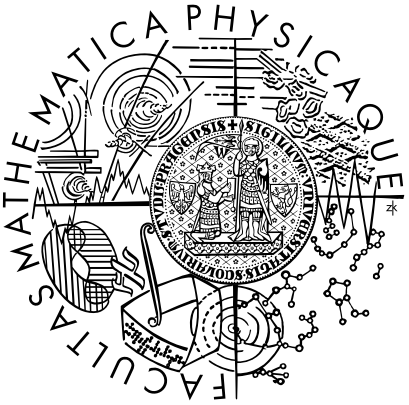
Faculty of Mathematics and Physics
Charles University in Prague

Master Thesis



Vojtěch Hála

Using XML Technologies to Apply Design Patterns

Department of Software Engineering

Supervisor: RNDr. David Bednárek

Study Program: Computer Science

In the first place I would like to thank to my parents and also my wife for providing me a comfortable background throughout the time I have been working on this Thesis. Thanks also to my colleagues that were cooperating on the Lestes project for many ideas on improving the nascent framework. Last but not least, I would like to thank to my supervisor RNDr. David Bednárek for worthy advices to the compiler project and to this work.

Prohlašuji, že jsem svou diplomovou práci napsal samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce.

V Praze dne 16. dubna 2009

Vojtěch Hála

Contents

Title: Using XML Technologies to Apply Design Patterns

Author: Vojtěch Hála

Department: Department of Software Engineering

Supervisor: RNDr. David Bednárek

Supervisor's e-mail address: david.bednarek@mff.cuni.cz

Abstract: Although contemporary programming style involves massive use of design patterns, programming languages does not offer suitable means tu support their application. Aim of this work is to show in practice that modern XML technologies, namely XSL Transformations, allow developers to avoid some routine tasks required by the objective language itself. This reduces the probability of errors, allows developers to focus on the key parts of the design, and makes maintaining the code markedly easier. These benefits come to light especially in large projects with hundreds to thousands of classes with complicated relations. In this Thesis we demonstrate these ideas on an example of a C++ compiler project.

Keywords: software, objective, design, patterns

Název práce: Využití XML technologií při aplikaci návrhových vzoru

Autor: Vojtěch Hála

Katedra: Katedra softwarového inženýrství

Vedoucí diplomové práce: RNDr. David Bednárek

E-mail vedoucího: david.bednarek@mff.cuni.cz

Abstrakt: Přestože moderní styl programování vyžaduje široké nasazení návrhových vzoru, programovací jazyky nenabízejí pohodlné prostředky k jejich efektivní aplikaci. Cílem této práce je ukázat v praxi, že zapojení moderní XML technologie, zejména XSL transformací, umožní vývojáři vyhnout se některým rutinním činnostem vyplývajícím ze samotného objektového jazyka. Tím se snižuje pravděpodobnost chyb, vývojárum je umožněno soustředit se na podstatu návrhu a významnou měrou se zjednodušuje údržba kódu. Tyto výhody se projeví zejména ve velkých projektech se stovkami až tisícovkami tříd se složitými závislostmi. V této práci demonstrujeme zmíněné myšlenky na příkladu projektu překladače C++.

Klíčová slova: software, objektový, návrh, vzor

Chapter 1

Overview and Motivation

1.1 What Is a Design Pattern

Design patterns were proposed in a landmark book [2] by Erich Gamma et al. in 1995. Their work defined and carefully clasified 23 frequently used basic elements of a transparent and reusable object-oriented software design. They proclaimed that knowledge of these (and similar) patterns makes the difference between experienced profesional designer and a newbie with basic knowledge of object-oriented principles. Time has clearly shown that they were right and the book is still actual despite its age. During that time, new patterns were described, the old definitions were specified more precisely or generalized, and other sorts of classification were made [1].

1.2 Generated Code

Using the patterns leads in some particular cases to repeated writing of very simmilar code in various classes. Gamma et al. have pointed that out already in the 1995 book [2]. Routine work is a possible source of mistakes and should be left to machines. Therefore, various code generators are used to help with coding.

