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• protected public</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• public protected</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• private protected</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• protected private</td>
</tr>
</tbody>
</table>

### C++CLIGeneric

<table>
<thead>
<tr>
<th>Name</th>
<th>Meta class</th>
<th>Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>C++CLIGeneric</td>
<td>Class, Interface, Operation, Property</td>
<td>Represent the usage of generic keyword.</td>
</tr>
</tbody>
</table>

---

*Copyright © 1998-2015 No Magic, Inc.*
C++ CLIGenericParameteredElement

<table>
<thead>
<tr>
<th>Name</th>
<th>Meta class</th>
<th>Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>C++CLIGenericParameteredElement</td>
<td>Class</td>
<td></td>
</tr>
</tbody>
</table>

Tag | Type                  | Description                                                                 |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>genericConstraintType</td>
<td>genericConstraintType[0..*] (Enumeration)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>See <code>genericConstraintType</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Specify generic constraint type including</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• ref class</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• ref struct</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• value class</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• value struct</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• gcnew</td>
<td></td>
</tr>
</tbody>
</table>

C++CLIGeneralization

<table>
<thead>
<tr>
<th>Name</th>
<th>Meta class</th>
<th>Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>C++CLIGeneralization</td>
<td>Generalization</td>
<td></td>
</tr>
</tbody>
</table>

Tag | Type                  | Description                                                                 |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>CLIInheritanceVisibility</td>
<td>C++CLIGeneralizationVisibility[1] = none (Enumeration)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>See <code>C++CLIGeneralizationVisibility</code></td>
<td></td>
</tr>
<tr>
<td>virtualInheritance</td>
<td>boolean[1] = false</td>
<td></td>
</tr>
</tbody>
</table>

**Enumeration**

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

<table>
<thead>
<tr>
<th>Literal</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ref</td>
<td>For defining a CLR reference class</td>
</tr>
<tr>
<td>value</td>
<td>For defining a CLR value class</td>
</tr>
</tbody>
</table>

**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.

<table>
<thead>
<tr>
<th>Literal</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>new</td>
<td>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.</td>
</tr>
<tr>
<td>override</td>
<td>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.</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Literal</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>initonly</td>
<td>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.</td>
</tr>
<tr>
<td>literal</td>
<td>Represent the utilization of literal keyword. It is the native equivalent of static const variable. Constraint: Is Static and Is Read Only must be set to true.</td>
</tr>
</tbody>
</table>

classKey

<table>
<thead>
<tr>
<th>Literal</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>class</td>
<td>Represent the keyword that has the word, class.</td>
</tr>
<tr>
<td>struct</td>
<td>Represent the keyword that has the word, struct.</td>
</tr>
</tbody>
</table>

Remark: enum class and enum struct are equivalent declarations.

interface class and interface struct are equivalent declarations.
C++ Managed Profile

**CLIvisibilityKind**

<table>
<thead>
<tr>
<th>Literal</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>internal</td>
<td>Represent the internal visibility.</td>
</tr>
<tr>
<td>protected public</td>
<td>Represent the protected public visibility.</td>
</tr>
<tr>
<td>public protected</td>
<td>Represent the public protected visibility.</td>
</tr>
<tr>
<td>private protected</td>
<td>Represent the private protected visibility.</td>
</tr>
<tr>
<td>protected private</td>
<td>Represent the protected private visibility.</td>
</tr>
</tbody>
</table>

**genericConstraintType**

<table>
<thead>
<tr>
<th>Literal</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ref class</td>
<td>Represent the usage of ref class keyword in generic constraint clause.</td>
</tr>
<tr>
<td>ref struct</td>
<td>Represent the usage of ref struct keyword in generic constraint clause.</td>
</tr>
<tr>
<td>value class</td>
<td>Represent the usage of value class keyword in generic constraint clause.</td>
</tr>
<tr>
<td>value struct</td>
<td>Represent the usage of value struct keyword in generic constraint clause.</td>
</tr>
<tr>
<td>gcnew</td>
<td>Represent the usage of gcnew keyword in generic constraint clause.</td>
</tr>
</tbody>
</table>

C++CLIGeneralizationVisibility

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

**NOTE** The profile table and description in this section does not include the tagged value inherited from other profiles.
Figure 131 -- C++ Managed Stereotypes

### C++ManagedClass

<table>
<thead>
<tr>
<th>Name</th>
<th>Meta class</th>
<th>Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>C++ManagedClass</td>
<td>Class</td>
<td>Represent the class declaration with version 1 of Managed Extension for C++.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Tag</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>managedType</td>
<td>managedType[1] = __gc (enumeration)</td>
<td>SeemanagedType</td>
</tr>
</tbody>
</table>

### C++ManagedStruct

<table>
<thead>
<tr>
<th>Name</th>
<th>Meta class</th>
<th>Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>C++ManagedStruct</td>
<td>Class</td>
<td>Represent the struct declaration with version 1 of Managed Extension for C++.</td>
</tr>
</tbody>
</table>
C++ ManagedOperation

<table>
<thead>
<tr>
<th>Name</th>
<th>Meta class</th>
<th>Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>C++ManagedOperation</td>
<td>Operation</td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Meta class</th>
<th>Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>C++ManagedDelegate</td>
<td>Class</td>
<td></td>
</tr>
</tbody>
</table>

Represent the delegate declaration with version 1 of Managed Extension for C++. It defines a reference type that can be used to encapsulate a method with a specific signature.

