C/C++ Code Engineering Guide for Sparx Enterprise Architect™ UML

Generation, Reverse Engineering and Synchronizing of C / C++ Code with EA UML Models

Version: 0.01
Authors: Günther Makulik
Last Updated: 01/11/2013

Table of Contents

1 Introduction

This guides purpose is to help users solving the C/C++ language specific tasks for reverse engineering, code generation and synchronisation of Enterprise Architect class and behavioral models.

There already exists a good document (see [1]) published by Sparx Systems that explains EA's code engineering facilities in more general form. But there are still a number of points to solve for practical use with C and C++ code in particular.

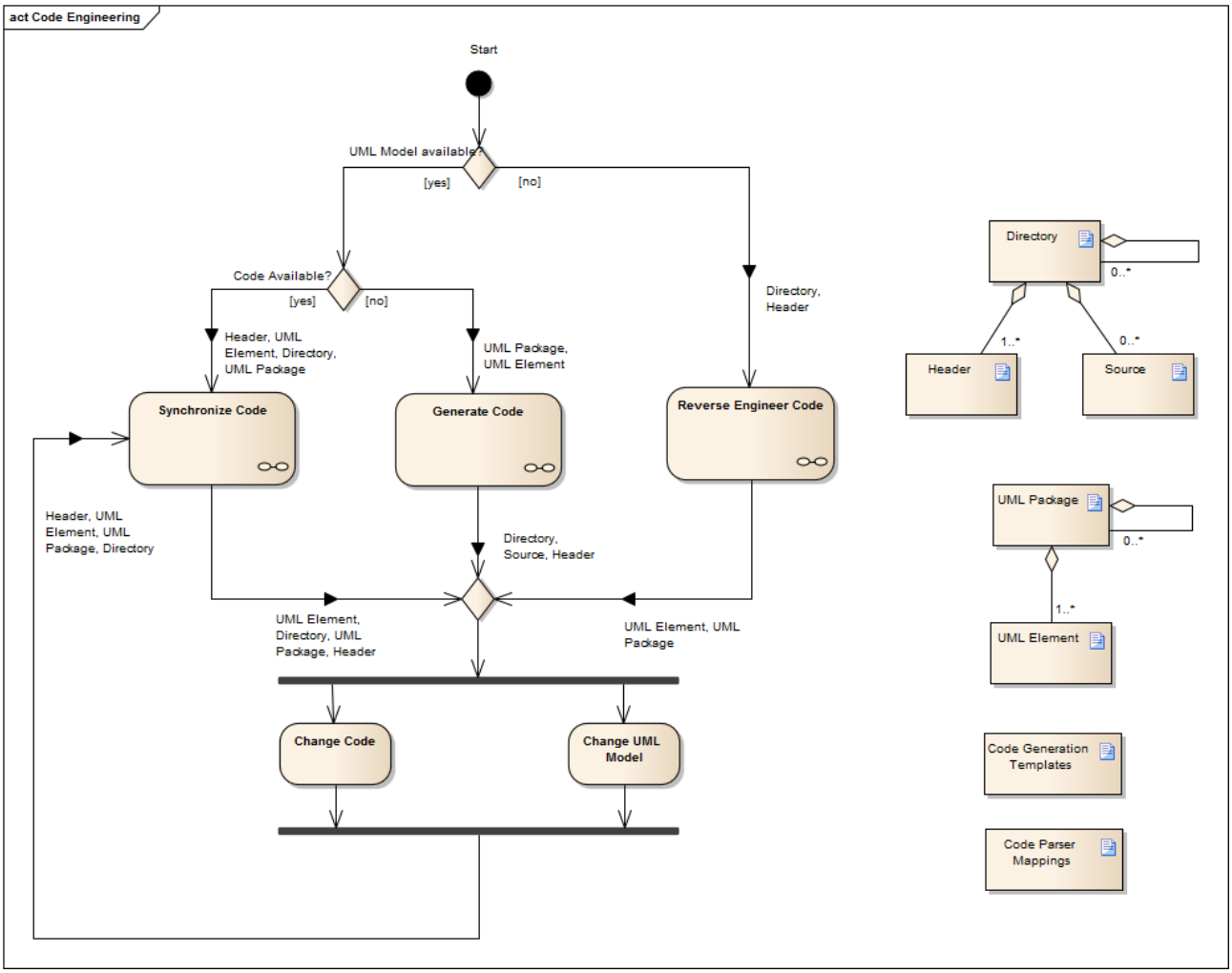
Also EA's extensive Help documentation is sometimes hard to navigate and leaky about details, so we'll try to point to the right links here.

1.1 Scope

The scope of this document covers the current EA version 9.3.

Some of the newer C++11 and C2011 standard features may not be considered for solutions in preference of the C++03 and C99 standards.

2 The EA Code Generation/Synchronisation Cycle



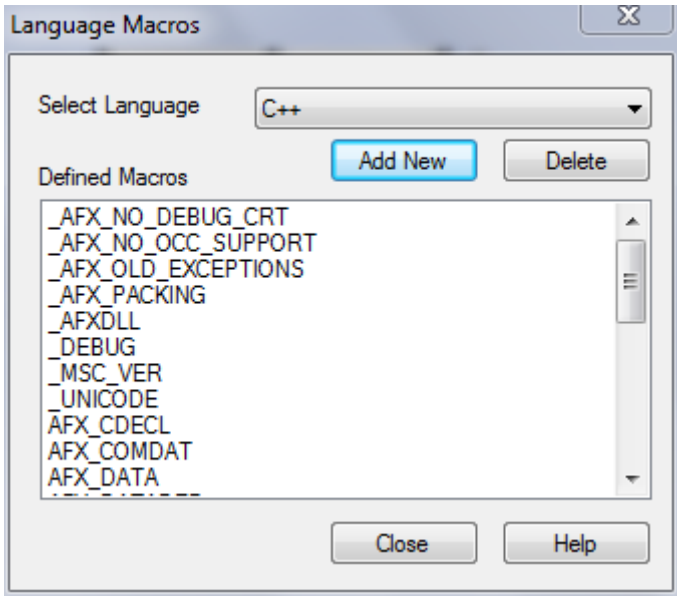
2.1 EA Code Generation Templates

2.2 EA Code Code Parsing Grammar Definitions[i]

3 Common C & C++ Techniques

3.1 Ignoring tokens when code is parsed

The only opportunity to ignore certain tokens like preprocessor defines and macros and compiler specific keywords and attributes is to define Language Macros using the 'Settings->Preprocessor Macros ...' dialog:



Despite the 'Select Language' drop down list offers only C++, the settings made here are also considered for C code.

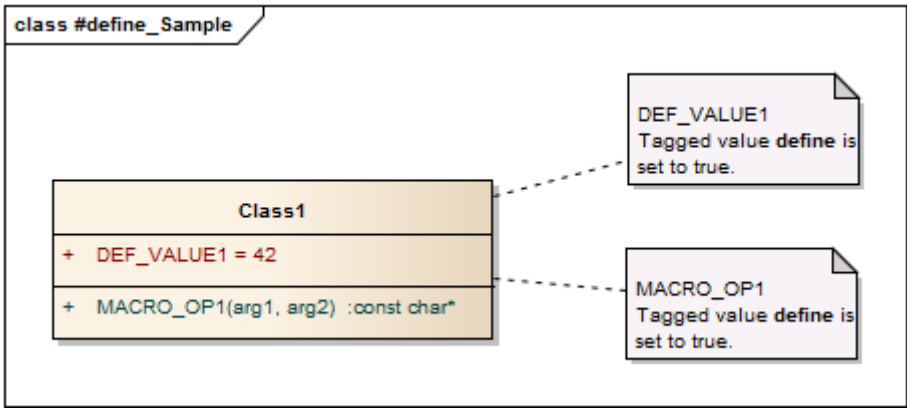
You can add new values of the form

- <define_tag>
- <macro_tag>()

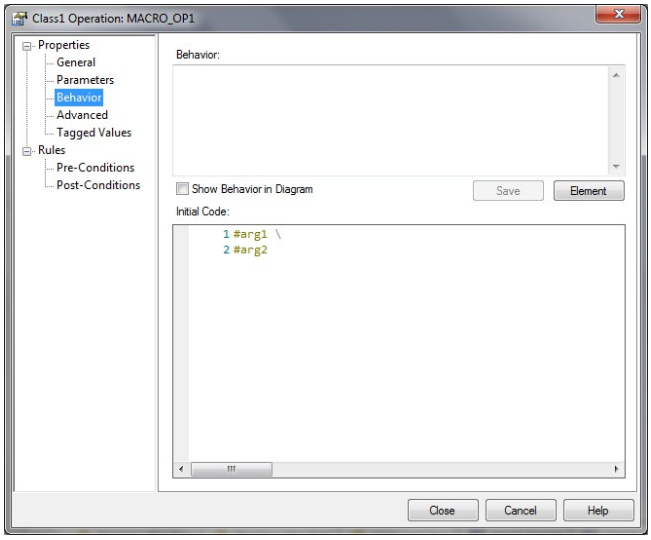
The latter will ignore anything between the parenthesis that is seen in the parsed code.