1.3 XML Technologies Approach

In this work and in the Lestes project [3], we decided to describe the patterns, classes and relations between them in a set of XML documents. The resulting compiler is to be used for educational purposes and the documents were meant to be a source for generating multiple

outputs such as data dependency graphs, request flow graphs and also the common parts of C++ codes for the classes. XML documents and XSLT stylesheets therefore seem to be the right natural tool to perform this task. The first idea was to use some existing and well defined set of XML tags to describe our classes and such set was the XSD Schema. It appeared, however, very soon that these tags fit good to the purpose they were designed for - describing XML formats, but they don't fit our needs, because the relations between classes are quite different from relations between XML elements and attributes in a document. Therefore we decided to develope our own set of elememts and format of XML files called LSD - Lestes Structures Definition. This choice has given us a freedom to change (mainly extend) the format operatively for our actual needs. The following chapter 2 of this Thesis describes what we achieved and how our tool can be used. Parts of the chapter are taken from the project documentation which can be found on the project website [3].

1.4 Source code included

Compact disk attached to this work contains the whole Lestes project source code, version 1.0. The `lestes-1.0/util/lsg` directory contains the code of the Lestes Structure Generator (LSG) and some example files. There is a `Makefile` in this directory, which is separate from the rest of the compiler build system and it serves only to compile the LSG sample files to perform generator tests and/or debuging. Instructions and requirements to compile the whole project can be found in the documentation on the project website [3].

Chapter 2

Structure Generator

The goal of the generator (LSG, Lestes Structures Generator) is to allow easy maintenance of complex data structures used at different compilation levels. It embraces hundreds of classes with some common parts, which can be generated automatically. Support for garbage collection, dumping, visitors etc. becomes nearly transparent this way.

The generator is written in the XSL Transformations language (XSLT) and takes XML files as an input. Result is a C++ code.

2.1 Data Formats and Processing

There are two types of source XML files: Lestes Structures Description (LSD, stored in `*.lsd` files) and Lestes Visitors Description (LVD in `*.lvd` files). LSD is used to describe classes, their members, inheritance and special properties. LVD is used by the visitors-generating mechanism, see the Visitors 2.14 section below.

To process LSD and LVD files a XSLT processor is needed, such as **xsltproc**. The process is fully supported by Lestes build system.

There are several stylesheets (`*.xslt`) in `util/lsg` directory, the most important ones are `lsd2hh`, `lsd2cc`, `lvd2lsd`, and `lvd2cc`. The `lsd2*` transformations take an `X.lsd` file as input and create one `X.g.cc` and one `X.g.hh` file (the `g` stands for "generated"). If there are methods to be implemented by hand, their code should be placed in hand-written `X.cc` file. The `lvd2*` stylesheets take `Y.lvd` file to generate `Y.v.lsd` and `Y.v.cc` (the `v` stands for "visitors"), see the Visitors 2.14 section below.

2.2 XML Validation

Validation, the process of checking whether an LSD or LVD document is correct, is not possible in this version. An XML Schema for LSD and LVD should been written in the future, but it is not necessary for using the framework. The generator itself does performs only a few checks and if a bug appears in an input file, the generated code may be wrong, or the XSLT processor may produce an error message.

2.3 Comments

Anywhere in LSD or LVD file a `<comment>` element may occur. Its content is ignored by generator. An alternative is a XML-like comment:

```
<!-- This text is ignored by XML parser. -->
```

The LSG-specific comments are recomended to be used because they can be extracted from the documents and used some way in the future.

2.4 LSD File Header

2.4.1 Root Element and XML Namespaces

It is recommended to read `util/lsg/example.lsd` and other examples in the directory when creating LSD files. The first two lines and the root element name are common for all LSD files.

```
<?xml version="1.0" encoding="UTF-8"?>
<lsd xmlns="http://lestes.jikos.cz/schemas/lsd"
xmlns:h="http://www.w3.org/TR/REC-html40">
...
</lsd>
```

The `xmlns:h` attribute can be omitted if we do not use HTML tags in Doxygen comments 2.9.

2.4.2 LSD Prologue

The header part of LSD contains the following elements in this order:

```
<dox>
<file-name>
<packages>
<imports>
<implementation-imports>
<hh-header>
<hh-footer>
<cc-header>
<cc-footer>
<include>
<default-base>
<default-collection>
<default-check>
```

Any of these elements may be omitted or left empty if we don't need it. Only the `<file-name>` element is required and contains the name of the LSD file without extension. It must not contain any peculiar characters because it's used as a part of an identifier in output C++ code. Dashes and dots are allowed.

