Integrating and Processing XML Documents with JavaBeans Components

Yin-Wah Chiou
Department of Information Management
National Penghu Institute of Technology
Makung City, Penghu, Taiwan, R.O.C.

ABSTRACT

The *eXtensible Markup Language* (XML) and JavaBeans component model have gained wide popularity in the *Object Web* computing. This paper explores how *JavaBeans* components can be used to integrate and process the XML documents. It covers *Bean Markup Language* (BML), *XML BeanMaker*, *XML Bean Suite*, and *Xbeans*. The most powerful JavaBeans connection language is *BML*, which represents an integration of XML and JavaBeans components to provide a mechanism for implementing active content. *XML BeanMaker* is used to generate JavaBeans from XML DTD files. *XML Bean Suite* is a toolkit of JavaBeans components to provide a comprehensive set of functionality to manipulate XML content. The *Xbean* is a powerful paradigm to process XML-based distributed applications.

Keywords: JavaBeans, XML, BML, XML BeanMaker, XML Bean Suite, Xbean.

1. INTRODUCTION

The W3C's XML and Sun's JavaBeans component model are playing an increasingly important role in the Object Web technology. XML is a new and promising metalanguage to describe the content of Web documents. JavaBeans components allow developers to package smaller grained pieces of reusable functionality. The key concepts of JavaBeans include *properties* (attributes of the component), *events* (notifications of the state changes), *persistence* (saving and restoring state), *methods* (services provided by the component), and *introspection* (discovering properties, methods, and events). In this paper, we examine how XML can be integrated and processed with JavaBeans component model. In this issue, we describe IBM's *Bean Markup Language* (*BML*), IBM's *XML BeanMaker*, IBM's *XML Bean Suite*, and *Xbeans* components (from Xbeans.org).

BML is an instance of an XML-based component configuration or wiring language customized for the JavaBeans component model [17]. There are two implementations of *BML processor*, including BML player and BML compiler. The *BML player* is a very small interpreter for processing BML documents and creating the desired bean hierarchy. The *BML compiler* converts any BML script into reflection-free Java code for capturing the inter-component structure of the application. *XML BeanMaker* is a software tool (written in Java) for generating JavaBeans out of XML DTD files.

XML Suite of Beans provides a comprehensive set of functionality for manipulating XML content, such as viewing, searching, editing, or processing XML documents [3]. The classes in the XML Bean Suite can be divided into five sets of related JavaBeans, including XMLCoreBean, XMLConvenience, XMLProcessing, XMLViewer, and XMLEditor. The *Xbeans* are JavaBeans that manipulate XML data; with the appropriate set of Xbeans and a JavaBean design tool, it is possible to build useful distributed applications with little or no programming [12]. Therefore, Xbeans are focused on the distributed applications.

2. XML WITH JAVABEANS

The growth of the JavaBeans component model has been fueled by the ability to integrate and process XML documents. In this section, we first describe the basics of XML technology and JavaBeans component model. We then examine *Bean Markup Language* (BML), *XML BeanMaker*, *XML Bean Suite*, and *Xbeans*.

XML Technology

The XML provides a data interoperability format for information exchange. It plays a critical role in the evolution of the Web's data representation. The Web's data representation is shifting from *structural HTML markup* (i.e., representing a presentation structure of a document) to *semantic XML markup* (i.e., representing logic data). As with HTML, XML identifies data using *tags* (i.e., *element types*). But unlike HTML, XML allows the users to define their own tags. The XML data elements have well defined *content-oriented* tags to describe their content.

The XML document is *self-descriptive*, and the document description is provided in the *Document Type Definition* (DTD). The *well-formedness* (i.e., dealing with physical structure) and *validity* (focusing on the logical structure of elements) are two important concepts of XML document. A valid document is always well-formed but a well-formed document is not necessary valid [14]. Therefore, DTD is an optional part of an XML document. That is, an XML document can be written without DTD. Sometimes the validation is not important in some applications. In this case, the specification of DTD is not needed to make processing as efficient as possible.

The *Application Programming Interface* (API) is used to process an XML document by accessing internal structure. There are two widely used APIs for XML processor (parser), including *Simple API for XML* (SAX) and *Document Object Model* (DOM). The SAX (defined by XML-DEV group) is an *event-driven* API to the process of parsing an XML document. The events include the start of an element, the end of an element, characters, and so on. In contrast, DOM (defined by W3C) is a *tree structured-based* API. DOM defines an object-oriented API to provide a powerful tool for managing XML document such as accessing every element in a document and updating the content and structure of documents.

JavaBeans Component Model

The *software components* are self-contained, reusable building blocks that encapsulate semantically meaningful application or technical services. In Java, a component is a set of related Java classes. JavaBeans Component model extends "Write Once, Run Anywhere" capability to provide support for reusability, portability, and interoperability. The JavaBeans components (or Java classes) are called *Beans* (reusable software components).

To develop Beans, it is necessary to have *Beans Development Kit* (BDK) and the *Java Development Kit* (JDK). A visual application *Java builder tool* (e.g., IBM's *VisualAge for Java*) can be used to build JavaBeans. The Beans can be visually manipulated by using the Java builder tool. The following is a list of salient features of *JavaBeans component model* [1, 11, 15, 16]:

- *Properties*: The properties expose a component's public attribute data via *accessor* methods wrapped around them. The Bean's appearance and behavior attributes can be changed at design time. By using *property editors* or Bean *customizers*, Bean's properties can be customized at design time.
- *Events*: The events specify a component's response to external stimuli or internal conditions, such as a property value changing. Beans use events to communicate with other Beans. The event notification scheme involves three Java interfaces: *Event*, *EventSource*, and *EventListener*. The source Bean notifies all registered listener Beans, passing each an Event object when the event of interested occurs.
- *BeanInfo*: The BeanInfo provides builder tools with enough information to guide users in using the Bean. A *Bean Information* class implements the BeanInfo interface.
- *Persistence*: Beans must support persistence, by implementing either *Serializable* or *Externalizble*. Persistence enables Beans to save their state, and restore that state later.
- *Methods*: Bean's public methods (or operations) describe Bean's *behavior*. The methods can be invoked by others (Beans or a scripting environment). The builder tools or the users can employ methods to construct connections between Beans.
- *Introspection*: The builder tools discover a Bean's properties, methods, and events by introspection. Bean introspection relies on the *core reflection* API to discover Bean features via design patterns (specific naming conventions). The reflection provides a runtime representation of the definitions a programmer has written.

Bean Markup Language

The *Bean Markup Language* (BML) from IBM alphaWorks is an XML-based JavaBean component configuration or wiring language to describe the creation, configuration, and interconnection structure of a set of JavaBeans. The XML *Document Type Definition* (DTD) consists of a set of markup tags (element types) and their interpretation (i.e., defining the meaning of tags in DTD). In BML, the language element types define operators to configure a set of JavaBeans. Therefore, the BML DTD's functions are to describe the relationship and configuration of components.

The *BML processors*, including *BML player* and *BML compiler*, are components to process documents conforming the BML DTD. Figure 1 illustrates an overview of these two implementations. The *XML parser* converts an XML file into DOM tree, which is further processed by using either BML player or BML compiler. That is, BML employs BML player or BML compiler to create and run groups of interconnected JavaBeans. The following describes the significant functionality of *BML player* and *BML compiler* [2, 6, 17]:

- *BML Player*: This processor is an *interpreter* to play a BML script for creating the desired bean hierarchy, which is then a running application. This is implemented

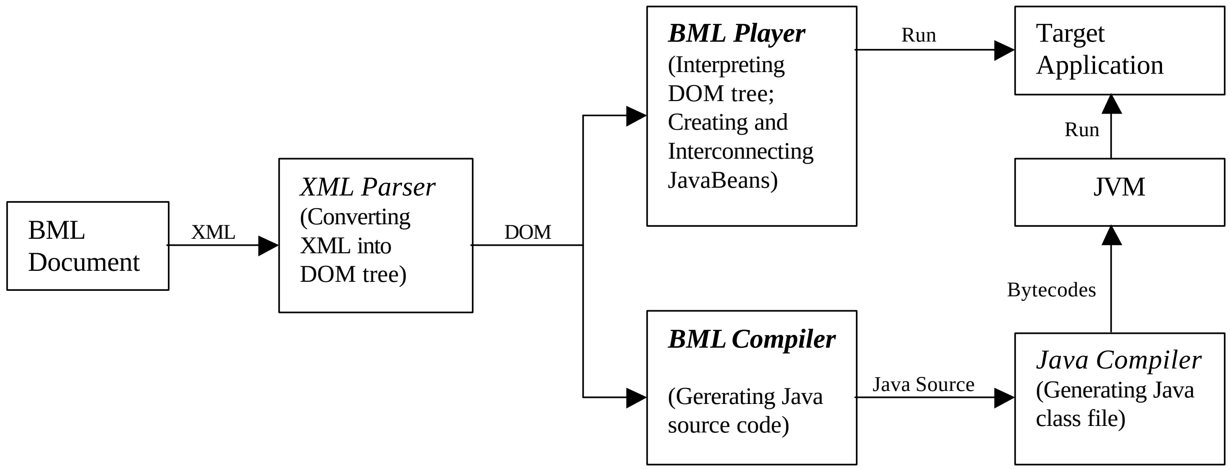
BML Document
XML
XML Parser (Converting XML into DOM tree)
DOM
BML Player (Interpreting DOM tree; Creating and Interconnecting JavaBeans)
Run
Target Application
Run
JVM
Bytecodes
BML Compiler (Gererating Java source code)
Java Source
Java Compiler (Generating Java class file)

Figure 1. Two Implementations of BML Processor

using *reflection*. The BML player is a very small run-time kernel to read the BML document using an XML parser. The player then traverses the DOM tree, creating and interconnecting JavaBeans as specified by the tree. Using the Java plug-in from JavaSoft, the BML player can be easily embedded in applications or Web pages. Since the BML player uses reflection to identify methods at runtime, the performance issue is a major concern. The BML compiler will solve this problem.

- *BML Compiler*: This processor also uses an XML parser to read the BML document. But unlike BML player (interpreting the DOM tree), the BML compiler converts any BML script into *reflection-free* Java source code, which is compiled with a *Java compiler* to generate a *class file* containing Java *bytecodes* (i.e., platform-independent codes interpreted by the Java runtime system). The *Java Virtual Machine* (JVM) loads the class file. The JVM provides a well-defined runtime framework to allow a Java application running on any operating system. The advantage of this implementation is that it allows one to capture the inter-component structure of the application.

Figure 2 illustrates a BML example, which covers the *tags* (*element types*): bean, string, property, args, cast, add, field, call-method, event-binding, and script. We summarize each of *BML language element types* [2, 6, 7] as follows:

- *bean*: A bean element is used to create new beans, as specified by its *class* attribute, or to look up beans by name. The JavaBean may optionally be registered into BML's *object registry* via the bean's *id* attribute, which gives a name to the newly created JavaBean instance.
- *string*: It is used to create a string bean or look one up. The string's contents can be specified either as a *Text* item inside of the <string> tag or using the *value* attribute to create an empty string. Also, a string may be given an *id* for registering it into the object registry.
- *property*: This element is used to *set* or *get* the values of bean's properties. A property's value with primitive type may be set directly via the *value* attribute. The property may also be of some other type. A *type converter* is used to convert objects from one type to another. The property's value can be obtained simply by specifying the property's name in the *name* attribute.
- *field*: Since BML was designed to be usable with all Java objects (not just JavaBeans), it provides the field element for setting or getting the values of an object's fields directly.
- *args* and *cast*: The args element is used to specify constructor requirements. It allows creation of objects using constructors with an arbitrary number of arguments. The cast element is used to convert the type of a bean or value.
- *add*: This element creates a hierarchy of beans by adding one to another. The *add* element treats the object as a container to add the enclosed objects to it.
- *event-binding*: This element provides the binding of events, specified by the *name* attribute, that are emitted by a bean. It also allows events to be bound, via a *script* element, to an arbitrary set of actions. The event-binding element establishes an event listener relationship between a *target* bean (event source) and some action performed when the event occurs.
- *call-method*: It enables inline method invocations. The invoked method is specified in the call-method element's *name* attribute. The invoked object is specified in the *target* attribute.
- *script*: This element defines an executable sequence of scripting statements to be used somewhere. BML v2.2 provides support for the use of BML, JavaScript, and NetRexx as scripting languages.

```
<? xml version = "1.0" ?>
<bean class = "FirstExampleBean" id = "Example-1">
 <!-- create a new bean of type FirstExampleBean;
       id attribute gives a name to the newly created JavaBean instance -->
     <property name = "title" value = "A BML Example" />
     <!-- set a property in a bean; its value may be a string or some other type -->
     <add>
     <!-- treat the object as a container; add the enclosed objects to it -->
        <bean class = "SecondExampleBean" id = "Example-2" >
        <!-- add the SecondExampleBean to the FirstExampleBean (a container)  -->
          <args> <cast class = "int" > <string> 100 </string> </cast> </args>
          <!-- define an argument to the SecondExampleBean's constructor;
                the string is to be cast to an int -->
           <property name = "ContainedExampleBean" value = "Inside Example Bean" />
          <event-binding target = "EventSourceObject" name = "action" >
               <script>
                  <field name = "starting" id = "STARTING" />
                  <call-method target = "InvokedObject" name = "StartMethod" />
               </script>
            </event-binding>
         </bean>
         <string value = "inside" />
      </add>
  </bean>
```

Figure 2. A BML Example

XML BeanMaker

The *XML BeanMaker* (from IBM alphaWorks) is a software tool written in Java to be used for generating JavaBeans out of XML DTD files. It reads a DTD file and generate Java class interfaces corresponding to the elements and attributes in the DTD file; the root element of the DTD is converted to a bean class, and every element inside of it is converted into an inner bean class [4]. That is, for a given DTD file, the XML BeanMaker can generate a JavaBean and all of its necessary Java classes.

XML Bean Suite

XML Bean Suite (from IBM alphaWorks) is a toolkit of JavaBeans components to process XML. It contains a large number of classes, which can be divided into five sets of related JavaBeans. The following describes the five categories of *XML Beans* [3, 8, 9, 10]:

- *XMLCoreBean*: This Bean set contains *nonvisual* beans (e.g., DOMGenerator, XMLFileGenerator, XMLStringGenerator, and NodeArray) for converting XML between text and DOM representations and managing DOM nodes. *DOMGenerator* is a JavaBean encapsulation of an XML parser for parsing XML data and producing a DOM tree. *XMLFileGenerator* encodes a DOM tree into an XML file. The *XMLStringGenerator* encodes a DOM tree as a String. *Nodearray* is a container to store a set of DOM nodes and provide various operations on the node set.
- *XMLViewer*: This set consists of *visual* beans (e.g., *XMLTreeView*, *XMLSourceView*, *XMLNodeListView*, *DTDSourceView*, and *XMLAttributeView*) to display XML documents or DTDs in various ways. The *viewer* beans are essentially Swing components and thus provide support for the multiple look and feel.
- *XMLEditor*: The *editor beans* can be used with AWT, Swing or any GUI components. The nonvisual operator beans allow for constructing DTD-directed editors. The XML editors can be formed by wiring the editor beans with GUI component. The XMLEditor beans can pick the DOM tree apart and use its pieces.
- *XMLProcessing*: This bean set contains *nonvisual* beans (e.g., *XMLSearch*, *XMLFilter*, *XMLTokenizer*, *ElementSelector*, and *AttributeSelector*) to provide searching/filtering XML documents, tokenizing DOM nodes to strings, and selecting specific elements/attributes. Once an XML document has been parsed into a DOM tree, the XMLProcessing beans handle the processing of XML.
- *XMLConvenience*: This bean set simplifies the creation of XML GUI editors. By combining *XMLEditor* beans and *java.awt* GUI objects, the beans can implement common XML editing subfunctions. Therefore, the XMLConvenience beans encapsulate the functionality of the XMLEditor beans and corresponding AWT components. These beans reduce the complexity of wiring in *Integrated Development Environments* (IDEs).

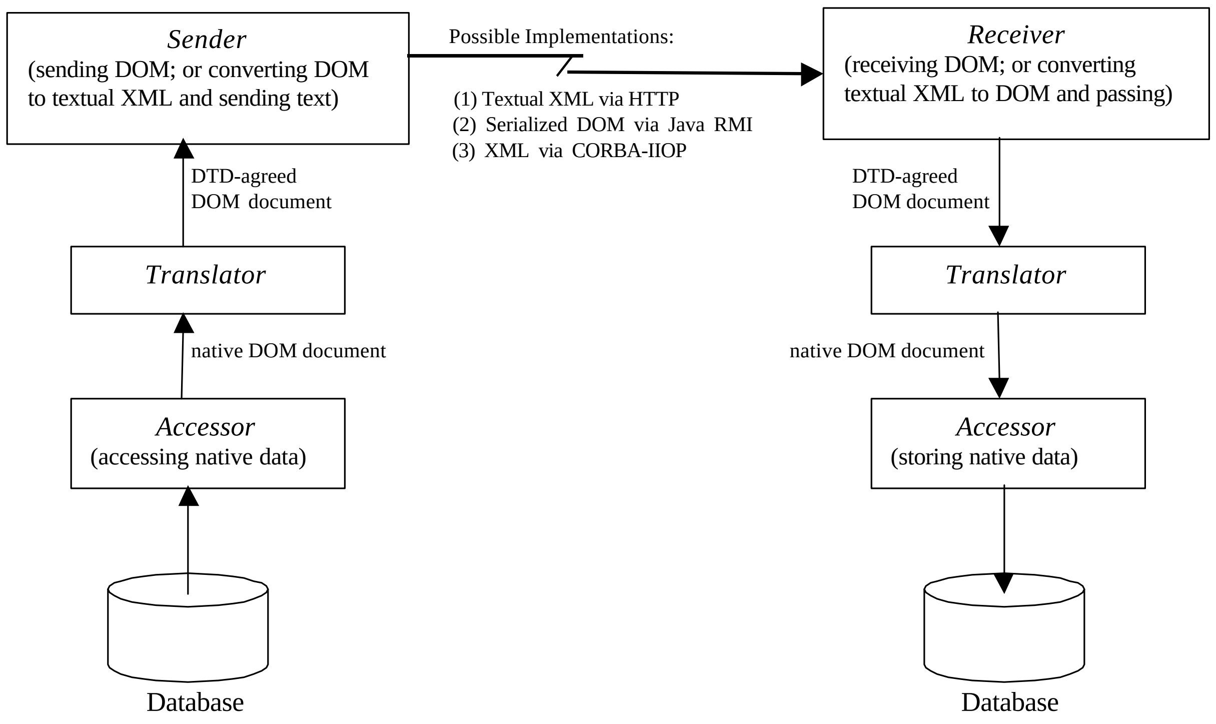

Figure 3. Xbeans for Data Exchange between Enterprises

Xbeans

The *Xbeans* (from Xbeans.org) are JavaBeans components for processing XML documents. An Xbean is a software component that takes XML as input, processes it in some fashion and then passes XML on to the next Xbean [18]. The IBM's *XML Productivity Kit for Java* (XPK4J) inspires the Xbeans. The XPK4J contains two components [5]: *XPK4J JavaBeans* (connecting XML processing beans in a visual builder such as VisualAge for Java), and *XBeans* (allowing Java developer to create JavaBeans from DTDs). However, the Xbeans are different from XPK4J. The Xbeans are focused on the *distributed* applications, whereas XPK4J is mostly used for *non-distributed* GUI applications.

Now let's look at creating a *distributed application* (i.e., communicating and processing XML between distributed computers) using Xbeans. Figure 3 presents Xbeans' functionality for data exchange between enterprises. Xbeans include *accessor*, *translator*, *sender*, and *receiver*. The following lists the salient features of *Xbeans* [12, 13]:

- *Accessor*: It is important to know that the Xbeans consume and produce XML as *DOM* documents (i.e., DOM representation of XML document). An already parsed document object is accessed via the DOM API. In sending side, the accessor Xbean performs a particular SQL query (for accessing native data) and represents the result as a native DOM document. In receiving side, the accessor Xbean is configured with an SQL query to store the incoming native data.
- *Translator*: The XML *DTD* represents the semantics and format of the data to be exchanged. Since no two enterprises represent the same data in the same way, it is necessary to have a translator for converting native data according to a standard DTD. In sending side, the translator Xbean translates the incoming native DOM document into a DOM document conforming to the standard DTD (agreed upon DTD) for exchanging data. In receiving side, the translator converts this DTD-agreed DOM document back to native DOM document.
- *Sender* and *Receiver*: These Xbeans are configured for cooperatively *transporting* the XML data over a network. There are three different implementations of the sender-receiver: *using a standard Web server* (i.e., sending XML text via HTTP), *communicating a serialized DOM representation via Java RMI*, and *using CORBA-IIOP as a transport* (i.e., transmitting XML via CORBA-IIOP). In using a standard Web server, the sender converts DOM to textual XML, and the receiver converts textual XML back to DOM.

Sun's Java *Remote Method Invocation* (RMI) is used for Java-to-Java communications across Virtual Machines. *Internet Inter-ORB Protocol* (IIOP) is a messaging protocol for communication between network-based client/server software programs and enterprise applications based on OMG's *Common Object Request Broker Architecture* (CORBA) standard. IIOP provides interoperability for the *Object Request Brokers* (ORBs) from different vendors to interoperate over the Internet.

3. CONCLUSIONS

We have seen the XML technology and JavaBeans components are complementary. XML provides many potential advantages such as inline reusability, portability of data, customized presentation of data, powerful hypertext linking capabilities, integration of data and metadata, human readability, and more. JavaBeans component model provides a way of developing and using Java components on the client side. JavaBeans are reusable components to support the packaging, reuse, connection, and customization of Java code.

The *BML* is an XML-based JavaBean component configuration or wiring language for describing the creation, configuration, and interconnection structure of a set of JavaBeans. There are two BML processor components, including BML player and BML compiler, to process documents conforming to the BML DTD. The *XML BeanMaker* is used to generate JavaBeans out of XML DTD files. The *XML Bean Suite* is a toolkit of JavaBeans to process XML. It provides a comprehensive set of functionality to manipulate XML content. The *Xbeans* are JavaBeans to process XML documents on distributed applications. They provide a powerful paradigm for data exchange between enterprises.

REFERENCES

[1] D. D'Souza, "JavaBeans: Coding and Design," *Journal of Object-Oriented Programming*, January 1998, pp.14-16.

[2] C. F. Goldfarb and P. Prescod, "*The XML Handbook*," 2nd ed., Prentice-Hall, Upper Saddler River, New Jersey, 2000.

[3] IBM, Inc., "*XML Beans*," http://www.alphaworks.ibm.com/alphaBeans, 2000.

[4] IBM, Inc., "*XML BeanMaker*," http://www.alphaworks.ibm.com/aw.nsf/techmain/xmlbeanmaker, 1999.

[5] IBM, Inc., "*XML Productivity Kit for Java*," http://www.alphaworks.ibm.com/aw.nsf/techmain/xmlpr oductivity, 1999.

[6] M. Johnson, "Cover Story: Bean Markup Language, Part 1, Learn the ABCs of IBM's Powerful JavaBeans Connection Language," *JavaWorld*, http://www.javaworld.com/javaworld/jw-08-1999/jw-08-beans_p.html, August 1999.

[7] M. Johnson, "Bean Markup Language, Part 2: Create Event-Driven Applications with BML," *JavaWorld*, http://www.javaworld.com/javaworld/jw-10-1999/jw-10-beans_p.html, October 1999.

[8] M. Johnson, "Process XML with JavaBeans, Part 1: Interconnect JavaBeans to Process XML," *JavaWorld*, http://www.javaworld.com/javaworld/jw-11-1999/jw-11-beans_p.html, November 1999.

[9] M. Johnson, "Process XML with JavaBeans, Part 2: How IDEs Internnect Components," *JavaWorld*, http://www.javaworld.com/javaworld/jw-12-1999/jw-12-beans_p.html, December 1999.

[10] M. Johnson, "Process XML with JavaBeans, Part 3: Simplify XML Processing with XMLConvenience Beans," http://www.javaworld.com/javaworld/jw-01-2000/jw-01-beans_p.html, *JavaWorld*, January 2000.

[11] D. Krieger and R. M. Adler, "The Emergence of Distributed Component Platforms," *IEEE Computer*, March 1998, pp.43-53.

[12] B. Martin, "*Creating Distributed Applications Using Xbeans*," http://www.xbeans.org/whitepapertxt.html, 2000.

[13] B. Martin, "Build Distributed Applications with Java and XML: Use Xbeans to Process Your XML as DOM Documents," *JavaWorld*, http://www.javaworld.com/javaworld/jw-02-2000/f_jw-02-ssj-xml.html, February 2000.

[14] H. Maruyama, K. Tamura, and N. Uramoto, "*XML and Java: Developing Web Applications*," Addison-Wesley, Reading, Massachusetts, 1999.

[15] E. Pelegri-Llopart and L. P. G. Cable, "*How to be a Good Bean*, " Sun Microsystems, Inc., http://java.sun.com/products/javabeans/docs/goodbean.pdf, September 1997.

[16] Sun Microsystems, Inc., "*JavaBeans*," http://java.sun.com/beans, 2000.

[17] S. Weerawarana and M. J. Duftler, "*Bean Markup Language*," IBM, Inc., http://www.alphaworks.ibm.com/aw.nsf/techmain/bml, 1999.

[18] Xbeans.Org, "*Xbeans: Frequently Asked Questions*," http://www.xbeans.org/faq.html, 2000.