Development manual

COM2CORBA

Date: 2009-06-03
Version: 0.3.1 (revision 424)

This document

Summary A manual to help with further development of COM2CORBA.

Author Joel Purra

Date 2009-06-03

Version 0.3.1 (revision 424)

Document history

Version	Date	Changes
0.1.0	2009-05-31	Document created
0.2.0	2009-05-31	Added templates
0.3.0	2009-05-31	Added information on further development and code hints
0.3.1	2009-06-03	Introduction and diff

Contents

Chapter 1

Introduction

This document specifies the syntax of templates and some hints on how to get started with further development of COM2CORBA. For the structure and design of the project we refer to the architecture notebook and the design document. Terms are explained in the glossary document.

COM2CORBA is built to be a template based code generator for IDL files. It is based on ACE+TAO and works well to create wrappers to their C++ generating backend. By using a template system, extensible by variables and subscopes, flexibility of code output is good.

Special constructs can be achieved by modifying the COM2CORBA Preprocessor. Some specifics are currently implemented to make COM2CORBA backwards compatible with FWIZ, a predecessor used internally at BAE Systems Hägglunds AB.

Chapter 2

Templates

Templates define the content generated for specific nodes in the AST. They are mostly plain text that pass the generator unparsed, except for the variables and children as defined below.

2.1 Naming

Templates are named with the same name as their respective visitor function, for example "visit_interface". See the ast_visitor interface for more details. There are no visit_interface or visit_structure templates, instead interfaces and structures are merged into classes which are generated from the visit_class templates instead.

2.2 visit_root

The first node to be visited, once and only once, which generates the basic file data, for example #include statements.

2.3 Variables

In the templates, text depending on the IDL file are can be accessed using variables. The value of these variables are found in the AST and replaced textually where applicable in the template.

For example, when generating code for an interface it's name is needed:

IDL

```
interface MY_INTERFACE { }
```

Template

```
class $Name$ { }
```

Generates

```
class MY_INTERFACE { }
```

2.3.1 Variable case

Depending on how the template looks, different variables values may need to be of different cases in different places. The same variable may be output in different case with the following syntax.

$name$ becomes all lowercase, for example *bankaccount*.

$Name$ becomes original case, for example *BankAccount*.

$NAME$ becomes all uppercase, for example *BANKACCOUNT*.

2.3.2 List of variables

This list is an incomplete, general version. For a full list, including node specific variables, see the source code in TemplateVisitor.cpp.

idl Name of the input IDL file, without the .idl extension.

$name$ Name of the current node being generated, for example MyString.

$degarnished_name$ A special version of a name where certain prefixes and suffixes are stripped. For example, "House_struct" becomes "House".

$fullscope$ The full name of the current node, for example MyModule::MyInterface::MyString.

$parentscope$ The full name of the current node's scope, for example MyModule::MyStruct.

$username$ The username of the user executing COM2CORBA.

$time$ Time in the format YYYY-MM-DD hh:mm:ss.

$template$ The name of the template, for example the relative path to the template file.

$guid$ A GUID for the module of interface.

2.4 Traversing

Children of a template will be traversed at the point in the template where the

```
$...$
```

syntax is inserted. To build upon the variable example:
IDL

```
interface MyInterface { string MyStringName; }
```

Templates

```
visit_class[1]: class $Name$ { public: $...$ }
visit_field: $Datatype$ $Name$;
```

Generates

```
class MyInterface { public:   CString MyStringName; }
```

[1] See Section 2.6 why this is not visit_interface.

2.4.1 Indentation

The indentation can be increased when generating code for children this is done with the following syntax

```
$...indentation=k...$
```

where k is the number of extra levels.

2.4.2 Scope

Some templates require child templates to generate different code for the same definitions. To define what code is generated where, scopes can be used. When specifying scope the generator will switch the change the current template folder into a subdirectory with the same name as the scope. The current template folder is the directory from which the generators looks for templates.
The syntax is

```
$...scope=subscope...$
```

where sub scope is also the name of a sub folder to the current scope. If a template is missing in the new scope, a new dummy file will be created.

2.4.3 Separator

Some templates are repeated in a list where they need to be separated. Because the separator character only is valid *between* template elements, not after the last one, a separator can be defined for the child templates.
The syntax is

```
$...separator=chars...$
```

where chars is one or more characters that will be used to separate the child templates. If undefined, this defaults to \n.

2.4.4 Combinations

An arbitrary number of arguments can be specified by separating them by "...".
The syntax is

```
$...option1=value...option2=value...$
```

where each option is separated by "...".