3.2 Preprocessor #define statements

Preprocessor `#define` statements for values or macros can be achieved modeling public operations or attributes and giving them a tagged value '**define**' set to 'true':



Macro code is defined as initial code for the operation. Note that you'll need to put line continuation ('\') characters before a following line as you would do in a source code file:



3.3 Include guards

For header files generated from the code EA's standard code generation templates automatically add include guards making use of the corresponding class elements' GUID.

These can be changed in the 'File' code generation template of the C or C++ Language:

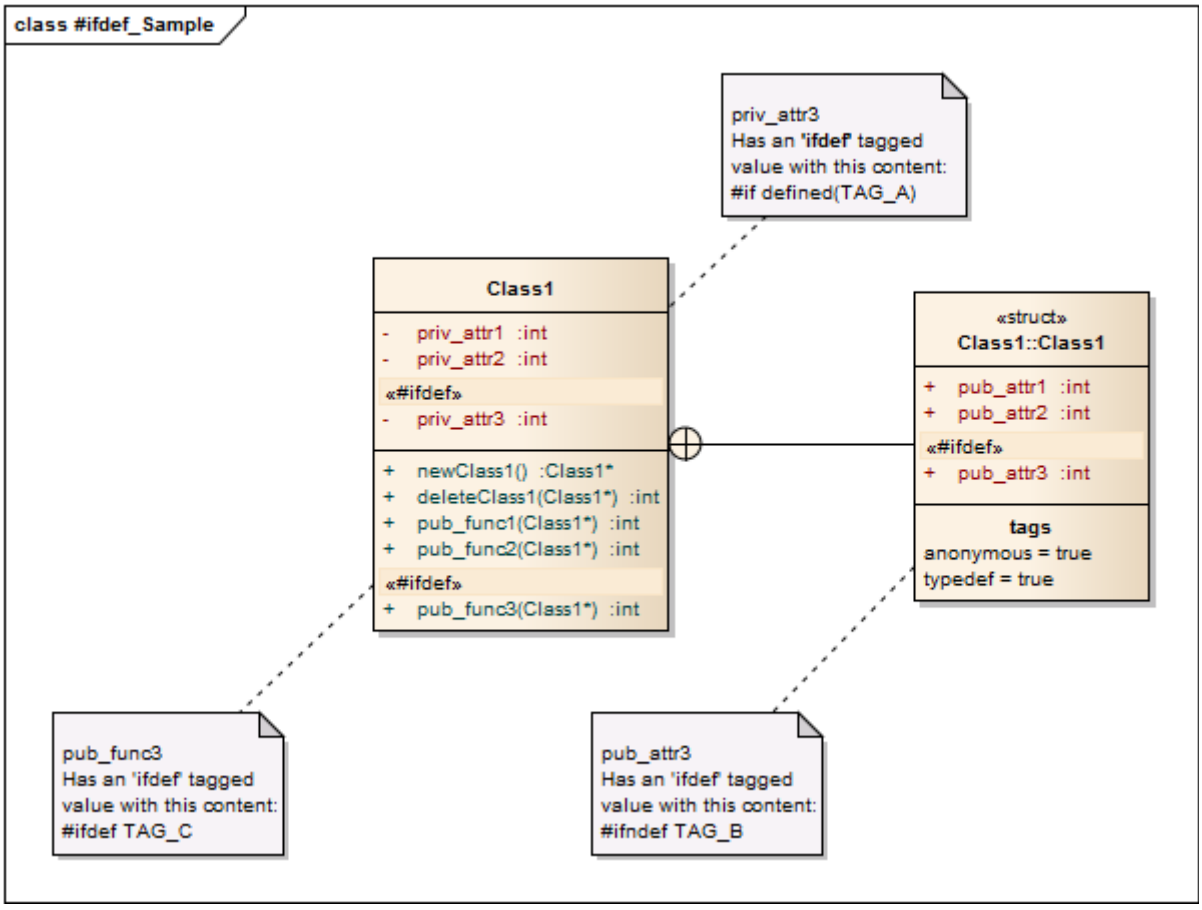
```
$guid = "EA_" + %TRIM(eaGUID,"{}")%
$guid = %REPLACE($guid,"-","_")%
$guid += "__INCLUDED_"

#if !defined($guid)
#define $guid
\n
...
\n
#endif /*!defined($guid)*/\n
```

If you not want to generate include guard statements for certain class elements you may provide a stereotype for these and a sterotyped override of the 'File' template.

3.4 Preprocessor #ifdef statements

If you want to have certain features of your class elements inside conditional preprocessor statements like `#ifdef`, `#if defined()` or `#ifndef`, you need to extend the C/C++ standard code generation templates.

For this solution I have chosen to mark the features in question with the stereotype «#ifdef». Additionally the particular condition statement is provided in a tagged value named 'ifdef':



The next step is to provide stereotyped overrides for the 'Attribute', 'Operation' and 'Operation Impl' code generation templates with the «#ifdef» stereotype.

'Attribute' code generation template (C) overriden for stereotype «#ifdef»

```
$ifdef_cond = %attTag:"ifdef"%
%if $ifdef_cond != ""%
\n$ifdef_cond
%endIf%
%AttributeNotes%
%AttributeDeclaration%
```

```
$ifdef_cond = %attTag:"ifdef"%
%if $ifdef_cond != ""%
#endif\n
%endIf%
```

'Operation' code generation template (C) overriden for stereotype «#ifdef»

```
$ifdef_cond = %opTag:"ifdef"%
%if $ifdef_cond != ""%
\n$ifdef_cond\n
%endIf%

%OperationNotes%
%PI=""%
%OperationDeclaration%

%if opStatic=="T"%
%if $ifdef_cond != ""%
\n#endif\n
%endIf%
%endTemplate%

$bodyLoc = %opTag:"bodyLocation"%

%if opTag:"inline" == "true" or $bodyLoc == "header" or $bodyLoc == "classDec"%
%PI="\n"%
%OperationBody%
%if $ifdef_cond != ""%
\n#endif\n
%endIf%
%endTemplate%

%if $ifdef_cond != ""%
\n#endif\n
%endIf%
```

'Operation Impl' code generation template (C) overriden for stereotype «#ifdef»

```
%if opTag:"define" == "true"%
%endTemplate%

$ifdef_cond = %opTag:"ifdef"%
%if $ifdef_cond != ""%
$ifdef_cond
%endIf%

$bodyLoc = %opTag:"bodyLocation"%
%if opTag:"inline" == "true" or $bodyLoc == "header" or $bodyLoc == "classDec"%
%endTemplate%
%PI="\n"%
%OperationNotesImpl%
%OperationDeclarationImpl%
%OperationBodyImpl%

$ifdef_cond = %opTag:"ifdef"%
%if $ifdef_cond != ""%
#endif
%endIf%
```

Note:

You should not try to do this on the level of the 'xxx Declaration' / 'xxx Declaration Impl' code generation templates. This will be problematic with synchronizing the generated code in later steps (see also: Help section 'Standard UML Models->Define a Modeling Language->Code Template Framework->Synchronize Code').

During reverse engineering of existing code all conditional preprocessor statments will be ignored by EA's C/C++ code parsers.

Thus you have to mark features that appear within such statements manually in the reverse engineered UML model with the «#ifdef» sterotype and appropriate 'ifdef' tagged values.

