Design Patterns in XML Applications

by Fabio Arciniegas A.

Part II: XML-specific patterns

Table of Contents

•Introduction
•XMLable Pattern
•Patterns in DTD Structures
•Patterns in Element Definitions
•A Little Advice
•References

Patterns are a useful technique for the transmission of knowledge about recurrent problems in software development. This article, the second of two complementary pieces (see Part I here), is focused on XML-specific patterns as opposed to traditional design patterns in XML specific contexts.
For the first part of this article, some basic knowledge about UML class diagrams will be useful (see our basic UML class diagram guide). For the second part, some basic knowledge of XML DTDs, such as entities, will also be useful.

What are XML patterns?

XML Patterns denotes two kinds of patterns: (1) Program Design Patterns, specifically treating XML-related problems; and (2) Information Structuring Patterns, for the design/implementation of DTDs, schemas etc.
XML patterns of the first kind tend to be compositions and refinements of traditional design patterns. Yet the process of naming and clearly defining them helps in two ways:

- It builds a common base language and base of knowledge for typical XML applications, thus improving understandability, and empowering developers at all levels of expertise.
- It helps XML integrate into the Object Oriented mainstream.

XML patterns of the second kind, those for information design, are focused on finding solutions for common problems in the design of document type definitions (DTDs).
The number of XML patterns is growing quickly, so choosing which ones to present has not been an easy task. I have decided to present here common patterns in the three categories that seem to be most stable in the XML patterns arena: Patterns for Program Design, Patterns for DTD design, and Patterns for DTD Implementation.
XML patterns is a formidable subject, which these articles can only hope to introduce. This article is therefore an invitation to further explore patterns, rather than a catalog of XML patterns. This exploration is not without its pitfalls, which is why I have included a short guide of common misconceptions and warnings at the end of this article. I hope that potential pattern writers can make use of them in order to build a clearer common base of knowledge.
Our tour of XML patterns starts with XML patterns in processing applications, examining the "XMLable" pattern.

XML Patterns in Processing Applications
XML Patterns in DTD structure
XML Patterns in Element Definition
A Little Good Advice
Acknowledgements and References

—

Feb. 16, 2000**Design Patterns in XML Applications: Part II**

XML Patterns in Processing Applications

by Fabio Arciniegas A. XML application design patterns (abbreviated here as "XADP") are named, reusable solutions for common problems at the application level. They are often refinements of traditional patterns.
Because of their nature, XADPs can be easily and neatly expressed in the same way traditional patterns are usually presented. That is, in sections for the name, synopsis, context, solution, consequences, and related patterns.
The following pattern is a typical XADP, called the "XMLable pattern." It has been successfully used in a number of applications, and tackles one common problem in XML-aware applications: the construction of internal representations of XML data as *meaningful* objects.

XMLable pattern

Also known as "XML-reader/writer"
Originator: Fabio Arciniegas A.
Synopsis

The XMLable pattern defines a solution to managing information that is persisted as XML data, but must also be managed as meaningful objects (i.e., not as a general data structure such as the DOM) inside an application.

Context

Suppose you are writing an e-mail program that uses XML documents for the persistence of the messages. This is pretty useful since you can do things like apply various stylesheets to these documents and get all sorts of nice presentations for them. But you also need to upload and manage that information into your program: you need objects that represent your messages.

Keeping the DOM representation of every object can be very memory-intensive, especially when you are managing a large number of messages. More importantly, DOM objects contain no semantics whatsoever about being a message. There is no such thing as an interface enabling other objects to interact with it as an e-mail message (no `setSender(String)`, no `getDate()`, just plain DOM manipulation). Choosing to maintain the DOM representation of hundreds of messages is in most cases a bad design decision; using it would probably lead to a poorly structured, hard to maintain program.

The XMLable pattern addresses the problem of how to create e-mail objects by using the data contained in the XML document without having to keep the DOM representation in memory.

The solution that the pattern suggests is to provide the *emailMessage* class with a partner class, *emailXMLPersistenceManager*, whose sole responsibility is to make the object persist in an XML representation. Whether recovering the state of the object or serializing it in XML, it is the PersistenceManager and not the object

itself that handles this activity.

Forces

The considerations that lead to the general solution proposed by the XMLable pattern are:

- Multiple objects, whose data is gathered from XML documents, need to be manipulated internally.
- Memory restrictions make DOM prohibitive.
- Design and program quality impose the need to represent the data as something more meaningful to the application domain than the DOM tree.