**Enumeration**

managedType

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

**Modeling with Microsoft Specific Profiles**

**CLR Data Type Keyword**

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

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

<table>
<thead>
<tr>
<th>Code</th>
<th>MD-UML</th>
</tr>
</thead>
<tbody>
<tr>
<td>ref class MyRefClass</td>
<td>&lt;&lt;C++CLIObject&gt;&gt; MyRefClass</td>
</tr>
<tr>
<td>{}</td>
<td>{ CLI Type = ref, abstract = false}</td>
</tr>
<tr>
<td></td>
<td>with «C++CLIClass» and tagged value CLI Type = ref</td>
</tr>
<tr>
<td>ref struct MyRefStruct</td>
<td>&lt;&lt;C++CLIObject&gt;&gt; MyRefStruct</td>
</tr>
<tr>
<td>{}</td>
<td>{ abstract = false, CLI Type = ref}</td>
</tr>
<tr>
<td></td>
<td>with «C++CLIStruct» and tagged value CLI Type = ref</td>
</tr>
<tr>
<td>value class MyValueClass</td>
<td>&lt;&lt;C++CLIObject&gt;&gt; MyValueClass</td>
</tr>
<tr>
<td>{}</td>
<td>{ CLI Type = value, abstract = false}</td>
</tr>
<tr>
<td></td>
<td>with «C++CLIClass» and tagged value CLI Type = value</td>
</tr>
<tr>
<td>value struct MyValueStruct</td>
<td>&lt;&lt;C++CLIObject&gt;&gt; MyValueStruct</td>
</tr>
<tr>
<td>{}</td>
<td>{ abstract = false, CLI Type = value}</td>
</tr>
<tr>
<td></td>
<td>with «C++CLIStruct» and tagged value CLI Type = value</td>
</tr>
<tr>
<td>interface class MyInterfaceClass</td>
<td>&lt;&lt;C++CLIObject&gt;&gt; MyInterfaceClass</td>
</tr>
<tr>
<td>{}</td>
<td>{ declarationKeyword = class}</td>
</tr>
<tr>
<td></td>
<td>with «C++CLIInterface» and tagged value declarationKeyword = class</td>
</tr>
<tr>
<td>interface struct MyInterfaceStruct</td>
<td>&lt;&lt;C++CLIObject&gt;&gt; MyInterfaceStruct</td>
</tr>
<tr>
<td>{}</td>
<td>{ declarationKeyword = struct}</td>
</tr>
<tr>
<td></td>
<td>with «C++CLIInterface» and tagged value declarationKeyword = struct</td>
</tr>
</tbody>
</table>

enum class, enum struct

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

**Code**

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

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

**MD-UML**

```cpp
<<C++CL Enumeration>>
EnumClassDay
{declarationKeyword = class, enumerationType = int}
```

**Code**

```cpp
property type property_name;   // property data member
modifier property type property_name {   // property block
    modifier void set(type);
    modifier type get();
}
```

**MD-UML**

```cpp
<<C++CL Property>>
propertyX
{propertyType = property data member}
<<C++CL Property>>
propertyY
{propertyType = property block with get and set}
<<C++CL Property>>
propertyZ [int, long]
{propertyType = property block with get and set, propertyIndexType = "int","long")
```

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

**MD-UML**

```cpp
MyClass
<<C++CL Property>> propertyX : String*""{propertyType = property data member}
<<C++CL Property>> propertyY : int{propertyType = property block with get and set}
<<C++CL Property>> propertyZ : int{propertyType = property block with get and set, propertyIndexType = "int","long")
```
delegate

access delegate function_declaration

access (optional)

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

function_declaration

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

To define an unbound delegate, the first parameter in function_declaration should be the type of the this pointer for the object.
```csharp
public delegate void MyDelegate(int i);
ref class A
{
    MyDelegate^ delInst;
};
```
event
t
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

dPublic deleGee void
MyDelegate (int);
ref class B {
    event MyDelegate^ MyEvent;
};
Override Specifiers

abstract

<table>
<thead>
<tr>
<th>Code</th>
<th>MD-UML</th>
</tr>
</thead>
<tbody>
<tr>
<td>ref class MyAbstractClass</td>
<td>&lt;&lt;C++CLIClass&gt;&gt;</td>
</tr>
<tr>
<td>abstract</td>
<td>MyAbstractClass</td>
</tr>
<tr>
<td></td>
<td>{</td>
</tr>
<tr>
<td></td>
<td>[abstract, CLI Type = ref]</td>
</tr>
<tr>
<td></td>
<td>with &lt;&lt;C++CLIClass&gt;&gt; and tagged value abstract=true</td>
</tr>
<tr>
<td>public:</td>
<td></td>
</tr>
<tr>
<td>virtual void func() abstract;</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
</tbody>
</table>

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

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

func() with «C++CLIOperation» and tagged value 
overrideSpecifier = new

code

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() { } // sub class cannot override g()
    virtual void g() sealed { }
};
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**
```cpp
generic <typename ItemType>
void G(int i) {}
```

**MD-UML**
```
<<C++Global>>

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

ItemType with «C++TemplateParameter» applied and set tagged value `type keyword = typename`.

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

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

**MD-UML**
```
<<C++CLStruct>>
MyStruct
{abstract = false,
 CL Type = ref}

<<C++CLGeneric>>+<Type1>G(i; Type1); void
<<C++CLGeneric>>+<Type2>H(i; int); void
```

Generic Classes

[attributes]

```cpp
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 ...

costRAINT-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[, ...]
**Generic Interfaces**

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

[type-parameter-constraints-clauses]  

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

**Parameters**

attributes (optional)

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

class-key

class or typename

type-parameter-identifier(s)

Comma-separated identifiers list.

type-parameter-constraints-clauses

Takes the form specified in Constraints

accessibility-modifiers (optional)
Accessibility modifiers (e.g. public, private).

identifier

The interface name.

base-list (optional)

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

interface-body

Declarations of the interface members.

declarators (optional)

Declarations of variables based on this type.

