Master's Project Proposal
An Application to create Problem-Specific DOMs for XML

Liangxiao Zhu
lxz0062@cs.rit.edu
October 9, 2002

1. Summary

The purpose of this project is to develop an application that turns XML sources into Problem-Specific Data Object Models (PSDOM). The W3C DOM (Document Object Model) [5] is the most popular API that parses XML documents into tree structures of data. But DOM provides a generic data model rather than a problem-specific data object model, which makes it complicated and inefficient. And in many cases, programmers don't use the generic data model. They usually need an object model that is specific to a particular problem. This is where this project comes in. The goal of this project is to provide an application that offers a convenient way to convert XML documents to Java objects. This application contains a Class Definition File (CDF) compiler and a binding framework. The CDF compiler takes a Class Definition File, which specifies the constraints of XML data and classes for XML elements, and compiles it into Java class files representing XML elements. It also generates observers (this "observer" idea is from Dr. Schreiner's "oops" [1]) that will be used to unmarshall XML documents based on the class definition file. The binding framework uses the observer to parse an XML document and turn it into a problem-specific Java object tree representing the XML document. The PSDOM makes it easy to create applications to process XML data. It maps XML into Java objects and allows user to manipulate the document in the same way the user manipulates Java objects.

Figure 1 shows the data flow of this application.

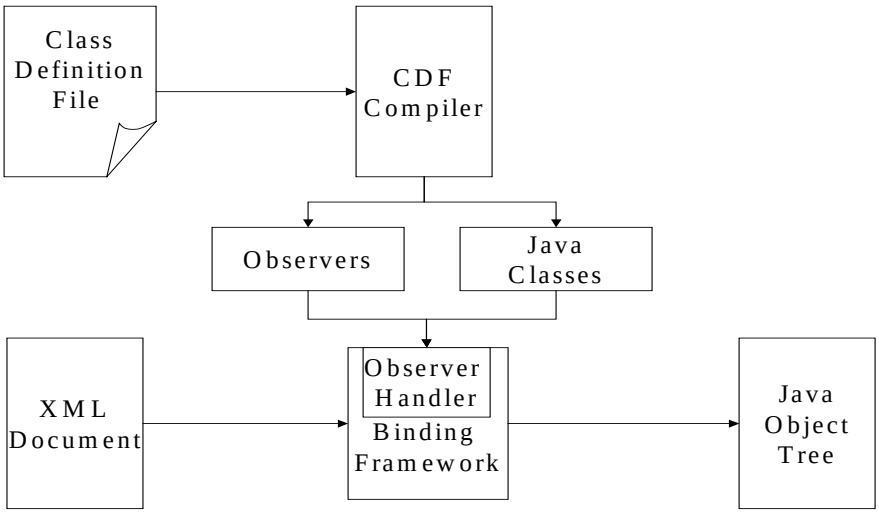

Figure 1. Data Flow

2. Overview

XML has certainly taken the programming world by storm. It is a form of self-describing data, which makes it the best candidate as a data exchange medium between different systems. Therefore, how to parse XML and manipulate the XML data becomes a very essential task of handling XML documents.

Basically, there are three major methods of accessing XML data [8]. The first one is Callbacks. Simple API for XML (SAX) is the W3C standard that uses this Callbacks method. It is an event-driven, serial-access mechanism for accessing XML documents. SAX is fast and less memory-intensive because it does not store any part of the document in memory. On the other hand, SAX API is a bit harder to visualize because of its event-driven model; therefore, it is hard to code with. Another method is Trees. The most popular one for this method is the W3C Document Object Model (DOM) [5]. DOM is a "platform- and language-neutral interface". It parses an XML document and constructs a generic object model in memory that represents the contents of the XML document. DOM is easier to use than SAX. Its tree structure makes it easier to visualize. And the API offers rich functionality. But what DOM provides are generic data models, which make it more complicated and inefficient to access than a problem-specific data model. It also carries a lot of ballast when handling a simple XML. The third one, also the newest method, is data binding. Sun's JAXB [2] is using this method. It maps the data in XML documents to small Java objects. These objects are related to the elements and attributes of the XML document. It becomes much easier for developers to access and handle these data in Java applications and convert the XML document from one format to another.

Of course, Java developers want to work with the XML document in an easy and directly perceived way. They want to put more time on the development of business logic instead of spending too much time boning up on XML and XML parser specification. They want to write applications caring more about XML as data than as documents. In this project, we will provide such a tool that will be as fast as SAX parsers, as directly perceived as W3C DOM and as easy to create application processing XML data as data binding. Some of the ideas in this project are inspired by Dr. Schreiner's Oops [1] and the observer pattern in Oops. Oops is an object-oriented parser generator targeted to Java. It takes a grammar as input and automatically generates a parse tree. The parse tree, combined with the scanner, composes the parser for this grammar. The parser will generate object trees for input files based on this grammar. How to execute this tree is accomplished by the idea of adding observers to the parser. Different users can write different observers for the same grammar, hence, execute different tasks on the grammar. This project borrows the observer idea to implement the binding framework.

3. Functional Specification

3.1 Class Definition File

Class Definition File itself is an XML document. It is written based on DTD file. It contains Java-like code as element contents, which gives the specification of generating the observers and the element classes.

Following is an example of a DTD and a Class Definition File that is based on the DTD.

A simple DTD:

```
<!ELEMENT person (firstName, lastName)>
<!ELEMENT firstName #PCDATA>
<!ELEMENT lastName #PCDATA>
```

The Class Definition File based on this DTD file:

```
<element name=”person”>
      String firstName;
      String lastName;
      <element name=”firstName”>
            firstName = new String(PCDATA);
      </element>
      <element name=”lastName”>
            lastName = new String(PCDATA);
      </element>
      return new Person(firstName, lastName);
</element>
```

The example shows that the person element contains one firstName sub element and one lastName sub element. The Person element class has two String type fields; one is firstName, and the other is lastName, which corresponding to its subelements. The types of firstName and lastName elements are both String. They take their element content as value. Here, the PCDATA represents the element content.

3.2 Compiling Class Definition File to classes and observers

The CDF compiler takes the Class Definition File as input, and maps it into observer source files and element classes.

Here are the results generated from the Class Definition File shown in 3.2:

A. Class for person element:

```
public class Person {
  private String firstName;
  private String lastName;

  public Person(String firstName, String lastName) {
    this.firstName = firstName;
    this.lastName = lastName;
  }
  public String getFirstName() { return firstName; }
  public void setFirstName(String data) { firstName = data; }
  public String getLastName() { return lastName; }
  public void setLastName(String data) { lastName = data; }
}
```

B. Observers used to unmarshall XML document to Java objects

```
protected class PersonAdapter extends ObserverAdapter {
  protected Object value;
  public Object value () { return value; }
  public Observer init (int rule) {
    switch (rule) {
    // <!ELEMENT person (firstName, lastName)>
    case 0:
      return new PersonAdapter ( ) {
        {
          String firstName;
          String lastName;
```

```
          }
          public void shift (Observer sender) {
            if (sender.getName().equals("firstName") {
              firstName = sender.value();
            } else if (sender.getName().equals("lastName") {
              lastName = sender.value();
            } else {
              throw Exception("Shifted wrong subelement!");
            }
          }
          public void reduce() {
            value = new Person(firstName, lastName);
          }
        };
      // <!ELEMENT firstName #PCDATA>
      case 1:
        return new PersonAdapter ( ) {
          public void shift (String data) {
            value = data;
          }
        };
      // <!ELEMENT lastName #PCDATA>
      case 2:
        return new PersonAdapter ( ) {
          public void shift (String data) {
            value = data;
          }
        };
      default:
        return null;
      }
    }
}
```

3.3 Turning XML documents into Java object trees

The binding framework is in charge of turning XML documents into Java object trees. It takes an XML document and parses the file using an XML parser (in our case, we use SAX parser). The XML parser calls the observer Handler, which in turn invokes observers defined for each rule in the DTD file like the one above. The observers build Java object trees by instantiated Java classes generated by the CDF compiler. Figure 2 shows this workflow.

3.4 Application usage

Using this application is quite simple. You just need to follow these steps:

- Get the DTD file of XML documents; write a Class Definition File based on DTD file. The CDF contains Java-like code, which specifies how the classes of elements and observers should be generated.
- Use CDF compiler and the given Class Definition File to generate Java class files and observer source file.
- Compile those Java class files and observer source file using Java compiler.
- Use binding framework to parse the XML document and turn it into a Java object tree.
- Manipulate the generated object tree.

4. Architectural Overview

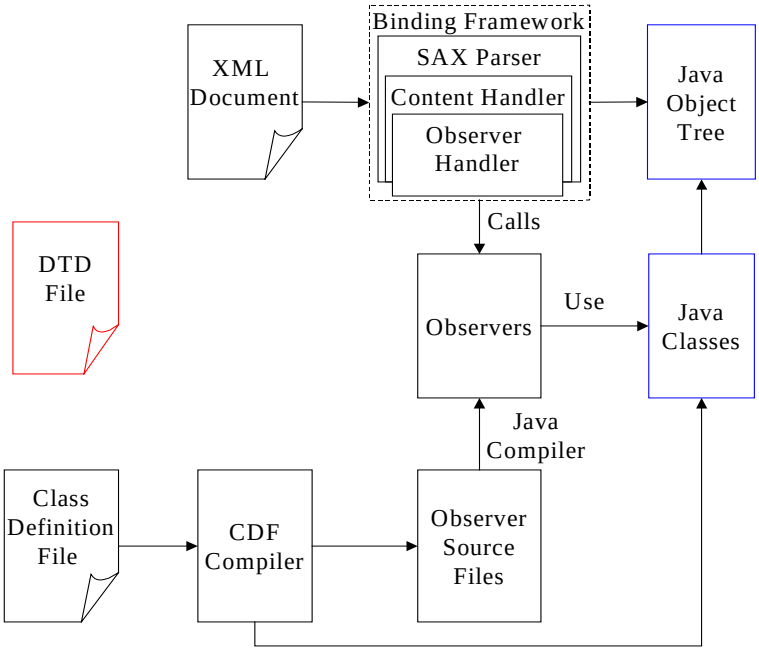

Figure 2. PSDOM Overall Architecture

Figure 2 shows the overall architecture of this project.

4.1 Class Definition File and CDF Compiler

The application developer needs to write Class Definition File according to the DTD file. The CDF is an XML document that is written in a specific format, which will be defined in this project. The CDF Compiler takes the Class Definition File and compiles it into Observer Source Files and element classes. The Observer Source Files can then be compiled with Java Compiler into Observers that will be later hooked into the Observer Framework. The generated Java Classes are in the bean-like style with gets and sets methods. They represent the object-oriented view of DTD elements definitions and attribute list definitions. At this time, the application developer has already been able to instantiate generated Java Classes to build the XML content tree and then manipulate the XML data with Java Objects.

4.2 Binding Framework and Observers

If an XML file exists, of course the XML should have been validated against the DTD. The application developer can use the Binding Framework to convert an XML document into Java Objects and form the content tree. The Binding Framework is composed with SAX parser (or other XML parser) and Observer Handler. The SAX parser consumes an XML file, generates SAX2 Events[6]. Observer Handler processes SAX2 Events, then in turn, instantiates Observer and shifts XML data to Observer by calling observer's methods. The observer actually does the real work to instantiate Java Classes, create Java Object and build the XML content tree.

4.3 Writing CDF Compiler

One interesting thought is that since the Class Definition File is written in XML, theoretically, we can use the same architecture of Binding Framework to build the CDF Compiler as shown in figures 3 and 4.

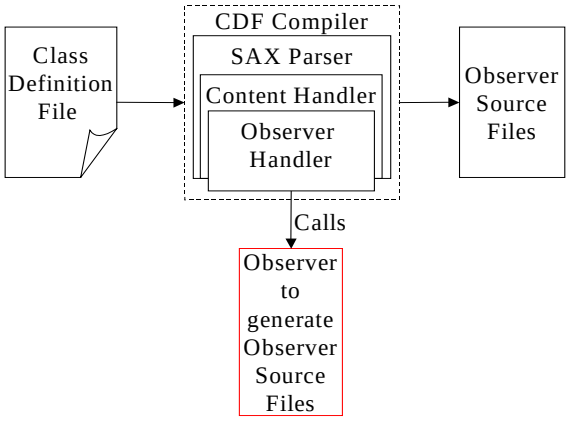

Figure 3. CDF Compiler Part 1

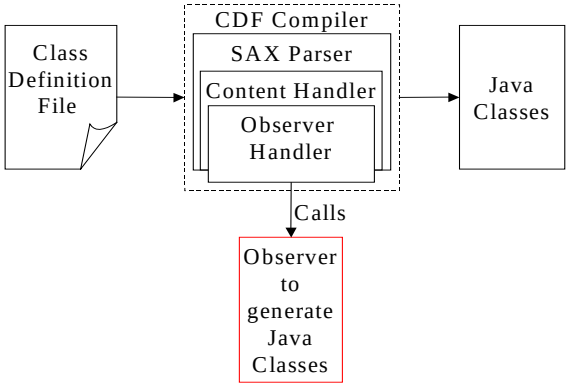

Figure 4. CDF Compiler Part 2

Then only thing left that we need to do is to write observes used to generate Observer Source Files and Java Classes.

5. Deliverables

The following are the principal deliverables of this project. All of these will be contained in a CD:

- Source code and executable files of CDF compiler and binding framework.
- A technical report for the design architecture and specifics of this project.
- Examples of using this tool, which include DTD, Class Definition File, XML documents and Java classes.

6. Draft Table of Contents of Final Report

- Description of XML data binding
- Architecture of this tool
- Design specification of Class Definition File
- Design of binding compiler
- Analysis of observers pattern
- Conclusion and future work
- Bibliography and References

7. Schedule

- **September 2002**
  Work on the feasibility for this project. Write observers and element classes by hand for simple DTDs. Use SAX parser to parse XML documents, to see if can get Java object tree as expected.
- **October 2002**
  Finish detailed design
- **November and December 2002**
  Implement the CDF compiler and binding framework.
- **January 2003**
  Code test and writing examples.
- **February 2003**
  Complete technical report and prepare the presentation.

References

[1] Axel-Tobias Schreiner. *RIT Oops homepage*
http://www.cs.rit.edu/~ats/projects/oops/edu/doc/
Introduce of the RIT Oops and Oops API.

[2] Mark Reinhold, Core Java Platform Group. *The Java™ Architecture for XML Binding (JAXB). Working-Draft Specification. Version 0.21*. 30 May 2001
JAXB specification.

[3] Sun Microsystems, Inc. *The Java™ Architecture for XML Binding User's Guide.* May 2001
JAXB user's guide.

[4] World Wide Web Consortium. *Extensible Markup Language (XML) 1.0 (Second Edition)* W3C Recommendation 6 October 2000
http://www.w3.org/TR/REC-xml
W3C XML specification.

[5] World Wide Web Consortium. *W3C Document Object Model Level 1 Specification*
W3C Recommendation 1 October 1998
http://www.w3.org/TR/REC-DOM-Level-1/
W3C DOM specification.

[6] David Megginson. *SAX*
http://www.saxproject.org/
Homepage of SAX project.

[7] Brett McLaughlin. *Data binding from XML to Java applications.* Enhydra Strategist, Lutris Technologies. August 2000
Tutorial of XML data binding with java applications.

[8] Brett McLaughlin. *Java & XML Data Binding*. O'Relly May 2002
In-depth technical look at XML Data Binding.

[9] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison-Wesley 1995
A book of design patterns that describes simple and elegant solutions to specific problems in OOD.