Separating Concerns of
Modeling from Artifact Generation
Using XML

J. Craig Cleaveland

Introduction

Domain Specific Visual Languages are used to develop models from which other software artifacts can be automatically generated such as implementations, system documentation, user documentation, and test scripts. Such a system can be divided into two subsystems. The first subsystem provides the visual display, editing and storage of the model. Second is the generation of other software artifacts. The position taken by this paper is that:

1) Formally dividing the system in this manner promotes separation of concerns,
2) A formal interface is necessary between these two subsystems to achieve separation of concerns,
3) XML is a reasonable language for the interface.

Separation of Concerns

The concerns of displaying and editing models are distinct and separate from generating software artifacts. Displaying models concerns include the visual representation of a model for human understanding, the appropriate choice of icons, colors, line styles and layout of information. Editing a model has another set of concerns including usability issues in the user interface, focusing on particular aspects of the model, and finding different elements of the model. These two sets of concerns are very different from the concerns of generating other software artifacts.

To generate other software artifacts one is concerned with the abstract structure of the information independent of how it might be visually displayed or edited. The concerns for generating other software artifacts include accessibility to information, completeness of the model or methods for automatically completing the model. Additional information may also be needed such as performance requirements or assumptions about the operating environment. These additional information requirements may not be necessary for the model.

Putting all these concerns into a single software system limits future opportunities. These concerns can be kept separate by building independent subsystems, one for displaying and editing models, the other for generating software artifacts.

Let’s examine a few scenarios where this separation of concerns becomes important. Our baseline is a working software tool that displays and edits models, and a separate subsystem for generating artifacts. In scenario one, a new development requires the same kind of implementation, but rather than have a systems engineer manually create a model using the editor, the information is already available in a database (perhaps generated from other tools). All we need do is extract the information from the database and pipe it into our generator. This scenario shows the importance of being able to replace the display/editor portion with alternative methods for obtaining equivalent information.

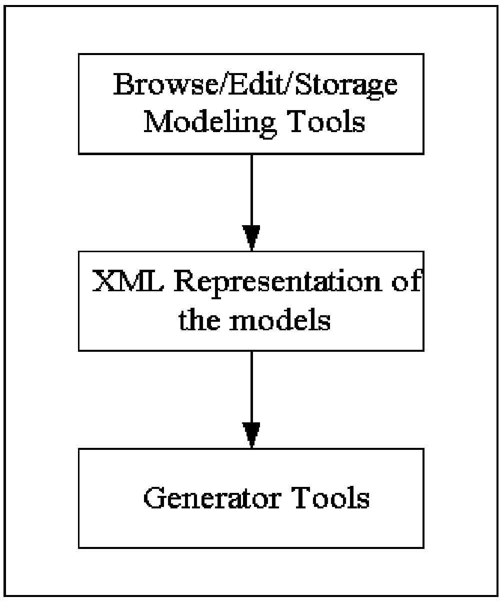
In scenario two, another new development wants to reuse the display/editor portion but requires different software artifacts. Again, a portion of the system can be reused without any substantial changes.

Formal Interface

To achieve separation of concerns, a formal interface must be developed between the two subsystems. This interface provides a representation of the model. Once this interface is formally defined and adopted, the two subsystems can be developed independently and concurrently. The scenarios described in the previous section are easily accommodated by this structure because only knowledge about the interface is necessary to provide alternatives for either subsystem. Next, we consider why XML should be chosen as the standard method for representing models.

XML

XML is a standard data representation language. It is a reasonable language for representing models and achieves separation of concerns. It is actually more than that because it integrates the system with the much wider world of XML technologies and tools, many of which are already suitable for generating other software artifacts. For example, XSLT is a standard transformation language for transforming between XML languages or to other text-based languages. Thus, by adopting XML as the interface, it is possible to readily use off-the-shelve standard components for creating generators.


Browse/Edit/Storage Modeling Tools
XML Representation of the models
Generator Tools
Figure 1: Using XML as the interface to formally separate the concerns of editing from generation.

There are three possible approaches to using XML for representing models. First, a structure could be devised that is used by all types of models. This generic structure is defined once and used in every domain and every type of model. This means no additional work is required to export the XML representation of a model of any domain. The second approach is to use a model-specific structure for each domain. The model-specific structure could be automatically derived from the model structure. Thirdly, the representation could be custom created that maps directly to the concepts and elements of that domain (which may be different from the model structure used in the modeling tool).

To illustrate these three approaches, consider the trivial example of a single state node with two properties: a label and a color. Let’s say in the modeling tool, the type of the node is declared as “`statenode`”.

First approach – Generic for all types of models:

```
<node type=”statenode” id=”33”>
  <property name=”label”>My Node</property>
  <property name=”color”>Red</property>
</node>
```

Second approach – Model-specific (automatically derived from model structure)

```
<statenode id=”33”>
  <label>My Node</label>
  <color>Red</color>
</statenode>
```

Third approach – Custom designed

```
<state id=”33” color=”Red”>My Node</state>
```

The XML exports of a modeling tool for the first and second approaches can be completely automated. One could even provide some simple customization in the first two approaches, for example, whether properties should be represented as XML subelements or XML attributes. The third approach would require the effort of designing an XML structure and the coding to export that XML structure. However, writing generators for the third approach (and perhaps the second approach) would be easier since the domain objects are represented more simply and naturally than in the generic approach.

Generation Techniques

There are several approaches to writing generators whose input is XML, including XML based standards such as XSLT, program based approaches using DOM and SAX, and template approaches.

XSLT is a transformation language to transform XML documents to other XML documents, HTML, or plain text based languages. XSLT is an XML language that describes the desired transformation. It is fed to an XSLT processor along with the XML input file. Although XSLT is primarily focused on XML to XML or XML to HTML transformations, it is also adequate for generating plain text documents. However, some transformations, particularly those that require some analysis of the information are difficult to express in XSLT.

Program based approaches use standard APIs for reading and extracting information from an XML document. The two major APIs are DOM (Domain Object Model) and SAX (Simple API for XML). These APIs are language independent so a variety of programming languages can be used such as Java, C, C++, and Perl. This approach requires considerable coding effort.

The recommended approach for non-toy generators is templates. A template will look like the desired output with the variable parts expressed in some language that extracts information from the XML document and computes the output information. JSP (Java Server Pages) is an example template language used for creating dynamic web pages, but JSP can also be used to create programs. Other template languages that are designed specifically for use with XML are ideal for this purpose such as XTags and TL (Chapter 12 of *Program Generators using XML and Java*). Both of these template languages hide the details of parsing, constructing and traversing data structures, and other programming details. XTags is a Java Tag Library for XML based on XPath and is freely available as part of the Apache Jakarta project. TL is a template language also based on Java and XPath but not based on JSP. TL is also freely available at http://craigc.com and described in *Program Generators using XML and Java*.

Conclusion

A system for implementing a visual domain specific language should be cleanly separated into two parts. One part focuses on the display and editing of models. The second part generates various software artifacts including software, documentation, and test support. The interface between these two parts should be XML using either a generic approach for all domains or domain-specific structures. This software architecture supports the appropriate separation of concerns between the creation and display of a model and the use of the model to generate a variety of products.

For more information about the author or topic, see http://craigc.com