Design

COM2CORBA

Date: 2009-05-20
Version: 1.5.3 (revision 424)

This document

Summary This document presents the design for the COM2CORBA project.

Author Daniel Svensson

Date 2009-05-20

Version 1.5.3 (revision 424)

Document history

Version	Date	Changes
0.1.0	2009-02-22	Document template created.
0.2.0	2009-02-24	Added system overview.
0.3.0	2009-02-25	Added subsystems.
0.4.0	2009-02-25	Added realization, and bibliography.
0.5.0	2009-02-26	Updated figures and fixes in whole document.
0.5.1	2009-02-27	Fixed some typos.
0.5.2	2009-03-02	Removed inclusion of the glossary document, fixed some typos.
1.0.0	2009-03-09	First version.
1.1.0	2009-03-18	Added class diagram for AST classes.
1.2.0	2009-05-06	Uppdated chapter 1-3, added empty section for templates and TemplateVisitor.
1.3.0	2009-05-06	Uppdated TemplateVisitor and Templates as well as updated pictures.
1.4.0	2009-05-07	Added odlVisitor image.
1.5.0	2009-05-07	Corrected version.
1.5.1	2009-05-08	More info on templates.
1.5.2	2009-05-19	Proofread and corrected.
1.5.3	2009-05-20	Added information on the COM2CORBA AST transfomer.

Contents

Chapter 1

Design structure

1.1 Product overview

COM2CORBA extends the TAO IDL compiler by adding support for generating C++/COM files which can be compiled to generate a dynamic link library that exposes CORBA services through standard COM interfaces.

1.2 Document dependencies

This documents assumes the reader has knowledge of the design of the TAO IDL compiler [1] and about what one must know to write a backend (which among ofter things include information about the AST) [3].

This document does not contain details about the code that should be generated since it should be almost identical with the output produced by the reference implementation [2].

Se the Glossary for definitions and explanations for many terms.

1.3 System overview

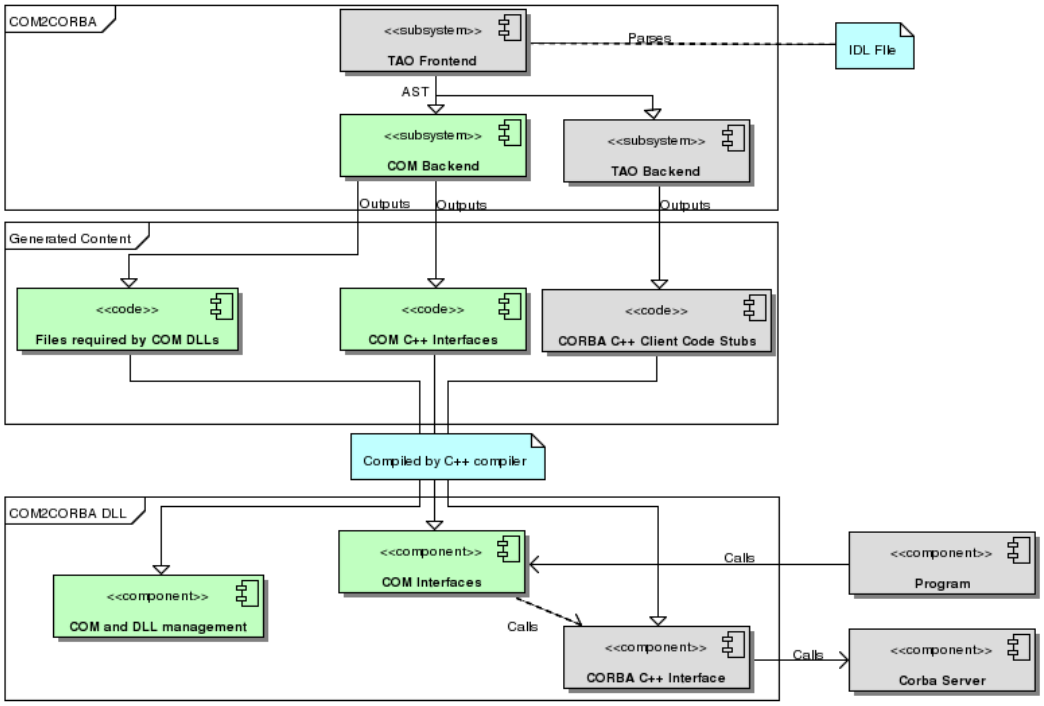

Figure 1.1: System overview of COM2CORBA. Gray objects represent non-modified existing products.