2.5 Counters

Some templates need to have a counter to remember an integer.
The syntax is

```
$counter=myCounter$
```

Creates a counter that counts up every time it is used, begins at zero.

```
$counter=myCounter:3$
```

Same as the one before but starts at three.

```
$counter=myCounter:free$
```

Destroys the counter with the name myCounter, if you trying to used it after it have been destroyed it begins over with zero as starting value if nothing else is declared.

2.6 Special cases

Some virtual modifications to the normal TAO node structure has been applied in COM2CORBA. These changes are inherited from the earlier FWIZ version of the generator. Since changes to the AST tree would mean disturbing the TAO generator, separate lists of COM2CORBA specific changes and nodes has to be kept. These lists should then be used for look ups and special virtual visitor methods called.

2.6.1 AST_Class

Both AST_Structure and AST_Interface are expected to generate the same code. They also have some overlapping and in the process of creating AST_Class objects, some merging by the Preprocessor is necessary. An interface called DepartmentItem converts to a class called DepartmentItem. A structure with the name DepartmentItem_struct converts to a class called DepartmentItem. If there is both an interface and a struct (disregarding the "_struct" suffix) with the same name, they are merged to one class.

In our own BasicVisitor, visit_interface and visit_struct have both been overloaded to invoke the virtual visit_class on the corresponding COM2CORBA internal AST_Class instead. There are no separate templates for interfaces and structures, both use the class templates.

2.6.2 AST_SeqSet

In example IDL files, there are a lot of sequence typedefs referenced later on. These sequences with the "seq_" prefix are treated in a special way in some places, and therefore they are extracted to a an COM2CORBA representation called AST_SeqSet.

The types of nodes are separated in the Preprocessor and treated specially in the BasicVisitor. Due to the fact that there are both AST_SeqSet and AST_Typedef nodes present, the TemplateVisitor implements both visit_seqset and visit_typedef_original methods.

Chapter 3

Further development

3.0.3 Using a diff program to create or update templates from an existing source

Creating templates can easily be overwhelming when first glanced at, but don't fret. Using a graphical diff program is very helpful as you can see the differences between your target code, your generated code or your other generated code.

First, generate code for one test case. Then generate code for a very similar test case - add, change or remove something small. When you spot the differences, make changes to the corresponding template. Remember to test big and advanced cases as well - they can be a whole lot different, especially if they have a lot of child nodes requiring templates in new subscopes.

We recommend the open source program Winmerge, found at `http://winmerge.org/`. It can compare entire folders, highlight syntax, color specific changes down to character level, ignore white space, detect moved blocks and more.

3.1 Extracting and reusing ODL file GUIDs

Currently, COM2CORBA only regenerates GUIDs each time the ODL is created. Instead, GUIDs should be extracted from the ODL and reused.

One idea would be to use the Microsoft tool on an existing ODL file to generate a binary format Typelibrary (TLB) file. The TLB can then be accessed using Microsoft's interfaces for TLB information, thus allowing for GUID extraction.

There's also the (less flexible?) possibility to extract the info using searching through the ODL file for certain strings.

3.2 A filtering proxy visitor

There are certain places where code is generated from nodes that are not in child scopes of the current node. Jumps and filters like these have not yet been implemented. One suggestion would be to create a new syntax in the form $...scope=*subscope*...action=argument...$ and use that to traverse the tree from the root node whilst filtering out nodes that aren't supposed to be rendered.

3.3 Sequences need to have another special case

Currently, sequence typedefs are extracted and used, but there's still the special cases where sequences (fields using a type with the prefix "seq_") generate special code in some places. They are reached through the fields of classes, thus there would need to be another check in the Preprocessor. This check would create new sequence based field node and then use them in BasicVisitor's visit_field to separate normal fields from sequences.

3.4 Collections aren't treated in a special way

No special treatment has been implemented for collections; that is, objects with a "_coll" suffix.

Chapter 4

Code hints

- Most special cases in the output are probably best fixed with new variables in the TemplateVistor (see visit_field).
- A few checks and a new map in the Preprocessor would make a special case for bigger changes where the structure is different in the IDL and the output (see AST_Class). Calls to the visitors are virtual, so make sure there are no virtual inherited virtual methods left after you've implemented the special cases in the BasicVisitor.
- Some places have too many calls to templates that generate code, typically typedef templates. Using empty templates would remove the extra comments, but make sure to use the $...scope=*subscope*...$ syntax to change to a new set of templates where an actual template is needed.
- The debug folder creates a template tree structure presentation based on your input IDL file. This is helpful for debugging.
- Templates that are missing will be created with dummy contents telling you the path of the template.