The `<dox>` part contains `<bri>` and `<det>` elements, which mark doxygen documentation for files generated from this LSD (brief and detailed). There may be two `<dox>` elements with `file="hh"` and `file="cc"` attributes, or only one with `file="both"`. The attribute controls whether the documentation is copied into the header file, the implementation file, or both, respectively. See also the Doxygen 2.9 section below.

The `<imports>` element can contain any number of `<i>` elements, each of them containing a path to a file. Such `<i>` element corresponds to one `#include` line in output `*.g.hh` header file. The file name is always bounded by angle brackets (<,>), because relative including with quotes is not used in Lestes project.

The `<implementation-imports>` part is analogous to the previous one, but the `#include` lines are generated only into `*.g.cc` implementation file instead of the header file.

The `<packages>` element is used to satisfy the namespace policy in Lestes project. It contains package names enclosed in `<p>` elements. All classes generated from this file are placed into one namespace determined by these package names. See also the Namespaces section 2.13 below.

The following elements:

```
<hh-header>
<hh-footer>
<cc-header>
<cc-footer>
```

may be used to add hand-written code into the generated C++ files. Generally, this is discouraged and should only be used as a last resort.

The <include> element may occur arbitrarily many times and has one required attribute - href. It contains a path to another LSD file. Its purpose isn't a true inclusion such as in C++. The only purpose of these links is to make other generated classes in the same namespace visible for generator, if they are bases of classes contained in the current file. A generated constructor or factory method 2.12 must initialize all data members of current class, including the inherited ones. Therefore parents must be accessible for generator. Note that the search for bases is done recursively, so you should include only the LSD file with the nearest descendant. The infinite inclusion loops are not detected, so users must avoid it. The generator also assumes that a class can be reached through exactly one path of includes. However it is easy to obey these restrictions if the <include> hierarchy is a tree corresponding to the natural ISA hierarchy between classes.

The <default-base> element has a type attribute. It denotes a base class for all classes with the base attribute omitted.

The <default-collection> element has a kind attribute. It denotes a kind for all collection members with kind attribute omitted.

The <default-check> element may contain a name of a macro to be used as default for this LSD file instead of checked(arg). Function of this macro is described below.

Some helper declarations follow the header part: <forward-class>, <using-class>, and <foreign-class>. Their usage is described below in sections Garbage collection 2.10, and Namespaces 2.13.

The main part of an LSD file contains declarations of classes, enumerations, and types.

2.5 Class Description

Ordering of classes may be nearly arbitrary, since a forward declaration is generated for each <class>. The only rule is that a class must precede any and all classes that are derived from it.

Element <class> has the following attributes:

- name (required): An identifier for the class.
- base: Name of a base class. Can be omitted if a default base is declared in header part of the document. Multiple and virtual inheritance is not supported.
- abstract: Contains "yes" or "no", default is "no". Abstract classes can't be in-

stantiated and lack factory methods. A class containing an abstract method must be explicitly marked as abstract.

Element <class> may have the following content:

- <field>: Describes a data member. Visibility of each data member is private. set(), and get() methods are public. <field> attributes are:
  - name (required): Field identifier.
  - type (required): Data type. If the member is a pointer to a class, there should be simply the identifier of the class. The generator will automatically create a smart pointer and treat it correctly. See also Garbage collection 2.10 below.
  - set: This describes method that is used to set a field's value. May contain "special", "none", or "". Default is "". By default, the set() method is generated and trivially returns the field value. "none" means that the member lacks set() method, "special" means that the set() method is present but hand-written. See also Class data access 2.6.
  - get: Describes the get() method in analogy to set().
  - specifier: Contains a specifier for the field, allowed values are "static" or "", default is "". A static member is treated specially as C++ requires (it's declaration is placed at the bottom of .g.cc file).
  - init: Contains an initializer for the field. May contain simply a value or "" or "void", default is "void". "void" means that the member is always initialized through a parameter of a factory method. Otherwise, the attribute value is used as an initializer. See also Creating an instance 2.12.
  - check: Value of each field is assertion-checked with a special macro before class construction (see 2.12). The default name of the macro is "checked" or may be specified in <default-check> element (in LSD header part 2.4). Setting the check attribute allows you to call a different macro.
- <collection>: Describes a collection of values, similarly to the <field> element, using the same attributes.
  - One difference is in the init attribute, which is restricted to two cases.
    - Leaving out the init attribute entirely, or setting it to "void" results in a collection initialized by a factory method parameter.
    - init="" means initialization as an empty collection.
  - Attribute kind may be "list", "vector", "set", or "map". It's default

value is specified in the top-level element <default-collection>. Lestes STL wrappers are always used to ensure garbage collection. The map collection has two additional attributes:

  * key (required): Contains a data type used as a key in the map.

