Part One: Traditional Patterns in XML applications

by Fabio Arciniegas A.

Adequate documentation of the experience gained during the development of XML-based systems is a prerequisite for XML's success as a widely used technology. *Design patterns* have proved to be a very good technique for transmitting, and to some extent formalizing, knowledge about recurring problems and solutions in the software development process.

This article, the first of two articles on XML and design patterns, is focused on the applicability of some well-known design patterns to XML-specific contexts.

This article assumes some basic knowledge about XML processing. Also, basic knowledge about UML class diagrams will be useful (see our basic UML class diagram guide).

Contents

•Design Patterns in XML Applications
•Command Pattern
•Flyweight Pattern
•Wrapper Pattern
•Iterator Pattern
•Bibliography

What are patterns?

Patterns are an effective way to transmit experience about recurrent problems. A pattern is a named, reusable solution to a recurrent problem in a particular context.

Patterns are not miraculous recipes that will work in every scenario, but they do convey important knowledge, a standard solution, and a common language about a recurrent problem. All this makes them powerful design tools.

Since common problems with (often) common solutions appear in many scenarios, patterns are now used in almost every part of development: there are process patterns, architectural patterns, implementation patterns, testing patterns, etc. However, one particular kind of pattern has received special attention from the development community: *design patterns*. Design patterns are a powerful reuse mechanism, and a way to talk about design decisions that actually work.

The expression *XML patterns* may be used to denote two kinds of patterns: (1) design patterns specifically treating XML-related problems, and (2) information structuring patterns for the design of DTDs, schemas, etc.

XML patterns will be discussed more fully in the next article. Here we will focus on the applicability of *traditional* design patterns to the design of XML applications.

Traditional design patterns are often classified in categories. One common set of categories is *structural* patterns and *behavioral* patterns. In this article we will explore the applicability of patterns in each of these categories to XML problems.

The patterns we will discuss are: *Command pattern*, *Flyweight pattern*, *Wrapper pattern*, and *Iterator pattern*. The choice of patterns for this article notwithstanding, any other pattern can be applied to the design of XML applications.

Choosing the right patterns to present has not been easy. I have tried to maintain a balance between the different options, thus there are two structural and two behavioral patterns; two DOM-oriented and two event-based application discussions; and two of the patterns are illustrated using C++ and two using Java.

Command Pattern

Synopsis

Command is a behavioral pattern used to encapsulate actions in objects. This is highly useful when you want to keep track of changes made to a model, for example in supporting multi-level "do/undo."

Structure

The following is a class diagram of the command pattern. Slightly different versions of this pattern can be found in the literature, however, I chose to present it in this fashion for clarity.

Figure 1. Command Pattern Structure

XML Context

Suppose you are building an application that uses the DOM representation of an XML document as its basic data—say a component for displaying vector graphics, or a simple shopping list manager.
The user of your program will perform many operations, like deletions and additions. Since you are using the DOM as your underlying model, these changes will sooner or later translate into calls to *removeChild* and other DOM-specific calls. However, depending on how you structure your program, these changes can become either a hard-to-maintain, hard-to-extend mess, or an organized, extensible solution. Here is where the command pattern can help.
Let's take the shopping list editor as an example. The user wants to delete, add, and annotate the shopping list, among other operations. You use a GUI, so one option would be to hard-code your menu widgets' member calls to DOM-specific methods. For example, when the user selects the menu item "*Insert*," call *insertChild*. This has a number of "advantages":

- Such code is fast to write.
- Most GUI builders will "lead" you towards this.
- It can be soft in terms of resource consumption.

It seems like it could be a real choice, but now you want to add undo/redo support to your program, and serious problems regarding this option become apparent:

- There seems no easy way to maintain your do/undo list: either you change all your hardcoded widget events to call both the DOM methods and log to some list, or you change your DOM representation to somehow log the changes performed (!)
- Even if you managed to successfully implement the do/undo lists from the hardcoded widget calls, you would be replicating that logic many times, which is hard to maintain and error-prone.
- There is no clear indication as to which part of your program will manage the undo logic and how it will do it.

The solution that the command pattern proposes is to encapsulate the changes to the DOM into objects, *command objects*, each capable of doing (and undoing) a particular action. The collection of command objects will be managed by a certain *command manager*, capable of holding the queue of executed commands, so the user may undo/redo them.

Example

This example reflects a very common approach to DOM processing using the command pattern. If you will be writing applications using DOM as the underlying data structure representation, you are very likely to find this approach useful.