---

**Code**

```cpp
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**

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

**MD-UML**

![UML diagram](image)

---

**Operation - >>**
Clr Data Member option

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

**Code**

```csharp
ref struct MyStruct {
    initonly
    static int staticConst1;
    initonly
    static int staticConst2 = 2;
    static MyStruct() {
        staticConst1 = 1;
    }
};
```

**MD-UML**

MyStruct with «C++CLIStruct» staticConst1 and staticConst2 have «C++CLIAttribute» applied with tagged value `field = initonly` and `Is Static = true`

literal

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

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

<table>
<thead>
<tr>
<th>Code</th>
<th>MD-UML</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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
Microsoft-Specific Native declaration keywords

**__interface**

Grammar:

```
modifier __interface interface-name [interface-definition];
```

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

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

**Code**

```
__interface MyInterface {};
```
**__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**

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

**MD-UML**

```
<<C++ManagedDelegate>>
MyDelegate
+() : int
```

**__event__**

Grammar: __event method-declarator;

**__event __interface** interface-specifier;

**__event** member-declarator;

**Native Events**

**Code**

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

**MD-UML**

```
Source
<<VC++Event>>+MyEvent(i : int): void
```

**Com Events**

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

**Code**

```cpp
__interface IEvents {  
};
class CSource {  
pUBLIC:  
    __event __interface IEvents;  
};
```

**MD-UML**

```
<<VC++Interface>>, IEvents

CSource
<<VC++Event>>: IEvents

__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

<table>
<thead>
<tr>
<th>Code</th>
<th>MD-UML</th>
</tr>
</thead>
<tbody>
<tr>
<td>public __delegate void D();</td>
<td>&lt;&lt;C++ManagedDelegate&gt;&gt;</td>
</tr>
<tr>
<td>public __gc class X {</td>
<td>D</td>
</tr>
<tr>
<td>public:</td>
<td>+() : void</td>
</tr>
<tr>
<td>__event D* E;</td>
<td>&lt;&lt;VC++Event&gt;&gt;+=E : D*&quot;&quot;</td>
</tr>
<tr>
<td>__event void noE();</td>
<td>&lt;&lt;VC++Event&gt;&gt;+=noE() : void</td>
</tr>
<tr>
<td>};</td>
<td>X</td>
</tr>
</tbody>
</table>

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;
};

<table>
<thead>
<tr>
<th>Code</th>
<th>MD-UML</th>
</tr>
</thead>
<tbody>
<tr>
<td>void *vpBuffer;</td>
<td>&lt;&lt;C++Global&gt;&gt;</td>
</tr>
<tr>
<td>void __based(vpBuffer) *vpData;</td>
<td>+vpBuffer : void**</td>
</tr>
<tr>
<td></td>
<td>+vpData : void*&quot;__based(vpBuffer)&quot;</td>
</tr>
</tbody>
</table>
Function calling conventions

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

```c
__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**

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

**MD-UML**

![MD-UML Diagram](image)
__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;
__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.
__fastcall
c
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();
};

__stdcall

class CmyClass2 {
    void __stdcall mymethod();
};
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 __fastcall mymethod2();
};

class CmyClass4 {
    void __thiscall mymethod();
};
**__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**

```c
// 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**

```
<<C++Global>>

+p1: int* __unaligned =
```
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

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

    // char __w64 c;  error, cannot use __w64 on char
}
```
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
ddimport
ddlexport
jitintrinsic
naked

text

int __w64 i;
noalias
noinline
noreturn
nothrow
notvtable
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, notvtable, 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**

class __declspec(dllimport) X {};

**MD-UML**

```
<<C++StorageClass>>
X
[declspecModifier="dllimport"]
```
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**

```c
// __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**

```c
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

[Diagram showing MD-UML related to __forceinline and max function]
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

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

MD-UML

```
<<VC::Base>>
CMyClass

[[-All attributes: coclass(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.

<table>
<thead>
<tr>
<th>Code</th>
<th>MD-UML</th>
</tr>
</thead>
<tbody>
<tr>
<td>public class S&lt;T&gt;</td>
<td></td>
</tr>
<tr>
<td>{</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
</tbody>
</table>

![Diagram of MD-UML and C# code comparison]
Generic Struct

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

<table>
<thead>
<tr>
<th>Code</th>
<th>MD-UML</th>
</tr>
</thead>
<tbody>
<tr>
<td>struct S&lt;T&gt;</td>
<td></td>
</tr>
<tr>
<td>{</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
</tbody>
</table>

![Diagram of Class S with «C#Struct» stereotype]
## Generic Interface

<table>
<thead>
<tr>
<th>Code</th>
<th>MD-UML</th>
</tr>
</thead>
</table>
| ```csharp
interface S<T>
{
}
``` | ![MD-UML Diagram](image) |

### 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.

<table>
<thead>
<tr>
<th>Code</th>
<th>MD-UML</th>
</tr>
</thead>
</table>
| ```csharp
delegate string D<T>();
``` | ![MD-UML Diagram](image) |

### 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
class C<T>
{
}

class D<T>
{
    private C<int> c1;
    private C<string[]> c2;
    private C<T> c3;
    private T c4;
}

The following shows how to create each attribute.

```csharp
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<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.

```csharp
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`.

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

```
private T c4;
```
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.

<table>
<thead>
<tr>
<th>Code</th>
<th>MD-UML</th>
</tr>
</thead>
</table>
| public class S<T>  
{  
    public void f1<T>() {}  
}  
| |

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>.