Chapter 2

Subsystems

2.1 TAO frontend

This is the frontend shipped with TAO and used in the TAO IDL compiler. It parses specified IDL files and generates an AST describing the files content. The frontend's design is documented in [1] and [3].

2.2 TAO backend

This is the backend shipped with TAO and is used in the TAO IDL compiler.

It generates C++ classes that encapsulates calls to CORBA services from the AST. It is also documented in [1].

2.3 COM backend

The COM backend generates all the COM related files required to compile a COM2CORBA DLL and is the heart of COM2CORBA. It consist of :

1. AST transformer
2. CPP interface backend
3. Project generator
4. ODL generator

2.3.1 AST transformer

Pre-visits all the nodes in the AST before the other visitors. The "transformation" is actually creation of a separate list of nodes, outside of the AST, to be used internally by COM2CORBA.

Currently, the visitor merges AST_Interface and AST_Structure nodes into (COM2CORBA internal, specific) AST_Class nodes. These nodes combine the behavior of special structures with a naming convention; they have the same name as existing interfaces from the same IDL, but with a "_struct" postfix.

2.3.2 CPP interface backend

The CPP interface backend uses the AST to generate the C++ code that defines and implements the classes produced by the old code of which some are defined in the IDL file. These COM implementations calls the corresponding classes and functions generated by the TAO backend. The CPP interface backend is made up of different code generators that generates code from templates based on the visitor pattern.

2.3.3 Project generator

The Project generator generates the code required by the COM2CORBA DLL not associated with any particular interface.

The project generator also implements the API functions defined under "REQUIRED API THAT EACH BE MUST SUPPORT" in [3].

2.3.4 ODL generator

The ODL backend generates an ODL file [1]. If a file already exists it uses the old file to reuse old GUID's and information about in which order functions and data members appear . GUIDs are then stored in memory using a dictionary so that they can be queried by another visitor. Entities missing GUIDS will be assigned new GUIDS.

[1] ODL file syntax can be found at `http://msdn.microsoft.com/en-us/library/ms221683.aspx`

Chapter 3

Patterns

3.1 Visitor

3.1.1 Overview

Since the internal representation already is fixed by the TAO frontend the visitor pattern allows the addition of new operations to existing classes without modifying those classes. Examples and motivations for using the Visitor pattern can be found in [1].

3.1.2 Details

The visitor class contains separate functions that should be called depending on the type of object visiting. When visiting an interface the visit_interface function should be called, for a enum a visit_enum functions should be called and so on. The objects to be visited has a function, often called accept, which takes a visitor class and in turn calls the appropriate visit function. See figure 3.1 for a simple sequence. More examples including code can be found in [5] and in [1].

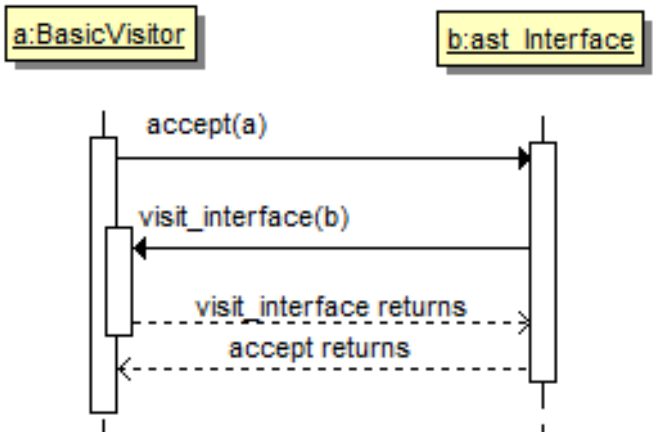

Figure 3.1: Sequence diagram for visiting a AST node of type ast_interface.