Solution

This figure shows a class diagram depicting the classes and interfaces participating in the XMLable pattern. The descriptions of the roles played by these classes in the pattern are below:

Client

A container responsible for the creation of the XMLableConcreteClass instances. In the e-mail example, this is the EmailProgram class.

XMLableAbstractClass

Gathers (provides the base class for) different classes that can be made persistent through the use of the correspondent ConcreteXMLPersistenceMgr.

XMLableConcreteClass

The actual class whose instances will be registered with the ConcreteXMLPersistenceMgr and finally saved as XML. In the e-mail example, this is the EmailMessage class.

XMLPersistenceMgr

A simple interface declaring the methods that provide XML persistence to an object. This also declares a method to register the concrete XMLPersistenceMgr object with the XMLable object.

ConcreteXMLPersistenceMgr

This is the core of the pattern. The class implements the XMLPersistenceMgr interface. It is also responsible for construcing the XMLable object from XML documents. To do that, the class implements the DocumentHandler methods (defined by SAX) in order to be able to update the registered class from the XML source.

DocumentHandler (defined in SAX)

The ConcreteXMLPersistenceMgr needs to be informed of basic parsing events. In order to do so, it implements this interface and registers with the SAX parser. The parser uses the instance to report basic document-related events such as the start and end of elements.

Consequences

- All the complexity involved in managing the persistence of the object is shifted to the PersistenceMgr.
- There is a tight coupling between the XMLable class and the PersistanceMgr.
- The size of the XMLable objects is smaller. This is very useful in applications handling many instances of the XMLable class.
- Responsibility for instantiation and update of the XMLable object is well separated, allowing for the creation and manipulation of the object even outside of the XML persistence process.

Related Patterns

- *High Cohesion:* This pattern encourages putting specialized methods in special-purpose classes. The use of the PersistenceMgr is a good example of a High Cohesion pattern.
- *Singleton*: The Singleton pattern ensures that only one instance of a class is created. This can be the case for the PersistenceMgr class if, among other reasons, concurrency considerations must be easily minimized.

- *Balking*: If an object's method is called when the object is not in an appropriate state to execute it, the method returns without doing anything. This pattern is useful for systems implementing PersistenceMgr as a Singleton, but where the client may start concurrent requests to save XMLable objects.

In this section we saw a common example of an XML pattern for XML processing applications. In the next section, we will study XML patterns for DTD structuring.

XML Patterns in DTD structure

These patterns are named solutions to recurring problems in the overall structure of document types. Note that the term DTD here is applied in the sense of *document type definition*. These patterns are not restricted to any given form of XML schema definition.
DTD structure patterns are usually smaller than application design patterns. Therefore, two examples will be presented. For more information, see the links in the resources section.

Choice Reducing Container

Originator: Toivo Lainevool
Synopsis
When creating large DTDs with many logical units, authors might be required to learn a large number of these units to know how to use the DTD. Reducing the number of choices the author has to make at any point in the DTD (by grouping related elements beneath newly introduced elements) will reduce the burden on the author.
Context
In a DTD with many logical units, a user of a document can be overwhelmed with the number of choices that have to be made. With many options users have a difficult time knowing how to compose all of the elements available. This is common in large, general-purpose DTDs where many logical units are presented.
Forces

- Either because of the nature of the data to be represented, or because of the intention of making the DTD applicable in many situations, large numbers of logical units need to appear in the DTD.
- Several of the elements can be naturally grouped as members of a higher abstraction (e.g., "magnolia" and "rose" under "flowers").
- The learning process of the user wants to be simplified, presenting him or her with a small number of choices at each point.

Example
Here is a DTD fragment that presents a lot of choice to the author:

```
<!ELEMENT Doc (Para | OrderedList | UnorderedList | Figure
                | Artwork )+>
```

Here the author has 5 different elements to choose from after creating the `doc` element. This choice could be limited by introducing new elements, and grouping some of the existing elements together as children of the new elements, like this:

```
<!ELEMENT Doc (Para | List | Illustration )+)>
<!ELEMENT List (OrderedList | UnorderedList )>
<!ELEMENT Illustration (Figure | Artwork )>
```