  * comparator: Describes a special comparator for the keys. For example: comparator="::std::greater<int>"

- <method>: Describes a method that will be written by hand. The attributes are:
  - name (required): Method identifier.
  - type (required): Return type. (See Garbage collection 2.10.)
  - qualifier: A qualifier for the method. For example "virtual".
  - specifier: A specifier for the method. For example "static" or "abstract".

An abstract method must have qualifier="virtual" and specifier="abstract". Class containing such method must have the abstract set to "yes" in its <class> element. Factory methods 2.12.2 are not generated for abstract classes.

Parameters of the method may be given as a sequence of <param> elements into the <method> element. Each <param> has one required attribute type; the name attribute is optional.

Additionally, these elements can be used inside <class>:

- <typedef>: Describes a type definition used in this class. It's visibility is public. See also Type description 2.8 below.
- <enum>: Describes an enumeration type used in this class. It's visibility is public. See also Enumeration description 2.7 below.
- <visitor>: See Visitors 2.14.

2.6 Class Data Access

All data members of generated classes are **protected**. By default two special methods are generated to allow access to the fields of the class. Visibility of these methods is **public**. Each of them may be user-defined or missing. For the field x of a type T the methods are:

- T x_get() - Returns the value of the field.

- `void x_set(T a)` - Assigns a given value to the field.

2.7 Enumeration Description

On the top level or in a class definition, an `<enum>` element can appear. It defines a new enumeration type and has one required attribute `name` with the identifier for the new type. `<enum>` contains a sequence of `<e>` elements representing enumerators, each of them having a required attribute `n` (stands for "name") with enumerator identifier. Optional `<e>` attribute `val` associates the enumerator with a constant expression. See also an example in the C++ Standard **[7.2/2]**.

2.8 Type Description

On the top level or in a class definition, a `<typedef>` element can appear. It defines a new type. It has two required attributes: `name` with the new identifier and `type` with a type specification.

2.9 Doxygen

Doxygen is a program used to generate documentation directly from source files. Hand-written parts of such documentation are placed in the source files as special comments describing the corresponding objects. Lestes Structures Generator allows users to insert such comments into the generated C++ files.

A `<dox>` element used to add a Doxygen-like comment can occur almost anywhere in the LSD file. It contains two elements: `<bri>` with a brief description and `<det>` with a detailed description. The `<det>` element can be omitted, `<bri>` is required. The `<dox>` element has no attributes (except the top-level `<dox>`, see LSD header part in 2.4).

```
<dox>
        <bri>Brief comment</bri>
        <det>Detailed description</det>
</dox>
```

The comment describes the element it occurs in. These LSD elements may contain <dox>:

```
<typedef>
<enum>
<class>
```

```
<field>
<collection>
<method>
```

The `method/dox` element should not contain `<det>` since the detailed description of a method should be placed near it's implementation (in hand-written `.cc` file).

The abstract visitor classes written by the LSG may be commented as well. In LVD files the `<dox>` element may occur in these elements:

```
/lvd
/lvd/visitors/v
/lvd/visitors/v/special
```

It is used the same way as in LSD files.

2.10 Garbage Collection Support

The C++ language lacks automatic garbage collection and forces programmers to treat each instance of a class to be deallocated. However, this is very difficult to achieve in a project like Lestes that uses about one thousand of different classes and huge heterogeneous data structures. It was found that a garbage collector is necessary. Its interface and usage is described in Garbage collector section of the Lestes documentation. Users have to use two types of smart pointers and write a marking routine for each class. The generator strongly helps with performing these two duties.

2.10.1 Marking Routine

The `gc_mark()` method is generated fully automatically for all LSD classes (including abstract ones). The visitor classes generated from LVD files are also made via LSD (see Visitors 2.14), so they have their marking routine generated in the same way.