<table>
<thead>
<tr>
<th>Code</th>
<th>MD-UML</th>
</tr>
</thead>
</table>
| public class S<T>  
{  
    public void f1<T> (T t)  
    {  
    }  
    public void f1<U, V> (T t, U u)  
    {  
    }  
}  
|
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

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;
    }
}
Generic Parent Class

class b<T, U>
{
}
class b1<X> : b<X[], X[,]>
{}
### 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.

**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 ( )

<table>
<thead>
<tr>
<th>Code</th>
<th>MD-UML</th>
</tr>
</thead>
<tbody>
<tr>
<td>public class A { }</td>
<td></td>
</tr>
<tr>
<td>interface IA {}</td>
<td></td>
</tr>
<tr>
<td>public class S&lt;T&gt; where T: A, IA{}</td>
<td></td>
</tr>
</tbody>
</table>

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.
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.
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:

```csharp
// Create a delegate instance
delegate void Del(int x);

// Instantiate the delegate using an anonymous method
```
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:
    attributes_opt class-modifiers_opt partial_opt class identifier type-parameter-list_opt
class-base_opt type-parameter-constraints-clauses_opt class-body_opt

struct-declaration:
    attributes_opt struct-modifiers_opt partial_opt struct identifier type-parameter-list_opt
    struct-interfaces_opt type-parameter-constraints-clauses_opt struct-body_opt

interface-declaration:
    attributes_opt interface-modifiers_opt partial_opt interface identifier type-parameter-list_opt
    interface-base_opt type-parameter-constraints-clauses_opt 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

```csharp
// 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()
  {
  }
}

// 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()
  {
  }
}
```

### MD-UML

<table>
<thead>
<tr>
<th>Class</th>
<th>Members</th>
</tr>
</thead>
<tbody>
<tr>
<td>PartialA</td>
<td>-a: int, -b: int, -actionA(): string, -actionB(): string</td>
</tr>
</tbody>
</table>

All child elements will be merged into one Class element.

<table>
<thead>
<tr>
<th>Class</th>
<th>Members</th>
</tr>
</thead>
<tbody>
<tr>
<td>PartialA</td>
<td>-a: int, -b: int, -actionA(): string, -actionB(): string</td>
</tr>
</tbody>
</table>

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.
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 "partial" tag will have a value of the file name that the class belongs to for each child element in *partial class element*.
The syntax `T?` is shorthand for `System.Nullable<T>`, and the two forms can be used interchangeably.

<table>
<thead>
<tr>
<th>Code</th>
<th>MD-UML</th>
</tr>
</thead>
<tbody>
<tr>
<td>class class1</td>
<td></td>
</tr>
<tr>
<td>{</td>
<td></td>
</tr>
<tr>
<td>int? a = null;</td>
<td></td>
</tr>
<tr>
<td>System.Nullable a = null;</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
</tbody>
</table>

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.
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
static class A
{
    ... 
}

MD-UML
<<C#Class>>
A
{static}

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.
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;
}
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
```
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.

```plaintext
pragma-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.

```csharp
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.

```
unsafe struct A
{
    public fixed int x[5];
}
```

<table>
<thead>
<tr>
<th>Code</th>
<th>MD-UML</th>
</tr>
</thead>
<tbody>
<tr>
<td>unsafe struct A</td>
<td>&lt;&lt;C#Struct&gt;&gt;</td>
</tr>
<tr>
<td>{</td>
<td>A</td>
</tr>
<tr>
<td>public fixed int x[5];</td>
<td>-x : int [5]</td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
</tbody>
</table>
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:

```csharp
public static class Extensions
{
    public static int ToInt32(this string s) {
        return Int32.Parse(s);
    }

    public static T[] Slice<T>(this T[] source, int index, int count) {
        if (index < 0 || count < 0 || source.Length - index < count)
            throw new ArgumentException();
        T[] result = new T[count];
        Array.Copy(source, index, result, 0, count);
    }
}
```
It becomes possible to invoke the extension methods in the static class `Extensions` using instance method syntax:

```csharp
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)
```

### Lambda Expression Conversion

A lambda expression is written as a parameter list, followed by the `=>` token, followed by an expression or a statement block.