Chapter 4

Requirement realizations

4.1 Project generator

Not really a module or class, the project generator generates code required by the COM2CORBA DLL not associated with any particular interface. The project generator also implements the API functions as defined under "REQUIRED API THAT EACH BE MUST SUPPORT" in [3]. These functions calls the corresponding functions in the TAO backend and uses the code generators of the CPP interface backend to generate all output files.

Since much of this code is almost the same for each project the code is stored as template files with special symbols that is replaced depending on the file name of the parsed IDL file etc. To achieve this the code generating functions of the BasicVisitor class is used.

For details about naming of files and how the code should look like refer to output provided by old version of generator with an empty IDL file as input.

4.2 Code generation

4.2.1 Templates

To produce the required code a careful mixing of static code (code that don't change) and dynamic code (code that depends on AST node data) must be generated. In order to make it as simple as possible to do this with limited programming skill, as well as making it simple and fast to fix defects and make changes in the produced code without requiring a recompilation. A approach based on templates is applied. Templates are just simple plain text which corresponds to the static text that should be produced by a certain type of node combined with certain variables, which are substituted at run-time, to enable the dynamic content to be generated. See Appendix B for a description of how templates work.

To make it simple to use templates from different sources like pure text, files or embedded resources a simple Template interface (figure Figure 4.1 on page 11) is used for loading templates.

4.2.2 Class hierarchy

As shown in figure Figure 4.2 on page 11 all code generators are based on the visitor pattern and are derived from the ast_visitor interface defined by the TAO IDL compiler. The interface of the ast_visitor class can be seen in Figure 4.3 on page 12. A class diagram for the different AST classes can be found in A on page 15. One instance of each concrete visitor type is created and they independently parses the AST produced by the TAO frontend. Since some pre-parsing

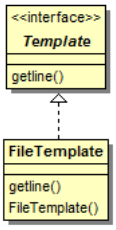

Figure 4.1: Template class diagram

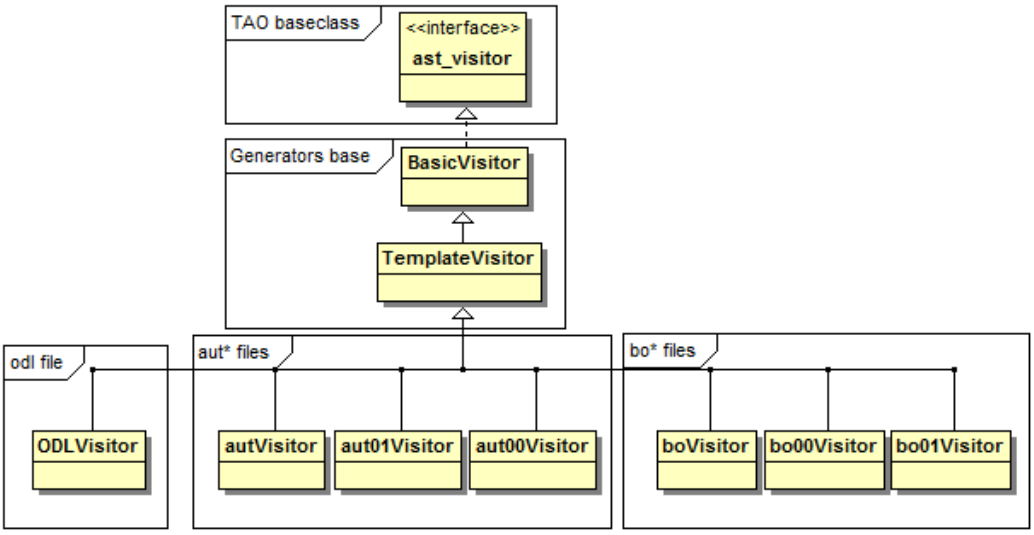

Figure 4.2: Class diagram for generator classes

is needed, the AST transformer, only inheriting from ast_visitor, is the first visitor to traverse the AST.

4.2.2.1 Transformer

Pre-parses the AST to extract COM2CORBA specific nodes.