Cross-Cutting Metadata

Also known as "Factoring Metadata"
Originator: Fabio Arciniegas A.
Synopsis
During the definition of a DTD, it is not unusual to find several elements sharing a common set of metadata needs. The Cross-Cutting Metadata Pattern identifies such common subsets and encapsulates them, in order to make a clearer DTD.
Context
Elements often have associated metadata (e.g., a unique identifier). Furthermore, many elements can share the same metadata needs. This is often the case in DTDs for element collections. Suppose you are developing a DTD for the items of a music and video shop. Your items, represented as elements, are bound to have many metadata needs in common: an identifier, an availability status, or maybe a recommendation status. The structure proposed by the Cross-Cutting Metadata Pattern is to encapsulate these common metadata needs (very often in a parameter entity), leading to a better organized and more maintainable DTD.
Forces

The needs that lead to the use of this pattern are straightforward:

- There are a number of elements that have metadata requirements.
- These elements share a subset of those requirements.
- The number of elements and the size of the subset are big enough to make the inclusion of a parameter entity (or an attribute group in XML: Schema) an improvement in readability and maintainability, instead of adding "bloat." For example, if there are only 2 elements, and the only thing they share is `ID`, introducing an extra construct is not an improvement.

Solution

Cross-Cutting Metadata takes the common subset of metadata needs and expresses it in whatever mechanism the schema definition language provides for encapsulation (e.g., parameter entities in XML DTDs). It then includes this construct in all the elements that share it. The pattern simply factors the metadata out of several elements. Even though metadata is often expressed in attributes, the pattern can also be applied if the metadata is in the form of elements.

Consequences

- Common metadata is easier to localize, and thus easier to modify.
- When applied to a large number of elements, readability is greatly improved.
- Reusability of metadata declarations is easier to achieve.

Example

This simple example deals with the music and video store DTD mentioned above. Consider the initial declarations:

```
<!ATTLIST video
                id              ID      #REQUIRED
                available       (yes|no|onrequest)  "onrequest"
                onSale          CDATA   #FIXED "yes">

<!ATTLIST CD
                id              ID      #REQUIRED
                available       (yes|no|onrequest)  "yes"
                recommendation CDATA   #IMPLIED >
```

From these declarations we can derive a parameter entity using the Cross-Cutting Metedata pattern:

```
<!ENTITY % cross-cutting-metadata "
         id              ID      #REQUIRED
         available   (yes|no|onrequest) onrequest"
>

<!ATTLIST video
          %cross-cutting-metadata;
          onSale         CDATA   #FIXED "yes"
>

<!ATTLIST cd
          %cross-cutting-metadata;
          recommendation CDATA   #IMPLIED
>
```

We can then simply include this entity in all the element declarations that share them.

Not only has readability improved, but maintainability is higher as well. Now, when we need to add additional metadata to each element (e.g., "onSale"), we can easily and safely add it without enduring the error-prone process of including it manually on each element type.

XML Patterns in Element Definition

Arguably, the most widespread kind of XML patterns are those related to DTD content. These patterns are named solutions to recurring problems in the design of element types.

Not all patterns can or should be expressed in the same way. For instance, traditional behavioral patterns commonly have a different expression from data definition patterns. In this section, I opted to keep the layout for the patterns as defined by Liam Quin.

Running Text

Originator: Liam Quin
This pattern is included in its original formulation.

Synopsis

The Running Text Pattern is used for general textual content that may contain markup at the phrase, word, or symbol level, but not at the block level.

Actors

The Running Text Pattern has these participants:

- *Block Level Elements*: The environment in which the pattern occurs.
- *Internal Markup:* Markup that can occur within Running Text.
- *Running Text Definition:* The implementation of Running Text.

Markup

Running Text is usually represented in a Document Type Definition as a Parameter Entity. The actual elements listed will vary from DTD to DTD, depending on the application; the Pattern specifies only the use of the entity RunningText:

```
<!ENTITY % RunningText
          '
            #PCDATA|Quote|Emphasis|MathML|Phrase|BibRef|
            FootNoteReference
          '
>
```

The pattern is used in the content model of other elements:

```
<!ELEMENT FootnoteBody
          (%RunningText;)*
>
```

The purpose of a single definition for Running Text is two-fold: firstly, to encapsulate the concept of generic running text, making the intent of a document type definition clearer; secondly, to ensure that the same set of basic elements is allowed everywhere text is allowed.