3.5 Additional #include statements

Usually EA's standard code generation templates manage to render `#include` statements with the 'Import Section' and the 'Import Section Impl' templates.

'Import Section' code generation template (C/C++)

```
%fileImports%
```

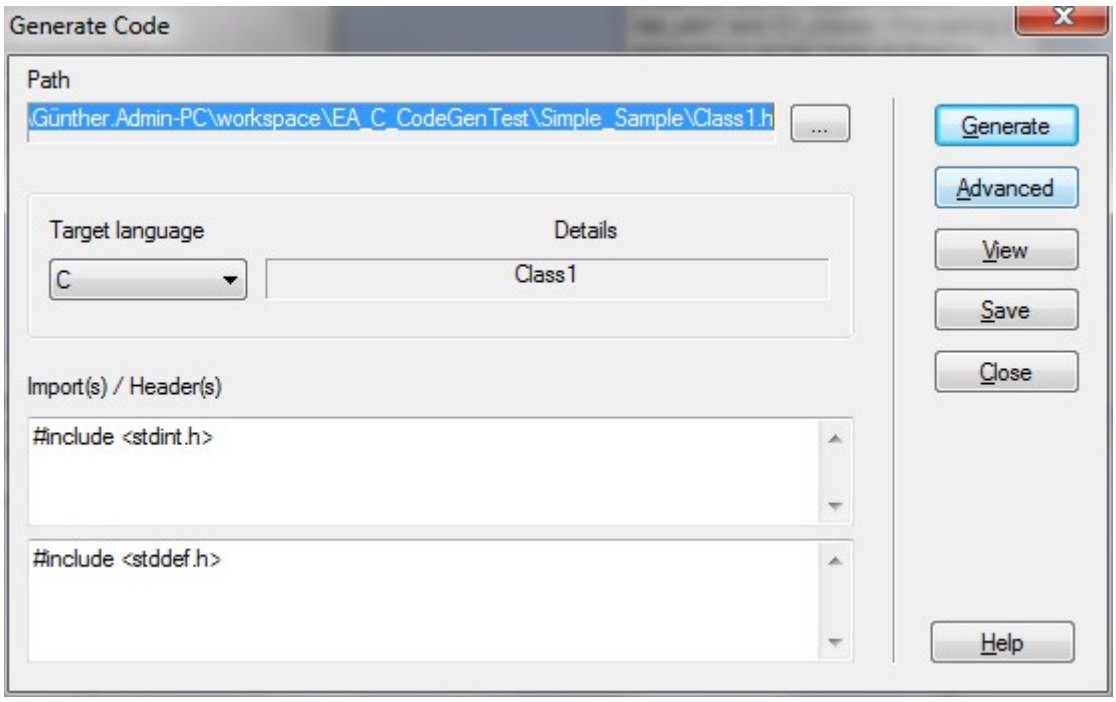
'Import Section Impl' code generation template (C/C++)

```
%fileHeaders%
```

The CTF macros used here are described as follows in the EA help:

fileHeaders	Code Gen dialog: **Headers**.
fileImports	Code Gen dialog: **Imports**. For supported languages this also includes dependencies derived from associations.

The dialog fields referred here can be set on class level choosing the 'Code Generation (F11)' command:

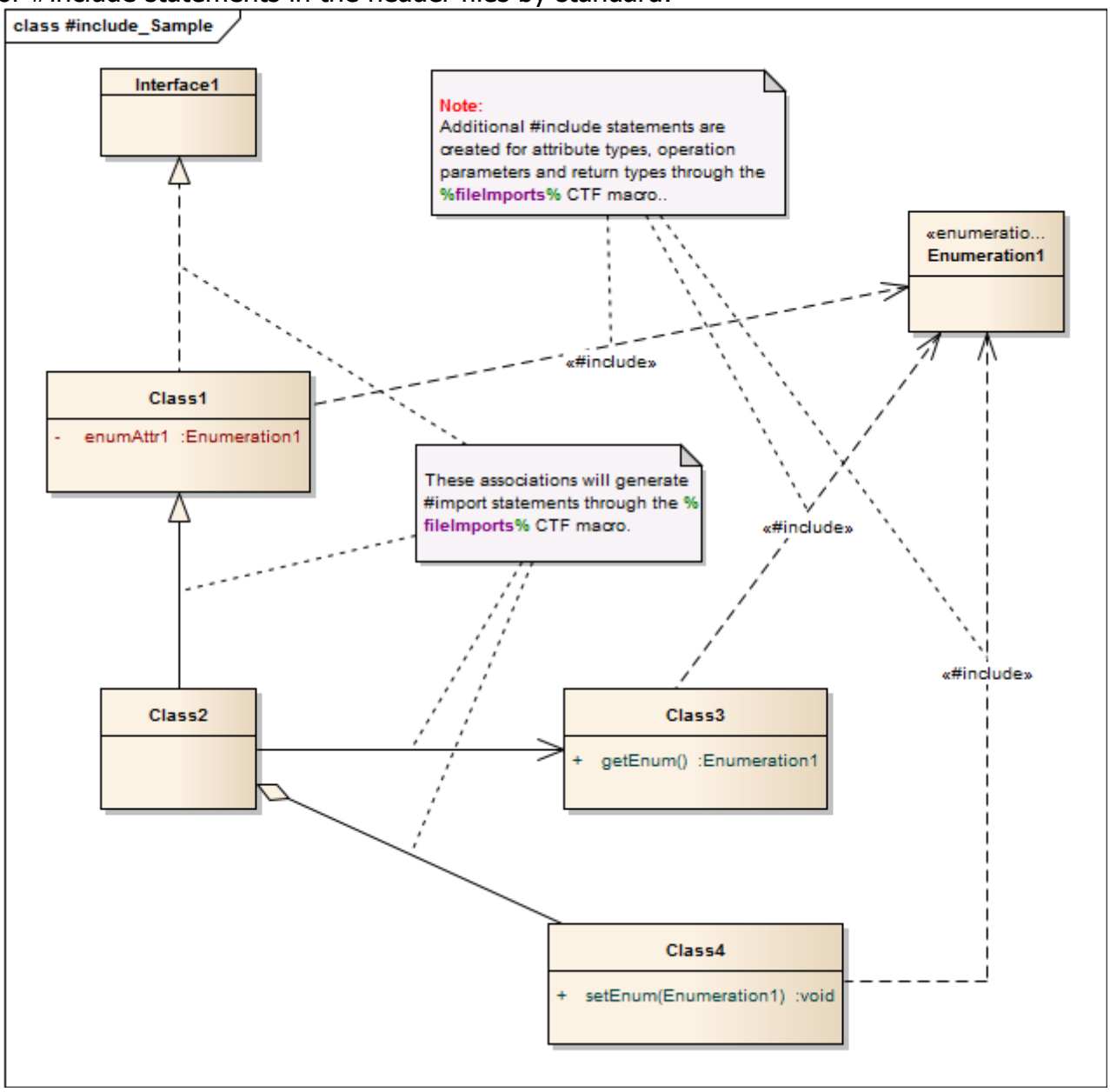


The association dependencies mentioned for the `%fileImports%` CTF macro are class level connections of the following types:

- (Directed) Association Links
- Generalization Links
- Realization Links

Additionally any other classifiers used in the UML model as attribute types, operation parameters and return values will cause `%fileImports%` to generate #include statements.