It's visibility is always **protected** and the prototype is common for all classes:

```
virtual void gc_mark();
```

The routine simply calls `gc_mark()` routines of all garbage-collectible members specified in this class. Then, control is passed to `gc_mark()` in base class where inherited members are handled. And there is no need to iterate through collections of garbage-collectible types, since the STL wrappers do that themselves.

2.10.2 Smart Pointers

The garbage collector forces users to use two kinds of pointers represented by `ptr<T>` and `srp<T>` templates (both point to a type `T`). The `srp` pointer must be used for members of a structure, class, or collection. The `ptr` should be used pointer everywhere else (return values, local variables, method arguments, etc.). The generated code obeys these rules.

2.10.3 Garbage-collectible Types

Garbage-collectible types must be handled by smart pointers, non-collectible types are handled by value. The rule is that all classes are derived from `::lestes::std::object` and are garbage-collectible and all simple types are not collectible. (A few exceptions exist but have no effect on the generator.) The generator tries to decide automatically whether an identifier denotes a class or a simple type, but sometimes it needs a hint.

The basic assumption is that unknown types are not collected. Types like `lint`, `ucn_string`, `character` are not described in any LSD file and the generator doesn't make pointers on them. The user must let the generator know about each violation of this assumption. Note that `<include>` tags are not used when searching for "known" classes, their purpose is different. For classes used in an LSD file (as a base, as a member, as a method argument, as a visitor type, or so) but declared somewhere else, there must be one of these three constructs:

```
<forward-class name="thomas">
<using-class name="alva" packages="::lestes::std::"/>
<foreign-class name="edison">
       <p>lestes</p>
       <p>std</p>
</foreign-class>
```

The naming and syntax of these constructs has historical reasons and shouldn't be taken seriously and may be a subject to change in the future.

The `using-class` element declares that the name in question is garbage-collectible. It is usually used to inform the generator that a typedef is to be handled using smart pointers. If it lies in a different namespace, the attribute `packages` should be filled, otherwise it may be omitted. No special C++ text is generated, except for the namespace qualification.

The `forward-class` element declares that a name denotes a class (that is not a template, nor a typedef). Compared to `using-class`, the difference is that `forward-class` causes a C++ forward (incomplete) declaration to be generated at the beginning of the `.g.hh` file. For example:

```
class thomas;
```

The class must be in the same namespace as the LSD file contents. This is needed whenever the name is used as a base class, or type of a field, or a parameter. Using `forward-class` has the advantage that the header file containing actual declaration of the class does not have to be in the `imports` part. The only case where it is not sufficient to put it in `implementation-imports` is when the name is used a a base class. This is mandated by C++, as the base class must be a complete type when declaring a class derived from it. On the other hand, when declaring a member of smart pointer to `T`, the type `T` can be incomplete.

The `foreign-class` element does the same as `forward-class` but the class lives in a different namespace. The sequence of `<p>` elements denotes the namespace. A forward declaration is generated with respect to the namespace policy and is placed out of the binding `package/end_package` braces of the generated file. Example:

```
package(lestes);
package(std);
class edison;
end_package(std);
end_package(lestes);
```

2.11 Dumping

LSG inserts code and data into each class for purposes of the dumper. Static member `reflection` contains meta data about the class - it's name and it's members names. The meta data is created on demand and only once by the `reflection_get()` method. The dumper also needs the actual values of the members; `field_values_get()` returns such list of values.

2.12 Creating a Class Instance

2.12.1 Constructor

Each generated class has just one constructor and it's visibility is **protected**. It has one argument for each member of the class including inherited ones. The values passed to the constructor are used to initialize all the members. If some of the data is passed by pointer, the pointer is checked for non-null-ness. Passing a `NULL` pointer causes an abnormal program termination. The goal is to avoid creation of incomplete classes that could cause bugs that would be hard to find.

2.12.2 Factory Method

Class instances in Lestes are almost every time created by so-called factory methods. Factory method is a static method called `create()` that allocates and initializes a new instance. For the purposes of initialization (see above) the factory method needs a value for each member of the class. One way to do it is to have one parameter of the factory method for each member (as it is in the case of constructor). This case is always enabled, since such a `create()` method is in each generated class that is not abstract. But we often don't want to fill all the members by hand when creating an object. Some fields may have initial values specified in the `init` attribute. In such a case the `create` method is overloaded and may have a different number of parameters. The second one lacks parameters for those fields that have the `init` attribute specified. If each member has an initial value, there will be a `create()` method without parameters.