Additional elements can be added for a specific situation as follows:

```
<!ELEMENT PlaceName
          (%RunningText;|PlaceAlias|GridReference)*
>
```

Processing

This pattern does not require special processing. It is normally only seen by a validating XML processor.

Variations

In a complex Document Type Definition, it may be convenient to include other parameter entities in the definition of RunningText:

```
<!ENTITY % RunningText
          '
              #PCDATA|Quote|Emphasis|Phrase|BibRef|
              %elements.footnotes;|%elements.MathML;
          '
>
```

Marker Attribute

Originator: Fabio Arciniegas A.

Synopsis

The Marker Attribute Pattern is used when certain elements need to be marked via an attribute so they can be processed in a different way by a style sheet/program that recognizes the mark.

Actors

The Marker Attribute Pattern has three participants:

- *Marker Attribute*: The marker is an attribute whose only purpose is to signal a binary state. If the attribute is present, the element must be treated differently.
- *Marked Element*: The element that may contain the Marker Attribute.
- *Processing Application*: The responsibility for performing the special action if the mark is encountered. This is usually encapsulated in a style sheet.

Markup

The markup necessary for this pattern is reduced to an attribute declaration:

```
<!ELEMENT video (title,artist,whatnot)>
<!ATTLIST video onSale  CDATA   #FIXED "yes">
```
and, possibly, the appearance of the attribute in the XML instance:
```
<video onsale="yes">
   ...
```
Processing
As mentioned above, a key characteristic of this pattern is outside of the XML document. The special behavior derived from the marking is usually achieved by means of a style sheet. The following example shows a simple case.
Example
A Marker Attribute for items on sale can be applied to the elements of a hypothetical DTD for videos as shown above. A simple XSLT style sheet can take care of a special presentation for the marked elements:
```
<xsl:if test="@onSale">
  <h4>
    <xsl:value-of select="artist"/> is on sale.
  </h4>
</xsl:if>
<!-- handle the rest of the element -->
```

Advice for the Use and Creation of XML Patterns

by Fabio Arciniegas A.

A Little Good Advice

During the use and creation of patterns, several misconceptions and pitfalls can be encountered. Since XML patterns are no exception, I would like to finish by briefly highlighting some of the main trouble spots. For more advice on healthy pattern use, I recommend John Vilissides' book "Pattern Hatching" (see References).

Patterns Are Not the Holy Grail

Patterns are a powerful way to communicate expertise: they create a common design language, they help make your system more understandable to others, etc.... But they are not a replacement for creativity, nor are they automatic quality assurances. *Patterns are just another tool in your box*—learn them, use them, enjoy them, but don't overestimate them.

Tautologies Are Not Patterns

This phenomenon seems to have cooled down in the traditional pattern world, but it appears to still be a problem in the XML patterns arena. XML "patterns" that merely state a tautology like "use an attribute where an attribute is needed" are not useful for anyone. This problem was pointed out a long time ago by Rick Jelliffe, but still seems common enough to merit mentioning here.

Patterns Are *Not* Restricted to Particular Aspects of XML Applications

Depending on our personal background, we tend to see some areas as more suitable for pattern creation than others. Some people take this to extremes, claiming XML patterns can only be used in particular situations. This is obviously a mistake. Opportunities to help others gain expertise about recurrent problems and solutions arise in every area. Patterns are a great tool—we don't have to restrain ourselves, let's use them wherever they are useful!

Conclusion

This concludes our brief introduction to XML Patterns. Please write to me (fabio@viaduct.com) if you have questions, suggestions, or want to discuss further work in this field.

Acknowledgements and References
I would like to thank Liam Quin, Rubby Casallas, and Toivo Lainevool for their contributions to this article.

Bibliography

Erich Gamma, Richard Helm, Ralph Johnson & John Vilissides, 1995, *Design Patterns: Elements of Reusable Object Oriented Software*.
John Vilissides, 1997, *Pattern Hatching*.
Sherman R. Alpert, Kyle Brown, Bobby Woolf, 1998, *The Design Patterns Smalltalk Companion*.
Ian Graham and Liam Quin's web pages *"Introduction to XML Design Patterns"* at http://www.groveware.com/xmlbook/patterns.html
Rick Jelliffe, 1998, *The XML & SGML Cookbook: Recipes for Structured Information*, Charles F. Goldfarb Series on Open Information Management, ISBN 0-13-614223-0.

See also

More XML patterns can be found at Toivo Lainevool's forthcoming site, http://www.xmlpatterns.com/