The following class diagram depicts, which associations and features will cause generation of #include statements in the header files by standard:



3.5.1 Custom #include statements

The basic approach for handling generation of additional custom `#include` statements is to provide implementations for the CTF 'Import' and 'Import Impl' base templates. Usually these are empty for the C/C++ code generation templates.
To enable them you'll need to change the 'File' and 'File Impl' templates in the following way (focus on the highlited code):

Enhanced 'File' code generation template

```
%synchNewClassNotesSpace="\n"%
%synchNewOperationNotesSpace="\n"%
%synchNewOperationBodySpace="\n"%
%synchNewAttributeNotesSpace="\n"%
```

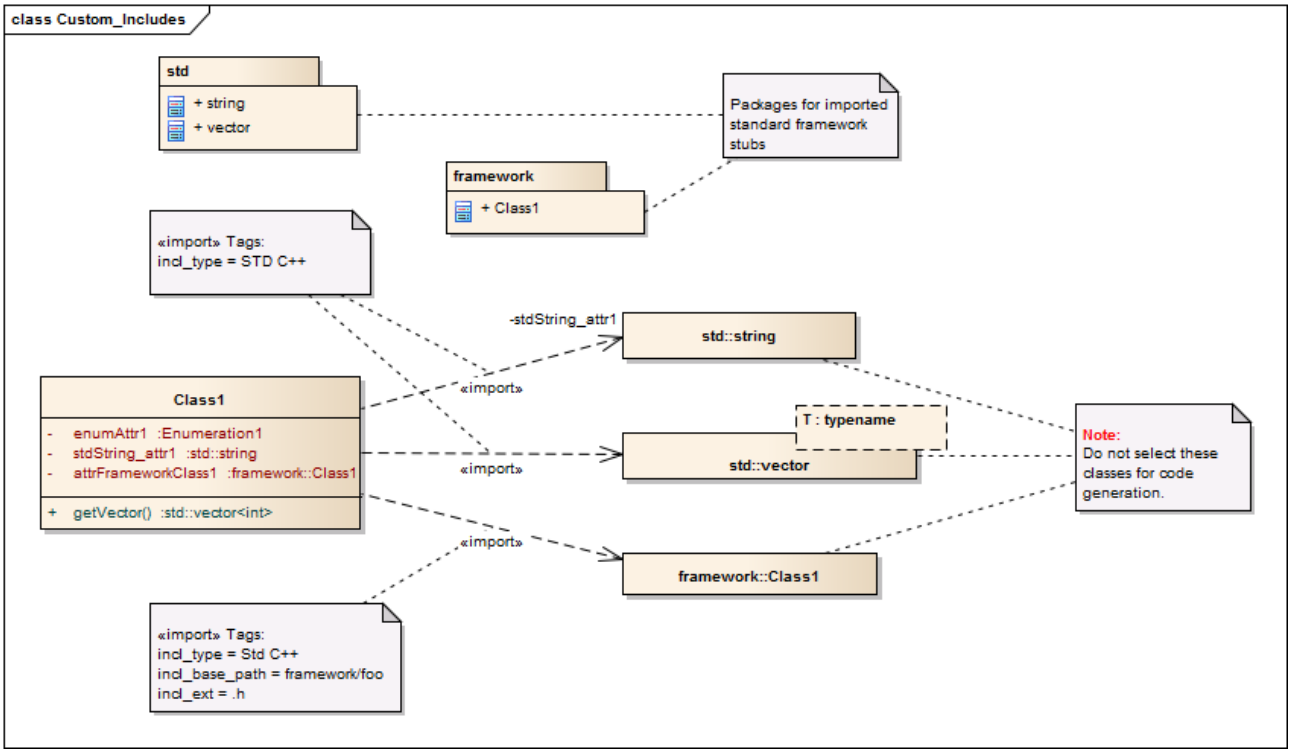
```
////////////////////////////////////////////////////////////
//  %fileName%
//  Implementation of the %elemType% %className%
//  Created on:      %eaDateTime%
%if classAuthor != ""%
//  Original author: %classAuthor%
%endIf%
////////////////////////////////////////////////////////////\n

$COMMENT="WARNING: DO NOT MODIFY THIS TEMPLATE BELOW THIS POINT"
$guid = "EA_" + %TRIM(eaGUID,"{}")%
$guid = %REPLACE($guid,"-","_")%
$guid += "__INCLUDED_"
#if !defined($guid)
#define $guid\n
%ImportSection%
%Import%
%list="Namespace" @separator="\n\n"%
#endif // !defined($guid)\n
```

Enhanced 'File Impl' code generation template

```
////////////////////////////////////////////////////////////
//  %fileNameImpl%
//  Implementation of the %elemType% %className%
//  Created on:      %eaDateTime%
%if classAuthor != ""%
//  Original author: %classAuthor%
%endIf%
////////////////////////////////////////////////////////////\n

$COMMENT="WARNING: DO NOT MODIFY THIS TEMPLATE BELOW THIS POINT"
%ImportSectionImpl%
%ImportImpl%
%list="NamespaceImpl" @separator="\n\n\n"%
```

Now you can provide custom implementations for the 'Import' and 'Import Impl' CTF templates.

Let's assume the following situation for a C++ environment:



You might want to use classes from the STL or a standard framework without having a fully (reverse engineered) UML class model for them.

The references to these class elements are expressed using «import» dependency links.

Additional attributes of these links are expressed using tagged values:

«import» dependency tagged value	Supported values
incl_type	• 'STD C' → generates `#include <target_element_name{incl_ext}>` statements • 'STD C++' → generates `#include <target_element_name{incl_ext}>` statements
incl_base_path	Generates `#include <{incl_base_path/}target_element_name{incl_ext}>` statements
incl_ext	Specifies the extension to use for the included target element header file. If nothing is specified, the following defaults are used: • incl_type == 'STD C' → extension = "`.h`"

«import» dependency tagged value	Supported values
	• incl_type == 'STD C++' → extension = ""

The 'Import' CTF templates to generate the code for these features are changed as follows:

'Import ' code generation template (C++) definition

```
// 'Import' template
%list="Connector__Import" @separator="\n" @indent=""%
\n
```

Custom CTF template 'Connector__Import' applied to connectors

```
%if connectorType!="Dependency" or connectorStereotype!="import"%
%endTemplate%

%PI=""%
$inclType = %connectorTag:"incl_type"%
$inclType = %TO_UPPER($inclType)%

$inclBasePath = %connectorTag:"incl_base_path"%
%if $inclBasePath != ""%
$termPathSep = %RIGHT("$inclBasePath",1)%
%if $termPathSep != "/"%
$inclBasePath += "/"
%endIf%
%endIf%

$inclExt = %connectorTag:"incl_ext"%
%if $inclExt != ""%
$extSep = %LEFT($inclExt,1)%
%if $extSep != "."%
$inclExt = "." + $inclExt
%endIf%
%else%
%if $inclType!="STD C++"%
$inclExt = ".h" + $inclExt
%endIf%
%endIf%

#include

%if $inclType=="STD C" or $inclType=="STD C++"%
<$inclBasePath%connectorDestElemName%$inclExt>
%endTemplate%

%qt%$inclBasePath%connectorDestElemName%$inclExt%qt%
```

These CTF template adaptions will generate the following C++ code in the dependent class header:

```
// 'Import' template
```

```
#include <string>
#include <vector>
#include <framework/foo/Class1.h>
```

This example shows the basic technique how to provide additional 'Import' / 'Import Impl' CTF template implementations. These will be applied once at header/source file level when the code is generated.