All instances of AST_Interface and AST_Structure are converted into the COM2CORBA specific AST_Class node in an effort to emulate what is rendered by previous versions of COM2CORBA. AST_Interfaces and AST_Structure are merged with each other, if they have the same name plus the postfix "_struct" on the AST_Structure. Merging means creating a new AST_Class node and then adding all the functions to that node. The BasicVisitor then, unless overridden, visits the AST_Interface and AST_Structure nodes only indirectly through their AST_Class representation.

4.2.2.2 BasicVisitor

The BasicVisitor contains common functions used by the different visitor classes such as a function to visit all children of an AST node in depth first order.

Each instance is associated with a file which it is responsible for writing formated code to. It can output both plain text and code generated from templates.

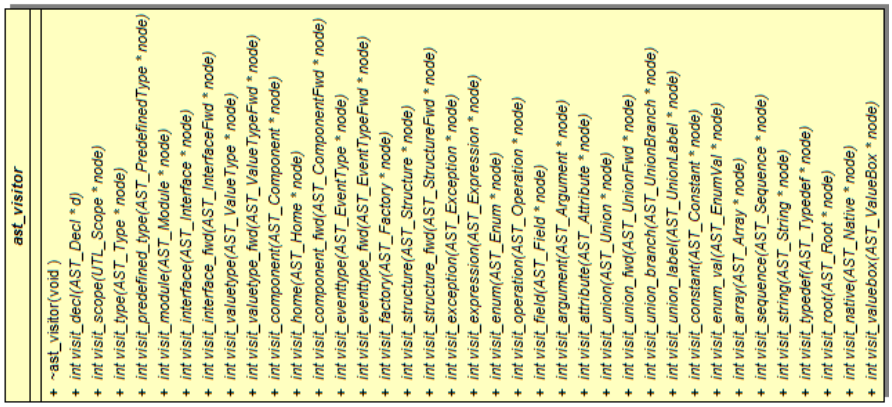

Figure 4.3: Definition of the visitor interface

The BasicVisitor also helps keeping track of the current context when parsing files. The context contains information about the parent nodes so that the the current module, class, function, etc can be obtained as variables when generating code from templates.

4.2.2.3 TemplateVisitor

The template visitor extends the BasicVisitor into becoming a general code generator based on the visitor pattern. When a TemplateVisitor is created a starting scope must be specified. The TemplateVisitor will append this scope to the search path for templates. [1] When any of the visit_* functions defined in the ast_visitor (figure Figure 4.3 on page 12) is called the TemplateVisitor first sets node specific variables (so that they can be accessed from within the template) and then it loads the template with the corresponding name in the current template folder and generates code from it. An output filename can be given or one is generated in the manner required by the aut*- and bo*-Visitors.

When beginning to parse children (as a consequence of a $...$ type variable) the TemplateVisitor is able to change the current directory if a scope argument is specified. In this way different code can be produced in different places for the same node in the AST without requiring different visitors to be written each time different output is required based on the nodes scope.

4.2.2.4 ODLVisitor

The ODLVisitor generates an ODL file [2] if none exists. If an ODL file exists it should load the ODL file and reuse old GUIDs if possible.

It also stores GUIDs in memory using a global dictionary that can be queried by a TemplateVisitor.

4.2.2.5 aut*Visitor and bo*Visitors

These classes are all responsible for one of the different output files, “IDL_aut.h”, “IDL_aut00.cpp” , “IDL_aut01.cpp”, “IDL_bo.h”, “IDL_bo00.cpp” and “IDL_bo01.cpp” where IDL is replaced by

[1] If the template folder is named “templates” and the scope is “scope” then it will look for templates under “templates/scope/”

[2] ODL file syntax can be found at `http://msdn.microsoft.com/en-us/library/ms221683.aspx`