```csharp
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:

```csharp
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.
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#.

<table>
<thead>
<tr>
<th>Tag</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>container</td>
<td>String</td>
<td></td>
</tr>
<tr>
<td>fixed</td>
<td>Boolean</td>
<td>Represents the usage of Fixed Size Buffer</td>
</tr>
<tr>
<td></td>
<td></td>
<td>unsafe struct A</td>
</tr>
<tr>
<td></td>
<td></td>
<td>{</td>
</tr>
<tr>
<td></td>
<td></td>
<td>public fixed int x[5];</td>
</tr>
<tr>
<td></td>
<td></td>
<td>}</td>
</tr>
<tr>
<td>readonly</td>
<td>Boolean</td>
<td>Represents the usage of read-only attribute</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Class A</td>
</tr>
<tr>
<td></td>
<td></td>
<td>{</td>
</tr>
<tr>
<td></td>
<td></td>
<td>public readonly int x;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>}</td>
</tr>
<tr>
<td>volatile</td>
<td>Boolean</td>
<td>Represents the usage of volatile attribute</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Class A</td>
</tr>
<tr>
<td></td>
<td></td>
<td>{</td>
</tr>
<tr>
<td></td>
<td></td>
<td>public volatile int x;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>}</td>
</tr>
</tbody>
</table>

### 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.

<table>
<thead>
<tr>
<th>Tag</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Static</td>
<td>boolean</td>
<td>Represents the usage of static class static Class A</td>
</tr>
<tr>
<td></td>
<td></td>
<td>{</td>
</tr>
<tr>
<td></td>
<td></td>
<td>public int x;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>}</td>
</tr>
</tbody>
</table>
**C# Delegate**

«C#Delegate» is a stereotype that indicates that model class represents C# delegate.

<table>
<thead>
<tr>
<th>Name</th>
<th>Meta class</th>
<th>Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>C#Delegate</td>
<td>Class</td>
<td></td>
</tr>
</tbody>
</table>

**C# Element**

«C#Element» is a stereotype that is used to define properties of all element in model.

<table>
<thead>
<tr>
<th>Name</th>
<th>Meta class</th>
<th>Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>C#Element</td>
<td>Element</td>
<td>This stereotype can be applied to all elements such as class, attribute, operation and parameter.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Tag</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
</table>
| 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# Enumeration Literal**

«C# Enumeration Literal» is a stereotype that is used to define enumeration.

<table>
<thead>
<tr>
<th>Name</th>
<th>Meta class</th>
<th>Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>C#EnumerationLiteral</td>
<td>EnumerationLiteral</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Tag</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>C#Initializer</td>
<td>String</td>
<td>Represents enumeration member’s constant value.</td>
</tr>
</tbody>
</table>

**C# Event**

«C# Event» is a stereotype that is used to indicate that operation represents C# event.

<table>
<thead>
<tr>
<th>Name</th>
<th>Meta class</th>
<th>Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>C#Event</td>
<td>Operation</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Tag</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>C#AddAccessor</td>
<td>String</td>
<td>Adds add accessor for event</td>
</tr>
<tr>
<td>C#AddAttributes</td>
<td>String</td>
<td>Defines C# attributes for add accessor.</td>
</tr>
<tr>
<td>C#RemoveAccessor</td>
<td>String</td>
<td>Adds remove accessor for event</td>
</tr>
<tr>
<td>C#RemoveAttributes</td>
<td>String</td>
<td>Defines C# attributes for remove accessor.</td>
</tr>
</tbody>
</table>
**C# Generic**

«C#Generic» is a stereotype that is used to define generic properties.

<table>
<thead>
<tr>
<th>Tag</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>type</td>
<td>generictype</td>
<td>Defines type for generic constraint parameter, class, struct, and new()</td>
</tr>
</tbody>
</table>

**C# Indexer**

«C#Indexer» is a stereotype that is used to indicate that operation represents C# indexer.

<table>
<thead>
<tr>
<th>Tag</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>C#GetAccessor</td>
<td>String</td>
<td>Adds get accessor for indexer.</td>
</tr>
<tr>
<td>C#GetAttributes</td>
<td>String</td>
<td>Defines C# attributes for get accessor.</td>
</tr>
<tr>
<td>C#SetAccessor</td>
<td>String</td>
<td>Adds set accessor for indexer.</td>
</tr>
<tr>
<td>C#SetAttributes</td>
<td>String</td>
<td>Defines C# attributes for set accessor.</td>
</tr>
</tbody>
</table>

**C# Language Property**

«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.

<table>
<thead>
<tr>
<th>Tag</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>internal</td>
<td>boolean</td>
<td></td>
</tr>
<tr>
<td>new</td>
<td>boolean</td>
<td></td>
</tr>
<tr>
<td>unsafe</td>
<td>boolean</td>
<td>Represents the usage of unsafe element</td>
</tr>
<tr>
<td></td>
<td></td>
<td>unsafe struct A</td>
</tr>
<tr>
<td></td>
<td></td>
<td>{</td>
</tr>
<tr>
<td></td>
<td></td>
<td>public fixed int x[5];</td>
</tr>
<tr>
<td></td>
<td></td>
<td>}</td>
</tr>
</tbody>
</table>

**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#.

<table>
<thead>
<tr>
<th>Tag</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>C#Operation</td>
<td>Operation</td>
<td></td>
</tr>
</tbody>
</table>
### Tag Definitions

| Tag                          | Type          | Description                                                                 
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>conversion type</td>
<td>String</td>
<td>Represent the usage of extern operation</td>
</tr>
<tr>
<td>extern</td>
<td>boolean</td>
<td>Class A</td>
</tr>
<tr>
<td></td>
<td></td>
<td>{</td>
</tr>
<tr>
<td></td>
<td></td>
<td>public extern int x();</td>
</tr>
<tr>
<td></td>
<td></td>
<td>}</td>
</tr>
<tr>
<td>initialization</td>
<td>String</td>
<td></td>
</tr>
<tr>
<td>override</td>
<td>boolean</td>
<td></td>
</tr>
<tr>
<td>partial</td>
<td>String</td>
<td></td>
</tr>
<tr>
<td>virtual</td>
<td>boolean</td>
<td></td>
</tr>
<tr>
<td>C#ExplicitInterface</td>
<td>Classifier</td>
<td>Defines C# explicit interface for explicit interface member implementation.</td>
</tr>
</tbody>
</table>

### C#Operator

«C#Operator» is a stereotype that is used to indicate that the operation represents C# operator.

<table>
<thead>
<tr>
<th>Name</th>
<th>Meta class</th>
<th>Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>C#Operator</td>
<td>Operation</td>
<td></td>
</tr>
</tbody>
</table>

### C#Parameter

«C#Parameter» is a stereotype that is used to indicate that the element represents C#Parameter.

<table>
<thead>
<tr>
<th>Name</th>
<th>Meta class</th>
<th>Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>C#Parameter</td>
<td>Element</td>
<td></td>
</tr>
</tbody>
</table>

### C#Params

«C#Params» is a stereotype that is used to indicate parameter arrays.

<table>
<thead>
<tr>
<th>Name</th>
<th>Meta class</th>
<th>Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>C#Params</td>
<td>Parameter</td>
<td></td>
</tr>
</tbody>
</table>

### C#Property

«C#Property» is a stereotype that is used to indicate that the operation represents C# property.

<table>
<thead>
<tr>
<th>Name</th>
<th>Meta class</th>
<th>Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>C#Property</td>
<td>Operation</td>
<td></td>
</tr>
</tbody>
</table>

### Additional Tags

<table>
<thead>
<tr>
<th>Tag</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>extend</td>
<td>String</td>
<td></td>
</tr>
<tr>
<td>params</td>
<td>String</td>
<td></td>
</tr>
</tbody>
</table>

### C#GetAccessor

`C#GetAccessor` adds `get` accessor for indexer.

### C#GetAttributes

`C#GetAttributes` defines C# attributes for `get` accessor.

### C#SetAccessor

`C#SetAccessor` adds `set` accessor for indexer.

### C#SetAttributes

`C#SetAttributes` defines C# attributes for `set` accessor.
C#Struct

«C#Struct» is a stereotype that is used to indicate that the model class represents C# structure.

<table>
<thead>
<tr>
<th>Name</th>
<th>Meta class</th>
<th>Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>C#Struct</td>
<td>Class</td>
<td></td>
</tr>
</tbody>
</table>

C#Using

«C#Using» is a stereotype that is used to indicate that the usage dependency is C# Using directive.

<table>
<thead>
<tr>
<th>Name</th>
<th>Meta class</th>
<th>Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>C#Using</td>
<td>Association, Realization, Usage</td>
<td>The clients of dependency are Component, and Namespace.</td>
</tr>
</tbody>
</table>

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.

![Data Type Diagram]

Figure 133 -- 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

- Open local project
  - Open project
  - Project is opened completely
    - (version <= 11.6) or (version > 11.6)
      - (version <= 11.6)
        - Show question dialog about this project needs to be translated and do you want to backup?
          - [Click No]
          - [Click Yes]
            - Show backup dialog
              - [Specify name and Click Ok] or [Click Cancel]
                - Backup project
                  - Show warning dialog about MD will continue to translate
                    - Translate
                      - Project is translated completely

Figure 134 -- Open local project Activity Diagram
Open teamwork project

Figure 135 -- Open teamwork project Activity Diagram
Import MagicDraw project

Figure 136 -- Import MagicDraw project Activity Diagram
Use module

Pre-condition: project opened

Right click on C# code engineering set and choose generate

[not use old C# project]

[use old C# project]

Show error message about this project use old C# project

Generate nothing

Show generation dialog box normally

Figure 137 -- Use module Activity Diagram
Update C++ Language Properties and Profiles

Pre-condition: project opened, version of project equal or more than 12 and needs to be translated (use C# language properties)

Right click on "Code engineering set" in the containment tree and select "Update C# Language Properties and Profiles" menu

Show question dialog about The project is going to be translated. Do you want to continue?

[Click Yes] [Click No]

[local project] [teamwork project]

[has permission to edit] [not has permission to edit]

[can lock all elements] [cannot lock all elements]

Show error dialog about cannot lock all elements and cannot translate project and this must be done manually by invoking menu item

Show error dialog about cannot translate project and C# code engineering set may not work correctly

Show the list of elements that need to be unlock

Translate

Project is translated completely

Project is not translated

Figure 138 -- 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)
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

<table>
<thead>
<tr>
<th>Old Value</th>
<th>Translation</th>
</tr>
</thead>
<tbody>
<tr>
<td>New</td>
<td>Mapped to tag “new” of «C#LanguageProperty» (The usage for this is to apply «C#Attribute» and set value to tag “new”)</td>
</tr>
<tr>
<td>Internal</td>
<td>Mapped to tag “internal” of «C#LanguageProperty» (The usage for this is to apply «C#Attribute» and set value to tag “internal”)</td>
</tr>
<tr>
<td>Unsafe</td>
<td>Mapped to tag “unsafe” of «C#LanguageProperty» (The usage for this is to apply «C#Attribute» and set value to tag “unsafe”)</td>
</tr>
<tr>
<td>Readonly</td>
<td>Mapped to “readonly” of «C#Attribute»</td>
</tr>
<tr>
<td>Volatile</td>
<td>Mapped to “volatile” of «C#Attribute»</td>
</tr>
<tr>
<td>Container</td>
<td>Mapped to “container” of «C#Attribute»</td>
</tr>
</tbody>
</table>
### Operation

<table>
<thead>
<tr>
<th>Old Value</th>
<th>Translation</th>
</tr>
</thead>
<tbody>
<tr>
<td>New</td>
<td>Mapped to tag “new” of «C#LanguageProperty» (The usage for this is to apply «C#Operation» and set value to tag “new”)</td>
</tr>
<tr>
<td>Internal</td>
<td>Mapped to tag “internal” of «C#LanguageProperty» (The usage for this is to apply «C#Operation» and set value to tag “internal”)</td>
</tr>
<tr>
<td>Unsafe</td>
<td>Mapped to tag “unsafe” of «C#LanguageProperty» (The usage for this is to apply «C#Operation» and set value to tag “unsafe”)</td>
</tr>
<tr>
<td>Override</td>
<td>Mapped to tag “override” of «C#Operation»</td>
</tr>
<tr>
<td>Virtual</td>
<td>Mapped to tag “virtual” of «C#Operation»</td>
</tr>
<tr>
<td>Extern</td>
<td>Mapped to tag “virtual” of «C#Operation»</td>
</tr>
<tr>
<td>Initialization</td>
<td>Mapped to tag “initialization” of «C#Operation»</td>
</tr>
<tr>
<td>Conversion type</td>
<td>Mapped to tag “conversion type” of «C#Operation»</td>
</tr>
</tbody>
</table>
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.

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.
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.
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