4 C specific Techniques

4.1 General Mapping of C constructs

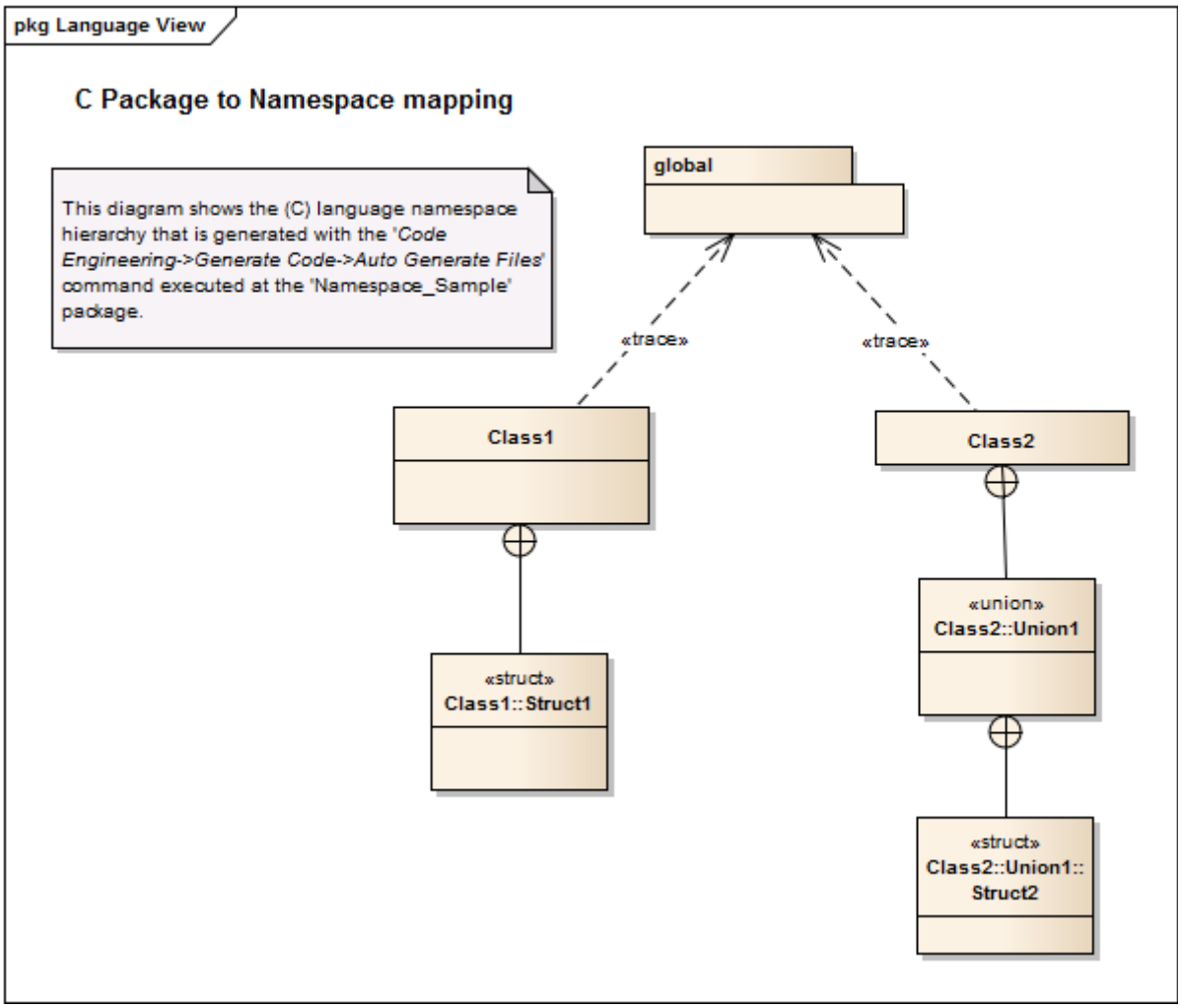
Packages and class elements

Classes are generated as C modules consisting of a <class>.h and a <class>.c file.

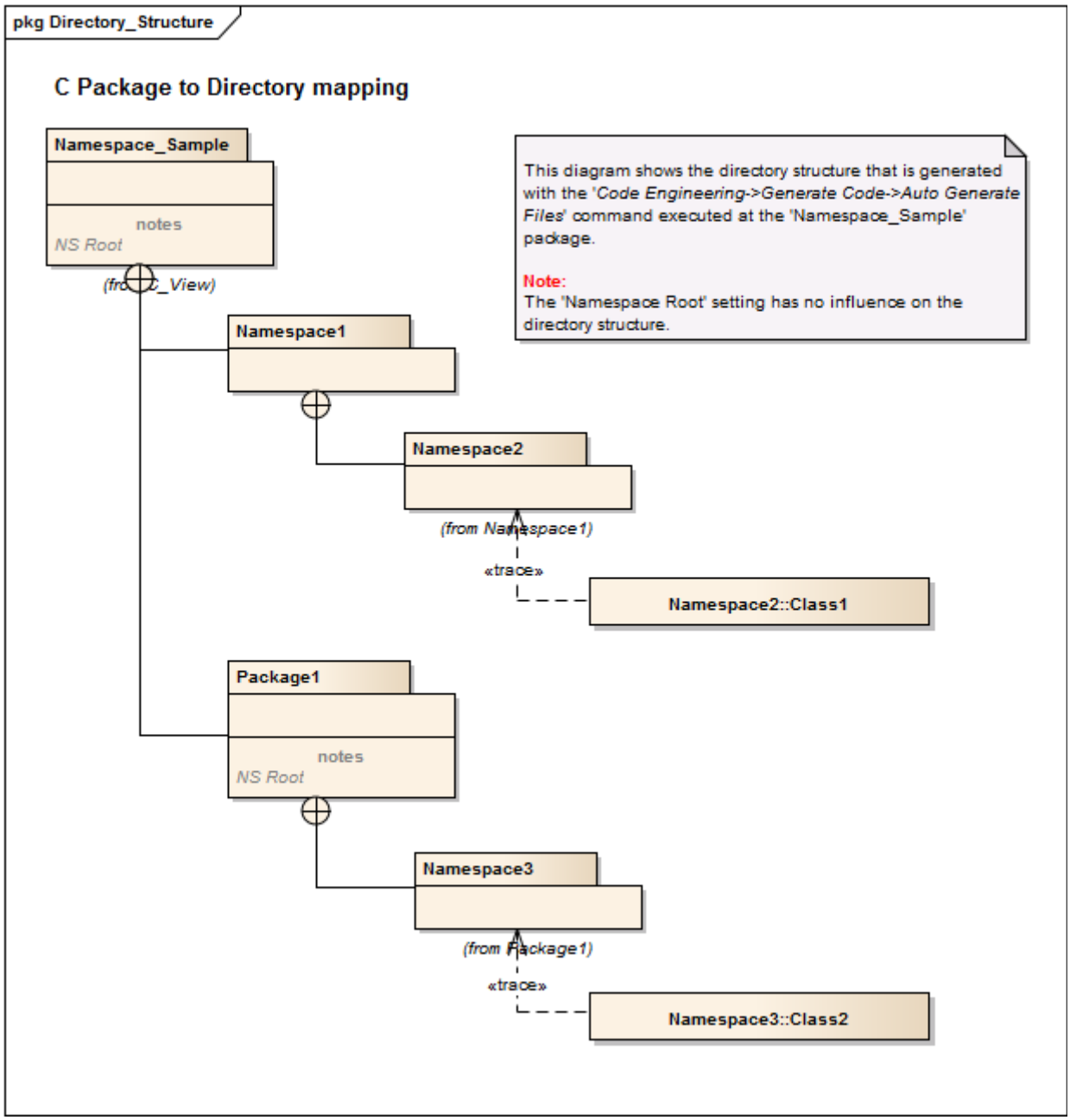
The header file will contain declarations for any public or protected features that appear as contained elements of the class.

The implementation file will contain declarations for any private features that appear as contained elements of the class, plus implementation (bodies) of the public features.

Package names and structure have no influence on the generated code:

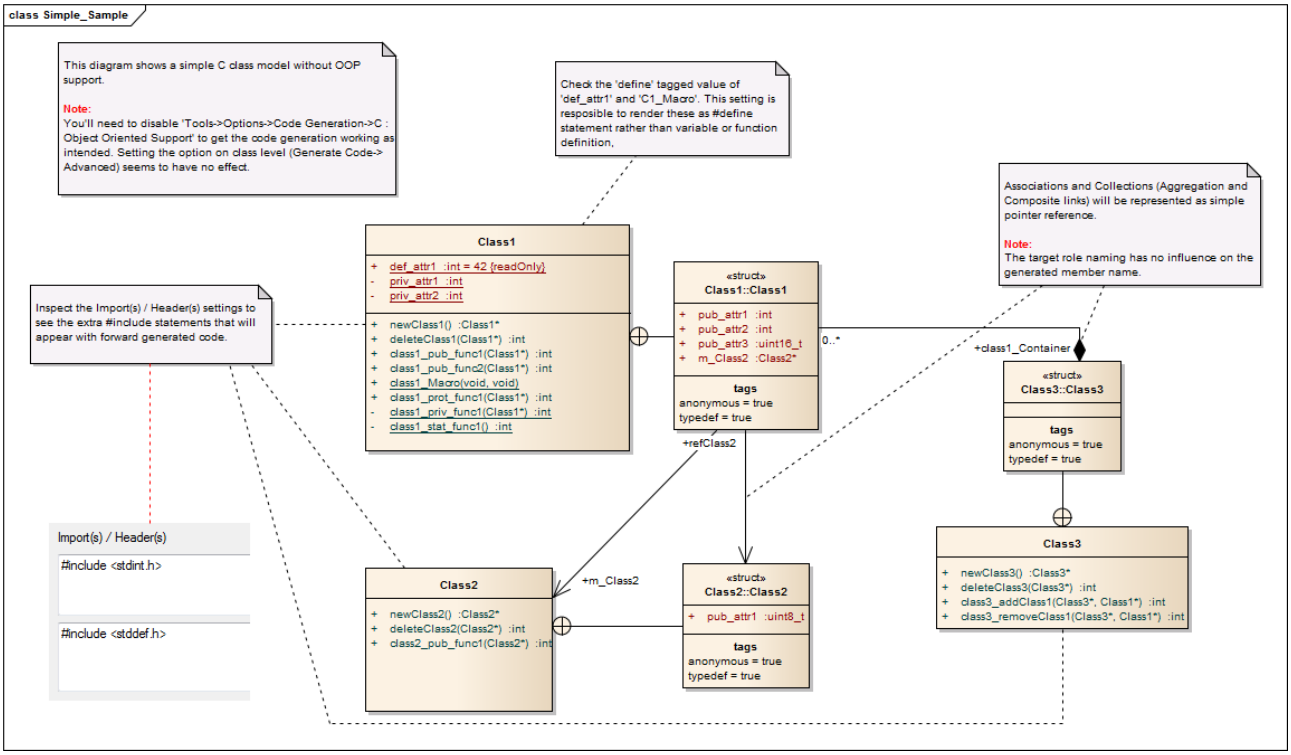


The files are placed in a directory structure that is a copy of the UML package structure though:



Setting the 'Namespace root' at packages has no effect for C code generation.

The following diagram shows how simple C modules can be modeled straightforward to generate useful initial code that can be synchronized after code changes without problems:



The following tables give an overview how UML model elements are mapped to C syntax and scope.

Attributes:

UML notation	UML Stereotype	UML Tagged Value(s)	C Header syntax	C Source syntax

Operations:

UML notation	UML Stereotype	UML Tagged Value(s)	C Header syntax	C Source syntax

Structs, Unions, Enums:

Bitfields:

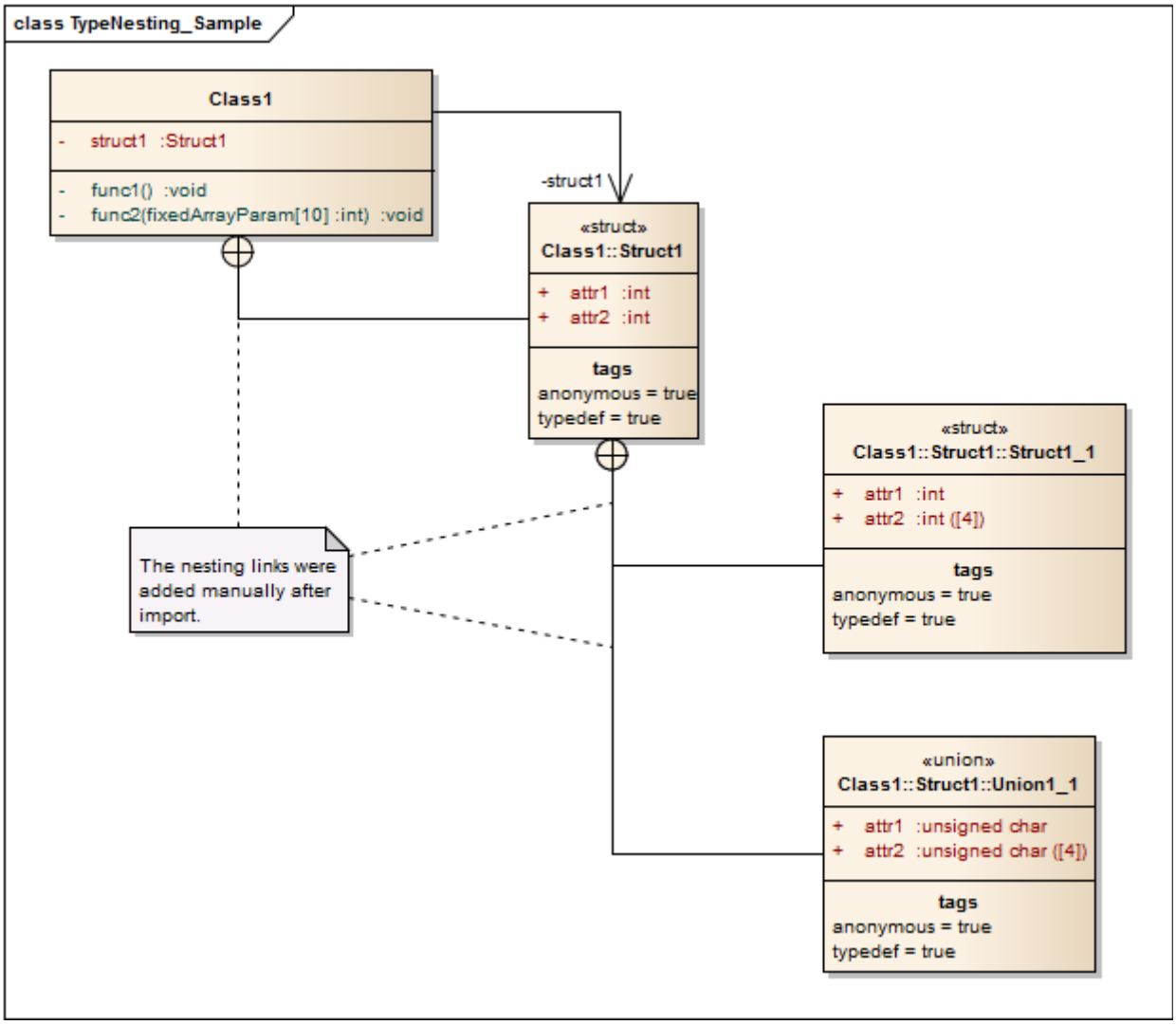
4.2 Anonymous nested structs and unions

To generate anonymous nested C `struct` or `union` elements like shown in the following code sample you'll need to set the 'typeSynonyms' tagged value vs. the 'typedef' and 'anonymous' tagged values shown in the general sample.

Header Class1.h

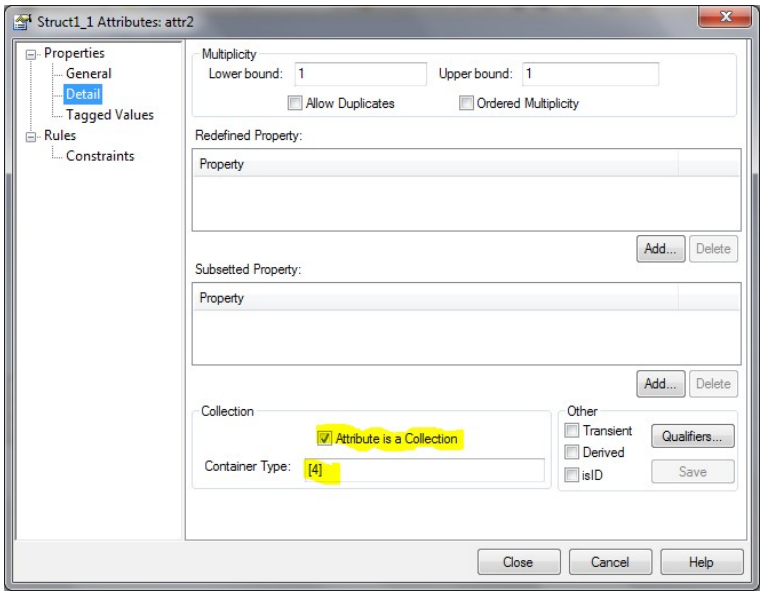
```
typedef struct
{
	struct Struct1_1
	{
		int attr3;
		int attr1;
		int attr2[4];
	}  Struct1_1;


	union Union1_1
	{
		unsigned char attr1;
		unsigned char attr2[4];
	}  Union1_1;
	int attr1;
	int attr2;
}  Struct1;
```