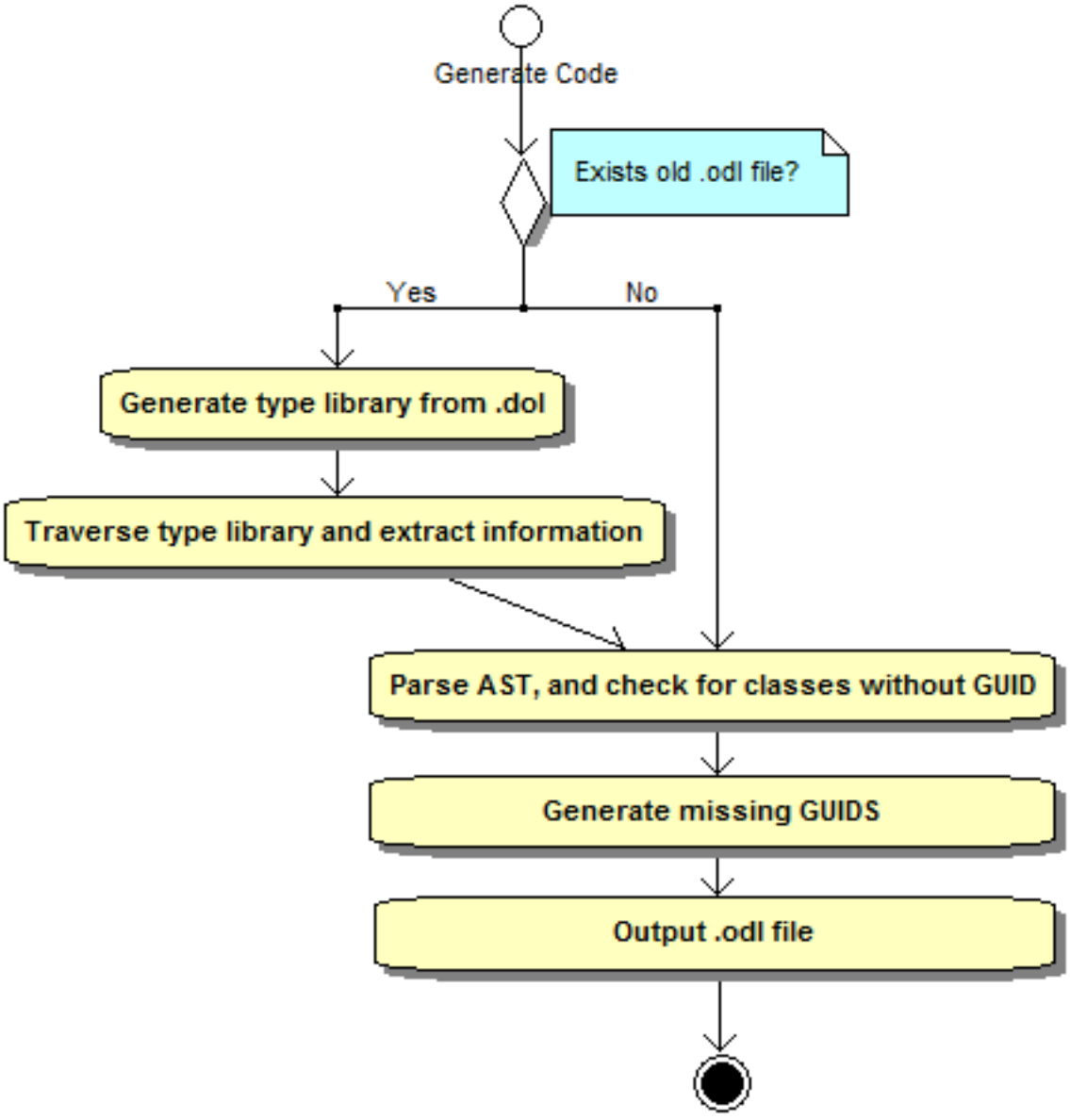

Figure 4.4: Flowchart for generating a .odl file

the name of the input idl (apart from the ".idl" part). If special handling of any of the visit functions is required for any of these files, the functions are overridden here. Apart from that the other use of these class is to provide a way to automatically generate the correct filenames.