```csharp
using System;
using System.Collections.Generic;
using System.Text;
namespace n0
{
class using1 {}
}
```

MD-UML

Open usage dependency specification
apply «C#Using» stereotype.
For using namespace, leave the name empty.
**Open usage dependency specification**

**Apply «C#Using» stereotype.**

**For using alias, enter the alias name.**

<table>
<thead>
<tr>
<th>Code</th>
<th>MD-UML</th>
</tr>
</thead>
<tbody>
<tr>
<td>namespace n0 { using n1 = n1.n1; using n2 = n1.n2; using n3 = n1.n3; class using1 : n2 { } } namespace n1 { namespace n2 { class using1 { } } } class using1 { }</td>
<td>namespace n0 { using n1::using1&lt;int&gt;; using n2 = n1::using1&lt;boo&gt;; using n3 = n1.n2; class using1 : n2 { } } namespace n1 { namespace n2 { class using1&lt;T&gt; { } } } class using1&lt;T&gt; { }</td>
</tr>
</tbody>
</table>

**Image 1:**
- C# code snippet demonstrating the use of namespaces and classes.

**Image 2:**
- UML diagram showing the usage dependency specification.

**Image 3:**
- Usage - n1 dialog box with properties and relations.
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.

<table>
<thead>
<tr>
<th>Code</th>
<th>MD-UML</th>
</tr>
</thead>
</table>
| ```csharp
//partial1.cs
namespace n1 {
    using LL = System.Text.Encoder;
    partial class using1 {
        LL a;
    }
}
``` | ![MD-UML diagram of partial1.cs](image1)