Figure 2. Command Pattern Structure

The figure shows the structure of a typical DOM-oriented application using the command pattern for its message passing. The following is the header file for the base class *AbstractCommand*, which is the foundation of the example. Please refer to command.zip for the complete example code.

Figure 3. Command Example Header

```
/**
   AbstractCommand is the base class for all commands.
   It provides do/undo operations as well as getDescription and
   getState operations for the easy tracking of the executed commands.
   (quite useful when keeping a menu of last performed operations).
 */

class AbstractCommand
```

```
{
 public:
  /**@name Comparison operators
   * The comparison operators in the base AbstractCommand are
   * provided in order to keep STL usability in the CommandManager
   */
  //@{

  /// equality operator
  virtual int operator==();

  /// unequality operator
  virtual int operator!=();

  /// increment operator
  virtual void  operator++();
  //@}

  /**@name Do / Undo methods
   */
  //@{

  /// Pure virtual operation that in child classes encapsulates the
  logic of the change
  virtual Notification do() = 0;

  /// Pure virtual operation that in child classes encapsulates the
  logic of undoing a change
  virtual Notification undo() = 0;

  /** Pure virtual operation that in child classes returns the description
  of the operation
   * (particularly useful for undo/redo lists presented to the user)
   */
  virtual string getDescription() = 0;
  //@}
};
```

Note that even when this example is written in C++, the main principles (and even the code) can be ported to other languages with ease.

Summary of Common XML Uses

My personal experience shows that the command pattern is especially useful in XML applications when:

- You have a DOM-based application and need to keep track of the changes made to the data model.
- You have a DOM-based application and need to keep open the possibility for easy and clean extension of the available commands that can be performed on the data model.

In this section we analyzed *Command*, a behavioral pattern, in object-model-based XML applications. In the next section, we will see a structural pattern in event-based XML applications, *Flyweight*.

Flyweight Pattern

Synopsis

Flyweight is a structural pattern used to support a large number of small objects efficiently. Several instances of an object may share some properties: Flyweight factors these common properties into a single object, thus saving considerable space and time otherwise consumed by the creation and maintenance of duplicate instances.

Structure

Figure 4. Flyweight Pattern Structure

XML Context

One of the biggest problems with keeping the DOM representation of the document, instead of constructing your own objects from the output of SAX (or another event-oriented interface), is the size of the representation. In this discussion we assume not only that you want to roll your own domain-specific objects, but that you want them to be as space-efficient as possible.

Suppose you are writing a SAX-based application that constructs CD objects from a file called `musicCollection.xml`. At the end of parsing you might want a collection of CD objects to be created. Those objects may look like:

Figure 5. Initial CD structure

As you probably already noticed, all the information about the *artist* (in this example we use only one, for simplicity) may be replicated many times. (Notice too that this artist information is unlikely to change over time.) This is a clear candidate for factorization into what we'll call a flyweight: a fine-grained object that encapsulates information (usually immutable) shared by many other objects.

Remember that CD objects should be constructed from an XML file that might look somewhat like this:

```
        <?xml version="1.0"?>
        <collection>
           <cd> <!-- This is quite simplistic, better XML
              representations could have been
                     chosen, but it only aims at
                     illustrating the pattern -->
     <title>Another Green World</title>
     <year>1978</year>
     <artist>Eno, Brian</artist>
   </cd>
           <cd>
     <title>Greatest Hits</title>
     <year>1950</year>
     <artist>Holiday, Billie</artist>
   </cd>
           <cd>
     <title>Taking Tiger Mountain (by strategy)</title>
     <year>1977</year>
     <artist>Eno, Brian</artist>
   </cd>
        </collection>
```

You decide to use Java and a SAX parser to do the job. Now you must construct a set of SAX handlers capable of creating CD objects with flyweight artists. This will be the subject of our example.

Example

The basic logic for the SAX handler is simple:

- Whenever a CD open tag is found, create a new CD object.
- Whenever `title` or `year` elements are found, enter them in the current CD.
- Whenever an `artist` element is found, *ask the artist factory to create it*. This is fundamental to the problem: the CD object does not know if it is sharing this object with others; only the factory keeps track of what has been created.

The following code illustrates a simple factory for the extrinsic objects, and the output produced by the example program if run with the above XML file.