Appendix A

AST class diagram

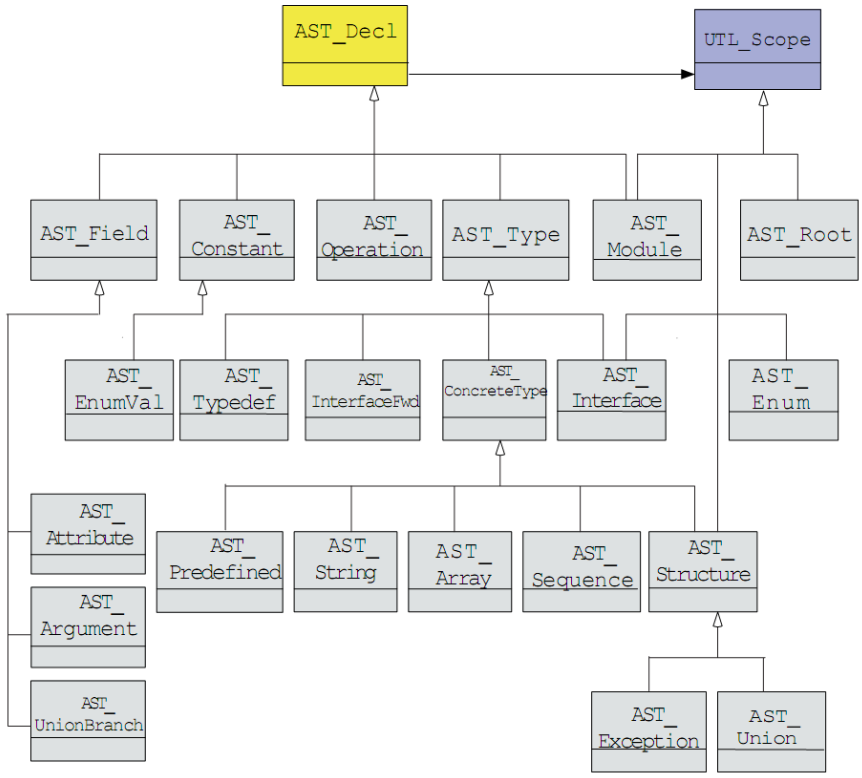

Figure A.1: Abstract Data Tree, class diagram

Appendix B

Templates and their use

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.

B.1 Generators

Each file that COM2CORBA generates has a special generator that inherits from the template visitor. This generator contains information on where to look for the template file; by default a subfolder to the templates folder with the partial filename of the generated file. This is the starting scope where the individual templates are expected to be found.

B.2 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.

B.2.1 visit_root

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

B.3 Variables

In the templates, many changes depending on the IDL file are transfered using variables. The value of these variables are found in the AST and replaced textually where applicable in the template.

For example, the name of an interface:

IDL

```
interface MY_INTERFACE { }
```

Template

```
class $name$: public EXAMPLE_INHERITANCE { }
```

Generated

```
class MY_INTERFACE : public EXAMPLE_INHERITANCE { }
```

B.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 MY_INTERFACE { string MY_STRING_NAME; }
```

Template

```
class $name$: public EXAMPLE_INHERITANCE { $...$ }
```

Generated

```
class MY_INTERFACE : public EXAMPLE_INHERITANCE { public:   CString MY_STRING_NAME; }
```

B.5 Indentation

The indentation level may be explicitly defined with the syntax

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

where k is the number of levels.

B.6 Scope

Some templates require child templates to render different code in several places. To define what code is generated where, scope is introduced. This defined subscope selects a subfolder where to look for further templates.
The syntax is

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

where subscope is also the name of a subfolder to the current scope.

B.7 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.

B.8 Combinations

Traversing options may be combined indefinitely by separating them.
The syntax is

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

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

Bibliography

[1] "Applying C++ Patterns, and Components to Develop an IDL Compiler for CORBA AMI Callbacks"

[2] Generated source code and reference generator that can be downloaded from the COM2CORBA project website [4].

[3] Included in TAO download at ACE_ROOT/TAO/TAO_IDL/DOCS/WRITING_A_BE

[4] The COM2CORBA project website at `http://redmine.dillus.se`

[5] `http://en.wikipedia.org/wiki/Visitor_pattern` (2009-02-26)

[6] `http://en.wikipedia.org/wiki/Abstract_factory_pattern` (2009-02-26)