![Usage - Alias1](image2)

```csharp
//partial2.cs
namespace n1 {
    using LL = System.Text.Decoder;
    partial class using1 {
        LL b;
    }
}
``` | ![MD-UML diagram of partial2.cs](image3)
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;

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

<table>
<thead>
<tr>
<th>Code used for reverse engineer</th>
<th>Code after generation</th>
</tr>
</thead>
<tbody>
<tr>
<td>//Case #1</td>
<td>public partial class PartialA{</td>
</tr>
<tr>
<td>//The partial class is written into one class file.</td>
<td>int a;</td>
</tr>
<tr>
<td></td>
<td>int b;</td>
</tr>
<tr>
<td>public partial class PartialA{</td>
<td>string actionA( )</td>
</tr>
<tr>
<td>int a;</td>
<td>(return ;)</td>
</tr>
<tr>
<td>string actionA()</td>
<td>string actionB( )</td>
</tr>
<tr>
<td>{}</td>
<td>(return ;)</td>
</tr>
<tr>
<td>public partial class PartialA</td>
<td>}</td>
</tr>
<tr>
<td>int b;</td>
<td>}</td>
</tr>
<tr>
<td>string actionB()</td>
<td>}</td>
</tr>
</tbody>
</table>

//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() |
{} |
}

Figure 143 -- Conflict Message Dialog
### Translation Constraints

#### Property Translation

As you may know that after version 12.1 MD C# will translate C#Property from operation to attribute. After the translation is finished, the message window will show the following message:

> Orphaned proxy found in module "C#.Profile.xml". Use search functionality with option "Orphaned proxies only" to locate them.

This message occurred because there is no Operation as a metaclass of C#Property (changing of profile), but this will not affect the functionality of reverse engineering and translation.

After the project has been saved and reopened, the message will not be appear any more.

<table>
<thead>
<tr>
<th>Code used for reverse engineer</th>
<th>Code after generation</th>
</tr>
</thead>
</table>
| ```
//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;
  }
}
```

```
NOTE: This functionality is available in Enterprise edition only.


Differences between UML Profile for CORBA specification and mapping in MagicDraw is listed below.

- Constraints defined in UML Profile for CORBA specification are not checked in MagicDraw.
- Stereotype CORBAAnonymousFixed is introduced. It is used to represent anonymous fixed types. Fixed types without names are mapped to inner classes with stereotype CORBAAnonymousFixed. These classes are bound to CORBA::fixed classes.

```idl
define struct baz
    fixed <8, 4> high_scale;
    fixed <8, 2> low_scale;
end define
```

This code is mapped to the following diagram:

MagicDraw presents a CORBA IDL diagram, which simplifies the creation of standard CORB IDL elements. The following elements are available in the CORBA IDL diagram:
CORBA Interface Implementation

You can select either the UML Interface or the UML Class as a base element for CORBA Interface. The default element as a base element for CORBA Interfaces is as follows:

- **UML Class.** A class as a base element for CORBA Interfaces is in projects created on earlier than MagicDraw version 17.0.1. The CORBA Interfaces are modeled as classes in these projects.
UML Interface. An interface as a base element for CORBA Interfaces is in projects created on MagicDraw version 17.0.1 and later. The CORBA Interfaces are modeled as interfaces in these projects.

**IMPORTANT!** The generalization relationship can be used only between CORBA Interfaces based on the same element. That is, you can use the generalization relationship between CORBA Interfaces based on a class or between CORBA Interfaces based on an interface but not between CORBA Interface based on a class and CORBA Interface based on an interface.

To set a base element for CORBA Interfaces

1. On the Options menu, click Project. The Project Options dialog opens.
2. In the Tab tree, select General project options.
3. In the General project options list, click the CORBA Interfaces implemented as value cell. The list of available values opens.
4. Do one of the following:
   - Select UML Class, to set a class as the base element for the CORBA Interface.
   - Select UML Interface, to set an interface as the base element for the CORBA Interface.
5. Click OK when you are done.