Flyweight Example: Factory

```
// Simple Flyweight factory for Artist classes (Artist is the extrinsic,
// flyweight class. CD is the client)

import java.util.Hashtable;
import java.lang.String;

public class ArtistFactory {
    // Whenever a client needs an artist, it calls this method. The client
    // doesn't know/care whether the Artist is new or not.

    Artist getArtist(String key){
 Artist result;
 result = (Artist)pool.get(key);
 if(result == null) {
     result = new Artist(key);
     pool.put(key,result);
     System.out.println("Artist: " +key + " created");
 }
 else
     System.out.println("Artist: " +key + " reused");
 return result;
    }
    Hashtable pool = new Hashtable();
}
```

Flyweight Example: Output

```
$ java -Dorg.xml.sax.parser=com.ibm.xml.parser.SAXDriver \
FlyweightDemo music.xml
Artist: Eno, Brian created
Artist: Holiday, Billie created
Artist: Eno, Brian reused
Artist: Eno, Brian reused
Artist: Eno, Brian reused
```

For the complete code, please download flyweight.zip

At the end of the parsing, the actual object structure will be:

Figure 6. Flyweight Object Diagram - Example

Summary of Common XML Uses

The flyweight pattern is useful in XML applications when:

- You have a domain-specific representation of your document, and you want to keep it as small as possible by taking advantage of shared information among objects.

This is often the case!
In this section we analyzed *Flyweight*, a structural pattern useful in event-based XML applications. In the next section, we will examine *Wrapper*, another structural pattern, also in an event-based context.

Wrapper Pattern

Synopsis
Wrapper is a structural pattern used to allow an existing piece of software to interact in an environment different from its originally intended one. Wrapper is very similar to the famous *Adapter* pattern. The difference between the patterns is not predominantly structural, but rather in their intentions: Adapter seeks to make an existing object work with other *known* objects that expect something, while Wrapper is focused on providing a different interface (without knowing in advance its clients) and solving platform/language issues.

Structure

Figure 7. Wrapper Pattern Structure

XML Context

Wrapper is one of the most easily identifiable patterns in the XML world. Even though its explanation is very simple, it is worth mentioning because of its frequency.
A wrapper pattern is used every time an existing parser is adapted to work in another language. A new interface that uses constructs of the new language is defined, yet little or no change in the functionality takes place.
One common source of wrappers in XML is James Clark's expat. Wrappers for expat (developed in C) have been written in numerous languages. Several wrappers are available for C++ (including expatpp), Perl, and other languages.
In the example, we will look at the original C interface of expat, and the C++ wrapper that adapts it for object-oriented manipulation. See also the end of the example section for pointers to complete wrappers of expat.

Example

Expat works by calling functions, called *handlers*, when certain events occur (for more about expat, refer to Clark Cooper's XML.com article on expat). The following is a small part of the original expat interface, defining the type of a handler, and a function to register handlers for listening to "start element" and "end element" events:
Original expat interface

```
        ...
  /* atts is array of name/value pairs, terminated by 0;
     names and values are 0 terminated. */
        typedef void (*XML_StartElementHandler)(void *userData,
    const XML_Char *name,
    const XML_Char **atts);
...
        void XMLPARSEAPI
```

```
  XML_SetElementHandler(XML_Parser parser,
        XML_StartElementHandler start,
        XML_EndElementHandler end);
```

Expat can be used directly in a C++ project, however, several wrappers have been devised to take advantage of C++ syntax. A good example is Andy Dent's *expatpp*.
All expatpp does is simplify the interface for C++ programmers by wrapping an expat parser in a class:
Simplification with expatpp

```
        class expatpp {
public:
 expatpp();
 ~expatpp();

 operator XML_Parser() const;

        // overrideable callbacks
 virtual void startElement(const XML_Char* name, const XML_Char** atts);
 virtual void endElement(const XML_Char* name);
 virtual void charData(const XML_Char *s, int len);
 virtual void processingInstruction(const XML_Char*
             target, const XML_Char* data);

 ...
```

In order to adapt the expat interface for the new object-oriented calls, the constructor binds the expat callbacks to the corresponding method. Thus, all you have to do in order to handle a particular kind of event is to override the method in a subclass. If you have never worked with expat, this could be a little confusing, but don't worry. The key to understanding it is to look at the code itself: wrapper.zip

Summary of Common XML Uses

The wrapper pattern is useful in XML applications when:

- You want to reuse a piece of XML software in an environment different from the one initially intended.