2.13 Namespaces

The `package` and `end_package` macros behave like left and right braces. The main content of a LSD file is placed into one namespace in both the `.g.hh` and `.g.cc` files, including the contents of `hh-header`, `hh-footer`, `cc-header`, and `cc-footer` elements. A class from a different namespace must be declared with `using-class` or `foreign-class` and then the namespace prefix is automatically placed before each occurrence of it's identifier. For example the `object` base class should always have a declaration like this[1]:

```
<using-class name="object" packages="::lestes::std::" />
```

Only the included files and forward declarations of foreign classes are placed outside the main package braces.

2.14 Visitors

2.14.1 Basics

Visitor is a design pattern, see Gamma et al., Design patterns. It's a masterpiece of object-oriented design in the Lestes project. The goal is to separate actions above the complex structure from classes that contain the data. Visitor is a class representing an action that

[1] Note that `using-class` is used, although `forward-class` is more suitable here. However, as complete declaration of the `object` class is included in all generated files, we can use the more compact form.

takes place on classes from a specific set. The set is determined by a name of one class and it includes the class together with all non-abstract classes derived from it. Abstract classes cannot be instantiated and therefore cannot be visited. The root class has a special note in it's LSD description. For example:

```
<class name="as_expression" abstract="yes" base="as_base">
        <!-- some content -->
        <visitor name="as_expr_visitor" type="void"/>
</class>
```

This defines a visitor class `as_expr_visitor` that can perform an action on any non-abstract class derived from `as_expression`. The class itself is abstract and cannot be visited. The action represented by a visitor has a result. In the above example the type is `void`.

A visitor is used every time when an action implementation depends on the class that it's performed on (the **visitee**).

2.14.2 Abstract and Concrete Visitors

The LSG is able to make a visitor class specific for a sub-tree of the class hierarchy. Such a class is always abstract, because the generator has no information about the action implementation. Note that there can be more than one action performed on the same set of classes. It would be nice to have a reusable abstract class to visit a set of classes and the concrete visitors (actions) would be derived from it. Different actions have different types of results. These have to be declared as different abstract visitor classes. This is the reason why we have later decided to prefer visitors with return type `void`. If there is a need to return a value, it can be stored in a special member in the concrete visitor class.

2.14.3 What Does the Generator Produce

Each class derived from a class with visitor has a generated method with the following prototype:

```
virtual T accept_V( ptr< V >);
```

where `V` is the visitor's name and `T` is it's return type. If the class is declared as abstract, the `accept()` method is also abstract. Otherwise an implementation is generated into the `.g.cc` file. It simply calls corresponding `visit_A()` method of the visitor class (see below).

The abstract visitor class is named as specified and it is derived from `::lestes::std::visitor_base`. It contains a public abstract method for each visitee.

```
virtual T visit_A( ptr< A >) abstract;
```

where `T` is the action result type and `A` is the visitee. The `visit_A()` method represents an action implementation on a specific class.

LSG can simplify other aspects of visitors implementation, see 2.14.6 on the following page.

2.14.4 What Does the LSG User Have to Do

Simply said, the user has to derive his concrete visitor from the generated abstract class and implement all the `visit*` abstract methods. If some data storage is needed for the implementation, it can be added to the concrete visitor class. (For example to store a result or whatever.) To perform the action on a given object, one has to instantiate the concrete visitor and call the visitee's `accept` method with pointer to your visitor.

Note that the abstract classes are generated into a *.v.lsd file in the LSD format (by `lvd2lsd.xslt` stylesheet). This allows users to derive their concrete classes via LSD with all the comfort (automatic dump support, marking routines, safe constructors, factory methods etc.).