If you reverse engineer this code, you'll get a class model as shown in the following diagram:

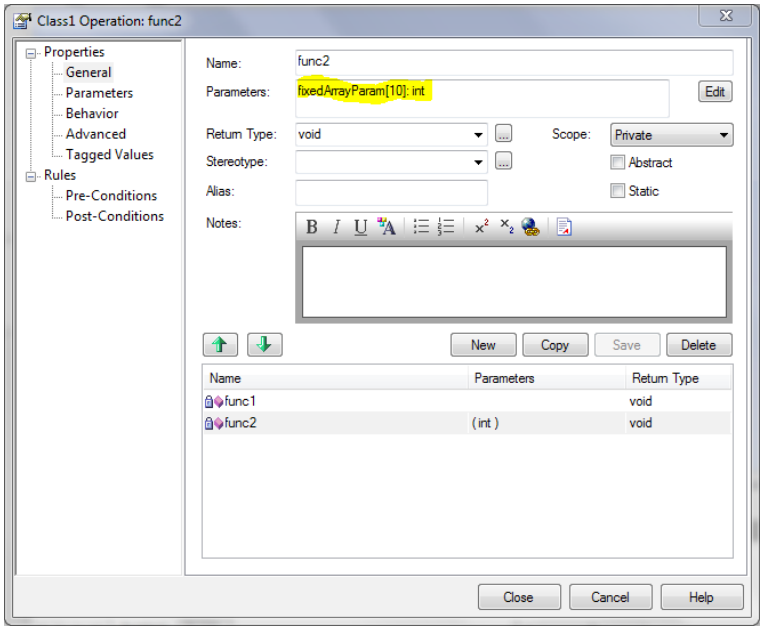


4.3 C (Fixed Size) Arrays

C array attributes can be modeled as shown in the sample from 4.2. To set the array specifier you can use the 'Collection' settings from the 'Details' page of the 'Attribute Properties' dialog:



C array parameters can be modeled as shown in the sample from 4.2. To set the array specifier you simply append it to the parameter name using the 'Parameters' field from the 'Operation Properties' dialog:



4.4 'extern' references

In general EA ignores the `extern` keyword during code parsing and no particular mapping for the C language is supplied with the standard code generation.

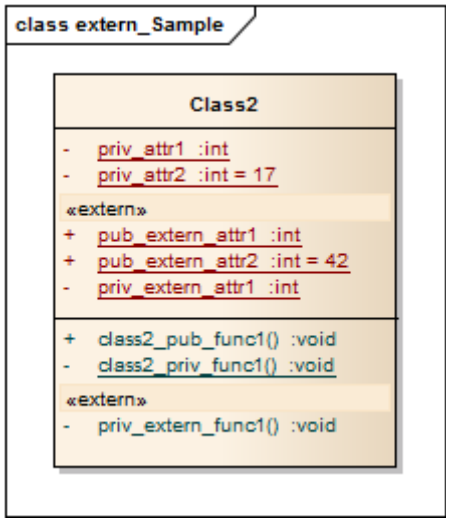
The extern keyword is implied for C99 by default, but might add more clarity for the actual declaration using it and where the attribute or function is finally defined in the compilation

units participating the model (view).

There are three cases to consider:

1. You want to export global (static) variable declarations where the definition is in scope of your module.
2. You want to explicitly import a global (static) variable into your compilation unit.
3. You want to explicitly import a global (static) function into your compilation unit.

Suppose the following class spec in UML:



This should look like this in code:

Header Class2.h

```
/** Exported attribute */
extern int pub_extern_attr1;
/** Exported attribute initialized */
extern int pub_extern_attr2;

/** Exported function */
void class2_pub_func1();
```

Source Class2.c

```
/** Exported attribute */
int pub_extern_attr1;
/** Exported attribute initialized */
int pub_extern_attr2 = 42;

/** Imported (static) attribute */
extern int priv_extern_attr1;
/** Private imported function */
extern void priv_extern_func1();

/** Private attribute */
int priv_attr1;
/** Initialized private attribute */
int priv_attr2 = 17;

/** Private function */
static void class2_priv_func1();
```

```
/** Exported function */
void class2_pub_func1()
{
    /* Implementation ... */
}

/** Private function */
static void class2_priv_func1()
{
    /* Implementation ... */
}
```

Unfortunately even though the C code generation templates can be adapted to generate the code correctly for attributes and operations marked with the «extern» stereotype, code synchronization will fail for the exported attribute definitions in the source file.

A trick to come around this behavior is to declare preprocessor macros for the definitions that need to appear in the source file and add these to the list of EAs (ignored) C++[1] preprocessor macros (see 'Settings->Preprocessor Macros …' command). These macros will be used then in the actual generated code.

Macro definitions in ea_common_defs.h

```
#if !defined(EA_pub_static_definition)
#define EA_pub_static_definition(type,name) type name
#endif

#if !defined(EA_pub_static_definition_init)
#define EA_pub_static_definition_init(type,name,initial) type name = initial
#endif
```

Code generated for Source Class2.c

```
/** Exported attribute */
EA_pub_static_definition(int,pub_extern_attr1);
/** Exported attribute initialized */
EA_pub_static_definition_init(int,pub_extern_attr2, = 42);
```

You'll need to specify `ea_common_defs.h` as imported class header or add it to the CTF `%File%` macro by default.

4.5 Function Pointers and Interfaces

4.6 Automatic OO Support for C

4.7 Behavioral Modelling for C

1 EA only provides C++ as language selection for preprocessor macro skipping on code reverse engineering, all the definitions there apply for C language though.

5 C++ specific Techniques

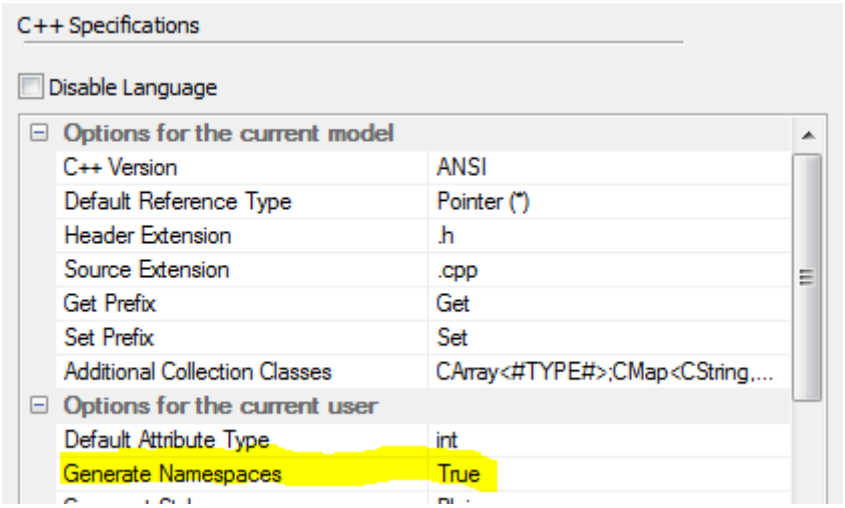
5.1 General Mapping of C++ Constructs

Packages and class elements

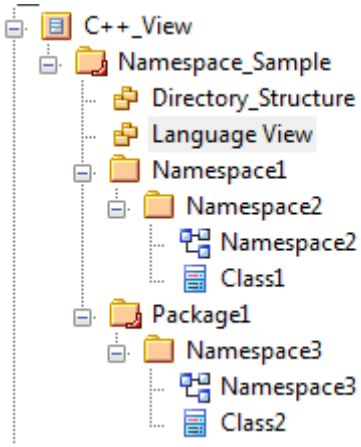
Classes are generated as C++ class modules consisting of a <class>.h and a <class>.cpp file.