In this section we reviewed Wrapper, a structural pattern useful for adapting XML applications and processors. In the next and final section, we will see Iterator, a behavioral pattern that is very useful in object-model-based contexts.

Iterator Pattern

Synopsis

Iterator is a behavioral pattern used to access the elements of an aggregate sequentially, without exposing the aggregate's underlying representation. It is particularly useful when you want to encapsulate special logic for the

traversal of a structure like a DOM tree.

Figure 8. Iterator Pattern Structure

XML Context

Suppose you are writing a tool that uses the DOM as its internal data representation mechanism. Presumably, there are a lot of actions you want to perform on the members of this collection of elements: search for a particular element, delete all elements with a given name, print elements of certain type, etc.
Since you have read the *command pattern* section, you decide to implement those actions as Commands, so now you have a nice, extensible way of working with those elements:

```
        applyToAll(AbstractCommand action) {
     // traverse the whole tree applying action to each
     // node
  }
```

This is good. However, you start to notice different traversals can work better in some cases, and some actions only need to work on certain kind of objects. So you start wondering about a way to isolate the traversal logic from the rest of the program.
The solution is in the *iterator pattern*. Using the iterator pattern you can create a parametric method applyAll that expects not only a generic action, but a generic iterator:

```
        applyToAll(AbstractCommand action, AbstractIterator iterator) {
    for(iterator.reset(); !iterator.atEnd(); iterator.next()) {
       action.target(iterator.value());
       action.do();
    }
  }
```

Now you can invent iterators for all kinds of traversals: pre-order, post-order, in-order, pre-order only over text elements, etc., without having to change a single line of your (already compact and elegant!) method.

Example

The iterator presented traverses the collection (the DOM) by levels, printing first all CD elements, then all title, year, artist, and finally all the text elements. Here is the code for such an iterator:

Iterator Sample code

```
/**
 ***************************************************************************
 * Name: LevelIterator
 * Description: This iterator traverses the tree by levels.
 *              Note that it could be replaced in the main program for
 *              any other iterator conforming with AbstractIteratorIF,
 *              without changing anything in the main program logic.
 ***************************************************************************
 */
import org.w3c.dom.*;
import java.util.Vector;

public class LevelIterator implements AbstractIteratorIF {
    public boolean end() {
 return (aux.size() == 0);
    }
    public void next() {
 if(aux.size() > 0) {
     current = (Node) aux.elementAt(0); //first get the new next element
     aux.removeElementAt(0);
 }
 // now add all of its children to the end... a typical
 // level traversal.
 if (current.hasChildNodes()) {
             NodeList nl = current.getChildNodes();
             int size = nl.getLength();
             for (int i = 0; i < size; i++) {
                 aux.addElement(nl.item(i));
              }
         }
    }
```

```
    public Node getValue() {
 return current;
    }
    public LevelIterator(Node c) {
 current = c;
 aux.addElement(current);
    }
    Node current;
    Vector aux = new Vector(); //auxiliar vector for the sublevels
}
```

This is the output of the IteratorDemo program that uses the previous iterator to walk the `music.xml` example from the Flyweight section.

Iterator Sample Output

```
--
Node Name: collection
NodeValue: null
--
Node Name: #text
NodeValue:
--
Node Name: cd
NodeValue: null
--
Node Name: cd
NodeValue: null
--

...

--
Node Name: #text
NodeValue: Eno, Brian
--
Node Name: #text
NodeValue: The Drop
--
Node Name: #text
NodeValue: 1999
```

Please refer to iterator.zip for the complete code.

Summary of Common XML Uses

The iterator pattern is useful in XML applications when:

- You need to encapsulate the way you walk a given collection. Most of the time in XML applications, this collection will be a DOM tree.

Iterator concludes this overview of the use of design patterns in XML applications. A forthcoming article will present an introduction to some patterns with particular applications to XML.

Design patterns are a powerful way to improve the quality and comprehensibility of your XML applications. Make sure to review the bibliography. You will certainly find more ways to boost your XML development.

If you have comments or questions, the author may be contacted at fabio@viaduct.com

Bibliography

Erich Gamma, Richard Helm, Ralph Johnson & John Vilissides, 1995, *Design Patterns: Elements of Reusable Object Oriented Software.*

John Vilissides, 1997, *Pattern Matching.*

Sherman R. Alpert, Kyle Brown, Bobby Woolf, 1998, *The Design Patterns Smalltalk Companion.*