2.14.5 LVD Files

The abstract visitor classes are generated by a mechanism completely different from LSD processing. A special XML description of what to do is needed. The header part of an LVD file is very similar to LSD, except for the different XML namespace.

```
<?xml version="1.0" encoding="UTF-8"?>
<lvd xmlns="http://lestes.jikos.cz/schemas/lvd">
        <dox><bri>Some abstract visitor classes</bri></dox>
        <file-name>zzz</file-name>
        <imports> . . . </imports>
        <implementation-imports> . . . </implementation-imports>
        <packages>
                <p>foo</p>
                <p>bar</p>
        </packages>
        <hh-header/>
        <hh-footer/>
```

A sequence of <uses> elements follows such a header. The element has a href attribute with a path to an LSD file, where some visitees are defined. The main part is in the <visitors> element that simply specifies names of the visitors.

```
<visitors>
        <v name="quo"/>
        <v name="vadis"/>
        <v name="domine"/>
</visitors>
```

All other necessary information is automatically searched in the LSD files via <uses>. Usage of Doxygen comments is described in the above section 2.9.

2.14.6 Implementing Visitors—The Inheritance Tree Cuts

Some visitors may visit a lot of classes. In practice, the action implementation is often the same for many of them and it would be annoying to write all the visit methods by hand. Any small change in implementation would force user to type it repeatedly, introducing a possible source of bugs.

Let's see a real-life example. Class ss_type_visitor is an abstract visitor for ss_type descendants. The action depends on a specific class. It is different for ss_array, ss_enum, ss_function, ss_builtin_type, ss_struct_base, and so on. But it doesn't depend on a concrete builtin type (15 different non-abstract classes!). And it's also the same for all descendants of ss_struct_base. Therefore, we would like to implement it just once for all ss_builtin_type descendants. And the second time for ss_struct_base descendants. The LSG offers such a feature.

```
<v name="ss_type_visitor" >
        <special name="ss_type2info_base">
                <cut at="ss_struct_base" method="process_ss_struct_base"/>
                <cut at="ss_builtin_type" method="process_ss_builtin_type"
        </special>
</v>
```

In this example, the LSG will generate an abstract class ss_type_visitor as usual. In addition, there will be a class ss_type2info_base derived from ss_type_visitor. All its visit methods for ss_builtin_type descendants are automatically implemented as a call to a process_ss_builtin_base method. For descendants of ss_struct_base the method process_ss_struct_base is called. Both process* methods are automatically added as abstract, since the generator doesn't know about the true action. Therefore the class ss_type2info_base is also abstract. The concrete visitor is derived from it and implements the two process* methods and all

the `visit*` methods that remain. Such a concrete visitor is `ss_type2info`. Its hand-written implementation is in `ss_type2info.cc` file and it contains just one method for each nearest descendant of `ss_type`, all in all 13 methods. That was our goal. Note that `ss_type2info` visits 28 non-abstract classes. And note that there is no problem to have more than one `<special>` in a specific abstract visitor.

One additional feature of the generator is used in the above example that was not yet mentioned. If a class implements some inherited abstract methods, C++ requires explicit declaration of all such methods in the class declaration (header file). It's easy to accidentally omit one of the declarations. Then the class remains abstract and the compilation fails at its instantiation. In the even worse case, some declaration may be extra and there remains an extra implementation that is never called. Both these problems may be prevented by using automatic generation of the declarations. Note that the class `ss_type2info` has a special line in it's LSD definition.

```
<declare-abstract-methods prefix="visit_" skip-defined="yes" />
```

It generates declarations of all abstract methods in the class that start with `visit_` prefix. Instead of the prefix, a suffix may be specified with `suffix="fff"` or an exact match may be required with `exact="identifier"`. The `skip-defined` attribute may be optionally used to skip the methods that have been defined previously. Omitting all of these attributes causes all abstract methods to be declared and forced to be implemented.

Chapter 3

Conclusion, Future Work

It would be difficult to design such code generator as a truly generic framework to fit the needs of different projects which may have quite different requirements on its functionalities. Our code generator is able to make some routine work, which would be difficult to maintain manually and would be a possible source of errors. Especially, thanks to the XPath `document()` function, it is able to do the search through the class hierarchy and generate headers of the methods to be implemented in visitor classes. It represents a relatively comfortable environment to reuse and redesign the project code which includes more than one thousand classes. Naturally, a cost had to be paid for that comfort - hours spent developing and redesigning the code generator.

Bibliography

[1] Andrei Alexandrescu. Modern C++ Design: Generic Programming and Design Patterns Applied, 2001.

[2] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design patterns - Elements of Reusable Object-Oriented Software, 1995.

[3] Lestes Team. Lestes C++ Compiler Project. `http://lestes.jikos.cz/`, 2005.