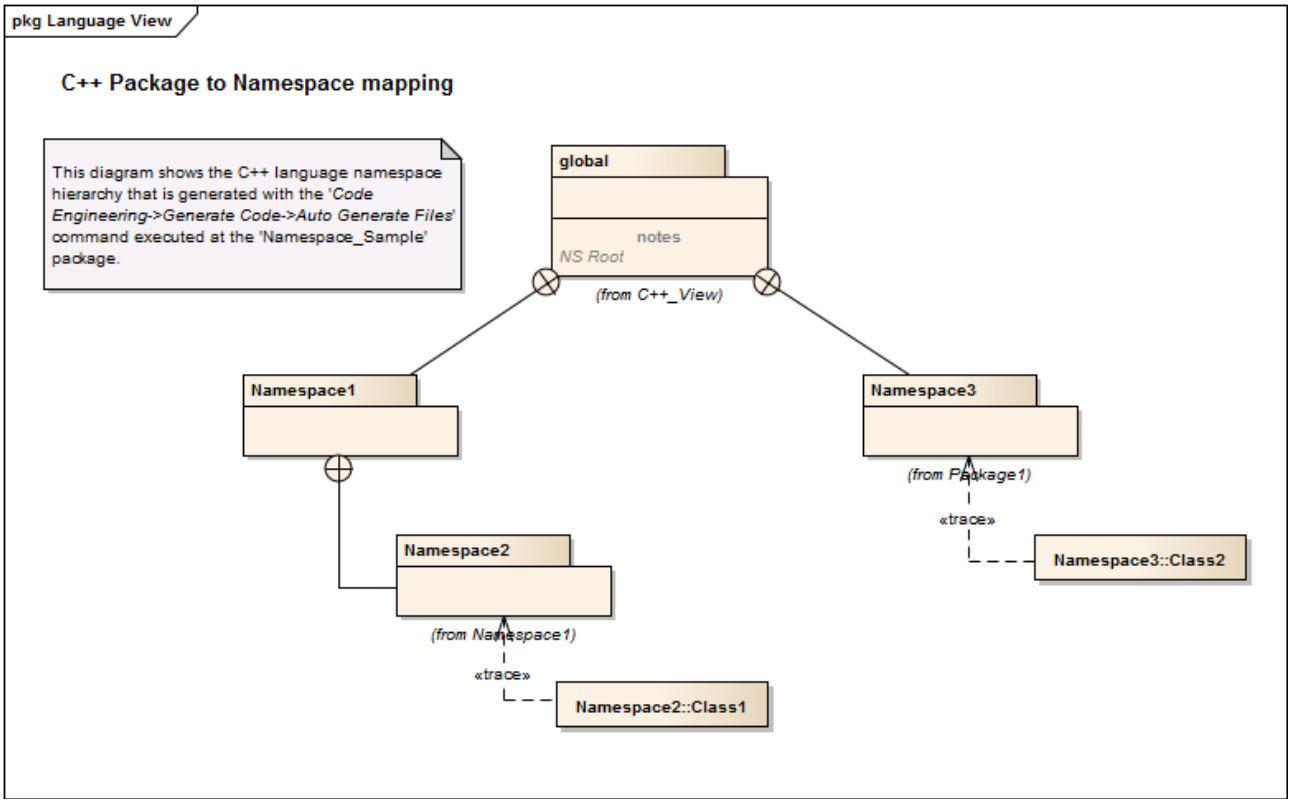
The header file will contain declarations for any features that appear as contained elements of the class.

If the 'Generate Namespaces' option in the 'Tools->Settings->Source Code Engineering->C++' dialog is set the class declaration is embedded in a namespace hierarchy:



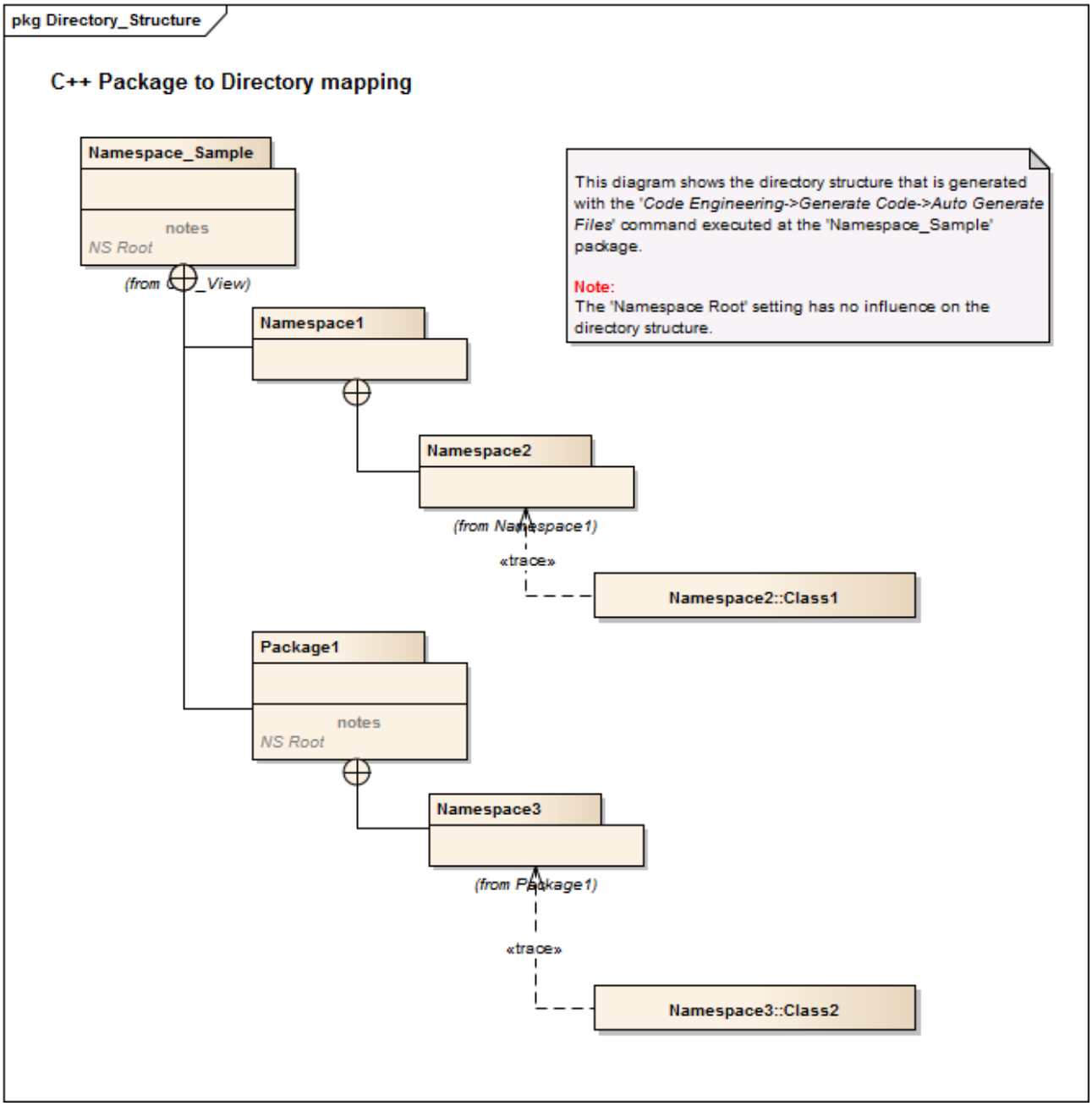
The namespaces are nested up to the next innermost package that is marked as 'Namespace Root' (see 'Code Engineering->Set as/Clear Namespace Root'):





The implementation file will contain (initial) definitions for the declared features.

The files are placed in a directory structure that is a copy of the UML package structure:



Attributes:

UML notation	UML Stereotype	UML Tagged Value(s)	C++ Header syntax	C++ Source syntax

Operations:

UML notation	UML Stereotype	UML Tagged Value(s)	C++ Header syntax	C++ Source syntax

UML notation	UML Stereotype	UML Tagged Value(s)	C++ Header syntax	C++ Source syntax

Structs, Unions, Enums:

Bitfields:

5.2 extern' references

5.3 Function pointers

5.4 Behavioral Modelling for C++

6 The Ext_C MDG Technology and AddIn

6.1 The Ext_C Profile

6.2 Ext_C AddIn support for stereotypes

6.3 Ext_C AddIn support for code generation features

Referenced Documents

[1] Geoffrey Sparks, Simon McNeilly, Vimal Kumar, Henk Dekker, Code Engineering Using UMLModels, 2010, http://www.sparxsystems.com.au/downloads/resources/booklets/uml_code_engineering.pdf

i This feature applies for the current EA10 Beta version only!