To change a base element for CORBA Interfaces

1. Select the CORBA Interface element in the Model Browser or its symbol on the diagram pane.
2. On the shortcut menu of the selected CORBA Interface, point to **Refactor > Convert To**, and then do the following:
   - Click **Class**, if an interface is the current base element for CORBA Interface.
   - Click **Interface**, if a class is the current base element for CORBA Interface.

**IMPORTANT!** Convert selected CORBA Interface only to the class or interface element. Conversion to other elements changes the CORBA Interface element to the selected element but not the element CORBA Interface is based on.

For more information about refactoring, see “Refactoring” in “MagicDraw UserManual.pdf”.
WSDL

NOTE: This functionality is available in Enterprise edition only.

Reference http://www.w3.org/TR/2001/NOTE-wsdl-20010315

WSDL is an XML format for describing network services as a set of endpoints operating on messages containing either document-oriented or procedure-oriented information. The operations and messages are described abstractly, and then bound to a concrete network protocol and message format to define an endpoint. Related concrete endpoints are combined into abstract endpoints (services).

MagicDraw UML supports WSDL code engineering: code generation, reverse and syntax checking. WSDL diagram is dedicated for WSDL modeling.

WSDL Profile and XML Schema Profile are used in WSDL code engineering.

WSDL Services are defined using six major elements:

- **types**, which provides data type definitions used to describe the messages exchanged.
- **message**, which represents an abstract definition of the data being transmitted. A message consists of logical parts, each of which is associated with a definition within some type system.
- **portType**, which is a set of abstract operations. Each operation refers to an input message and output messages.
- **binding**, which specifies concrete protocol and data format specifications for the operations and messages defined by a particular portType.
- **port**, which specifies an address for a binding, thus defining a single communication endpoint.
- **service**, which is used to aggregate a set of related ports.
WSDL Mapping to UML elements

Defined stereotypes

<table>
<thead>
<tr>
<th>Element</th>
<th>Stereotype name</th>
<th>Applies to</th>
<th>Defined TagDefinitions</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Definition</td>
<td>WSDL:definitions</td>
<td>Component</td>
<td>extension - string name - string targetNamespace - string</td>
<td></td>
</tr>
<tr>
<td>Message</td>
<td>WSDL:message</td>
<td>Class</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Port Type</td>
<td>WSDL:porttype</td>
<td>Interface</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Binding</td>
<td>WSDL:binding</td>
<td>Class</td>
<td>extension - string</td>
<td></td>
</tr>
<tr>
<td>Port</td>
<td>WSDL:port</td>
<td>Instance</td>
<td>specification</td>
<td>extension - string</td>
</tr>
<tr>
<td>Service</td>
<td>WSDL:service</td>
<td>Component</td>
<td>extension - string</td>
<td></td>
</tr>
<tr>
<td>Type</td>
<td>WSDL:types</td>
<td>Component</td>
<td>extension - string</td>
<td></td>
</tr>
<tr>
<td></td>
<td>WSDL:import</td>
<td>ElementImport,</td>
<td></td>
<td></td>
</tr>
<tr>
<td>xmlns</td>
<td></td>
<td>PackageImport</td>
<td></td>
<td>From the XML Schema Profile</td>
</tr>
<tr>
<td>XSDnamespace</td>
<td></td>
<td>Package</td>
<td></td>
<td>From the XML Schema Profile</td>
</tr>
<tr>
<td>WSDL:response</td>
<td></td>
<td>Parameter</td>
<td>extension - string</td>
<td></td>
</tr>
<tr>
<td>WSDL:operation</td>
<td></td>
<td>Operation</td>
<td>extension - string</td>
<td></td>
</tr>
<tr>
<td>WSDL:part</td>
<td></td>
<td>Property</td>
<td>typing Attribute - string</td>
<td></td>
</tr>
<tr>
<td>WSDL:fault</td>
<td></td>
<td>Parameter</td>
<td>extension - string</td>
<td></td>
</tr>
<tr>
<td>WSDL:request</td>
<td></td>
<td>Parameter</td>
<td>extension - string</td>
<td></td>
</tr>
</tbody>
</table>

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/wsd1"/>
Reversed UML model example:

```
<definitions xmlns="http://schemas.xmlsoap.org/wsdl/">
  <import location="http://www.bbb.net/wsdl" namespace="http://www.aaa.org/wsdl"/>
</definitions>
```

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:

```
<definitions xmlns="http://schemas.xmlsoap.org/wsdl/">
  <import location="http://www.bbb.net/wsdl" namespace="http://www.aaa.org/wsdl"/>
</definitions>
```

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"/>
```
WSDL Mapping to UML elements

xmlns="http://schemas.xmlsoap.org/wsd/"

<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>

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:

```xml
<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 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:

```xml
<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"/>
            <soap:header message="tns:SubscribeToQuotes" part="subscribeheader" use="literal"/>
        </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>
```
WSDL Mapping to UML elements

Reversed UML model example:

```
<all>
  <element name="tickerSymbol" type="string"/>
</all>
</complexType>
</element>
<element name="SubscriptionHeader" type="uriReference"/>
</types>
</definitions>
```

**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"/>
    </operation>
  </portType>

```

<output message="tns:GetTradePriceOutput"/>
</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 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 its 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:

```xml
<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">
```

```xml"
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:

```xml
<definitions name="HelloService"
    targetNamespace="http://www.echerami.com/wsd1/HelloService.wsdl"
    xmlns="http://schemas.xmlsoap.org/wsdl/
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/
    xmlns:tns="http://www.echerami.com/wsd1/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">
            <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: