JavaBeans and Software Architecture

Nenad Medvidovic, Ph. D.
Assistant Professor
SAL 338, Department of Computer Science
University of Southern California
Los Angeles, CA 90089, USA
Phone: +1-213-740-5579
Fax: +1-213-740-4927

Nikunj R. Mehta, M. S.
Ph. D. Candidate
SAL 327, Department of Computer Science
University of Southern California
Los Angeles, CA 90089, USA
Phone: +1-213-740-6504
Fax: +1-213-740-4927

OUTLINE

1 Introduction 6
JavaBeans 8
Usage and Applications 10

2 JavaBean Characteristics 11
Methods 11
Events 13
Properties 14
Persistence 17
Packaging 18
Introspection 19
Customization 22
Distributed JavaBeans 23

3 Enterprise JavaBeans 24
Characteristics 25
Internal Architecture 26
Comparison with JavaBeans 28

4 JavaBeans in Other Java technologies 29
Java Abstract Windowing Toolkit (AWT) 30
Java Database Connectivity (JDBC) 30
JavaMail 31
Java Management Extensions (JMX) 31

5 Java and Architecture-Based Software Development 32
Concepts of Software Architectures 32
Support for Architecture in JavaBeans and Related Technologies 33

6 Summary and Conclusions 35

GLOSSARY

Architecture-based software development:

An approach to software development where the architecture of the software system forms the blueprint for its development and evolution

Architecture style:

A set of rules that describe or constrain the organization of components and connectors and the manner in which components interact

Asynchronous:

A mode of interaction where the initiator continues with its processing while the target of interaction is processing the interaction request

Container:

A software program or runtime environment that provides necessary infrastructure facilities and services such as communication, multi-threading, security and transactional isolation to other software systems

Deployment:

The process whereby software is installed into an operational environment

Design-time:

The state and behavior of a software component while it is being composed in a system

Enterprise application:

An application that comprises an enterprise's systems for handling company-wide information

Event:

An occurrence of an incident of interest to a software system

Listener:

A software component that receives and processes events

Middleware:

Middleware is software used to provide standard interface to low-level operating system and network services, and a runtime environment for deploying components.

Naming pattern:

A naming convention for classes, fields or methods followed in order to enable a systematic discovery of functionality

Notification:

The communication of an event of interest to a listener

Observer:

A software component which expresses interest in one or more events

Property:

An attribute of a software component that can be read and/or written

Run-time:

The state and behavior of a software component when the component is executing

Software Architecture:

The overall design of a software system expressed in terms of coarse-grained computational and data components, and connectors used for interaction between the components

Software component:

An element of a software architecture that performs processing and records state

Software Connector:

An element of a software architecture that performs communication, coordination, facilitation, and conversion between software components

Synchronous:

A mode of interaction where the initiator halts processing while the target of interaction performs processing

Wizard:

A tool that simplifies software development by guiding a designer through a well-defined sequence of steps

ABSTRACT

Java has emerged as a popular programming language and platform for Web applications. JavaBeans™ defines the software component model in the Java programming language that is used for creating reusable, coarse-grained components. Beans are used in numerous complementary technologies including Enterprise JavaBeans, Java Abstract Windowing Toolkit (AWT), Java Database Connectivity (JDBC), Java Mail, and Java Management Extensions (JMX) to create software components. Beans provide a hybrid of object-oriented and loosely-coupled architectural styles, where the interaction between components is in the form of both events and method calls. This chapter focuses on the support for component technology in Java, specifically in the form of JavaBeans and its technology variants that fulfil the needs of Web application development. The chapter also discusses the role of JavaBeans and its technology variants in architecture-based software development.

INTRODUCTION

A number of techniques have recently emerged to address the problem of consistently engineering large, complex software systems. The three most widely embraced efforts are component-based software development standards such as COM (Brockschmidt, 1994; Sessions, 1997) and JavaBeans (Hamilton, 1997; DeMichiel, 2002), middleware platforms such as CORBA (Orfali, Harkey & Edwards, 1996) and .NET (Microsoft, 2001), and software architecture (Perry & Wolf, 1992). These three approaches are complementary to each other, and a judicious mix of the three is often employed in developing large, complex systems such as those targeted for the Internet.

Java is a popular programming language increasingly being used for component-oriented and architecture-based software development for the Web. The acceptance of Java technologies in

both industrial and academic circles can be attributed to both Java language features and the Java platform. While the Java platform has been designed to support the paradigm of "write once, run anywhere", the Java programming language has been designed to reduce sources of common programming mistakes, and provide greater control over the organization of software systems (http://java.sun.com). Java's support for organizing an application from coarse-grained components has an added benefit. It results in greater acceptance and use of the Java programming language and platform in Web-based applications. This chapter focuses on the support for component technology in Java – JavaBeans – and various accompanying technologies, such as a web server and an application server, that utilize JavaBeans to fulfil the needs of Web application development.

Support for component-oriented software development in Java has gradually improved from the early days of Java to the present. The earliest manifestation of Java's suitability for organizing applications around components was the JavaBeans framework, first released in late 1996, as a software component model for Java (Hamilton, 1997). JavaBeans are aimed at creating components that can be reused and integrated into bigger applications. Beans follow specific conventions and provide a loosely-coupled interaction mechanism based on events. Beans also provide design and run-time facilities such as introspection, customization, persistence, and security. Support for JavaBeans is provided in the form of standard libraries packaged by the Java runtime environment in the *java.beans* package. Beans are commonly used for developing graphical user interface (GUI) widgets as well as non-graphical elements (e.g., data structures) that provide processing support for the GUI widgets.

The rest of this chapter is organized as follows: Section 2 introduces the syntactic and semantic constructs of JavaBeans and its use in developing and delivering components. Section 3

focuses on the use of JavaBeans for distributed applications in the form of Enterprise JavaBeans (EJB). JavaBeans are also widely used in other Java technologies, and Section 4 discusses major Java Application Programming Interfaces (API) that are built on top of JavaBeans for developing Internet applications. Section 5 presents a discussion on the use of JavaBeans in component-based software development and its relationship to software architecture. Section 6 summarizes the JavaBeans technology and its role in software architectures.

JAVABEANS

Java has emerged as a widely-used object-oriented programming language, and its successful use can be attributed to its support for a variety of technologies that are used to solve specific infrastructure and application requirements. The JavaBeans technology has enabled large-scale, coarse-grained reuse of components, and has influenced a number of other Java technologies. JavaBeans is the technology that enables the construction of visual components as well as the binding of those components into an GUI application using a visual design tool such as the one shown in Figure 1. This tool supports “drag and drop” design of a GUI application composed from smaller visual components called GUI widgets. Such a tool is capable of generating Java source code for the application based on the visual design of the application. An application programmer may then modify the source code to provide additional details of the application logic.

The JavaBeans API was originally proposed as a software component model for Java (Hamilton, 1997) chiefly to allow users to manipulate and connect widgets in a GUI. However, applications in many other domains can also benefit from the loosely-coupled architecture and support for introspection, customization and persistence. For example, JavaBeans are used in mail clients to manage connections to a mail server and manipulate the contents of one’s mailbox.

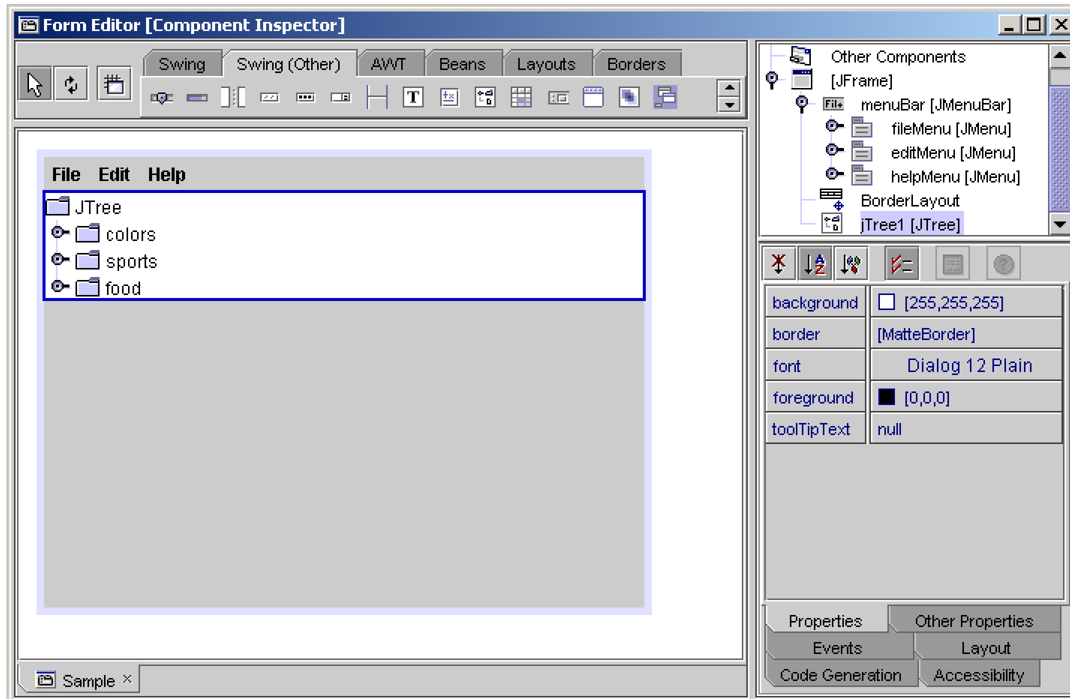

Figure 1 NetBeans IDE 3.4 Visual Form Editor

In the original JavaBeans specification (Hamilton, 97, p.9), a JavaBean is defined as "a reusable software component that can be manipulated visually in a builder tool." This definition matches the goal of JavaBeans, which is to create a software component model. In order to realize the goal, Java defines an API, programming conventions and *naming patterns* to promote component-oriented application development. Nevertheless, JavaBeans adhere to the same rules as those followed by any Java class at the level of source code as well as the byte code produced. A single JavaBean can be implemented as a collection of classes all working together to provide a unified set of functions. This enables the creation of coarse-grained components out of a set of Java classes.

Usage and Applications

JavaBeans provide separate interfaces for design-time and run-time capabilities. The design-time configuration and customization capabilities supported in JavaBeans set them apart

from regular Java classes and class libraries. Some JavaBeans are GUI elements such as windows and buttons, some are more complex visual elements such as database viewers or spreadsheets, still others are non-visual components such as database connections and algorithms. Java AWT is the earliest library of GUI widgets that employ JavaBeans. AWT was aimed at supporting the development of richer user interfaces for the Web using applets. AWT only provides simple widgets such as button, panel, and window, and has serious performance drawbacks due to its use of heavyweight components for every AWT widget. To overcome the limitations of AWT, the Java Foundation Classes (JFC) library was introduced with a much richer set of widgets (Geary, 99). Moreover JFC, also known as Swing, uses an improved lightweight architecture to deliver better performance by using far fewer components per widget.

Beans are primarily targeted for use in visual builder tools such as Web page builders, integrated development environments (IDEs), visual application builders, GUI layout builders, and server application builders, although JavaBeans are entirely human-programmable as well. In this context, it is necessary to understand the distinction between the two sets of interfaces offered by JavaBeans. Design-time capabilities allow a JavaBean to run inside a visual builder tool and provide design information about itself to the tool so that a designer can customize the bean. Run-time capabilities allow a JavaBean to manifest its behavior in an application so that the end-user can use its functionality. The separation of design-time from run-time capabilities enables richer visual editing of the design, while not burdening the eventual application with redundant design logic.

JAVABEAN CHARACTERISTICS

A JavaBean does not extend any standard class, nor implement any standard interface, although visual components always extend *java.awt.Component* or its subclasses. However,

JavaBeans typically share a set of characteristics in the form of a standard set of facilities that enable beans to be easily combined into applications. These facilities in order of usage from design- to run-time are: *introspection*, *customization*, *properties*, *events, persistence*, *packaging*, and *methods*. Introspection and customization are used at design-time; properties, events, persistence, and packaging are used at both design and run-time; methods are used exclusively at run-time. We discuss these facilities in the reverse order, that is, from run-time to design-time, to allow the reader to grasp the significance of the design-time facilities after understanding the utility of the run-time facilities. Figure 2 shows a sample JavaBean called *Circle*, which is used as an example for the rest of this section.

Methods

A JavaBean provides methods for services it exposes. Methods are defined for use at run-time when the application wishes to use the bean's services. They provide a low-overhead, synchronous communication mechanism for components in the JavaBeans component model. JavaBean methods are defined as Java methods with the *public* modifier which makes them available to other components in the application. Other method modifiers such as *final* and *synchronized* can be used with JavaBean methods. JavaBean methods follow the same lexical and syntactical rules as those followed by Java methods, and have the same invocation semantics and mechanisms as available to regular Java methods. However, not all methods defined in a JavaBean are considered to be JavaBean methods. This distinction is necessary to distinguish high-level services offered by the bean from low-level implementation mechanisms such as Java declarations, without burdening the language with unnecessary keywords and concepts.

Any public method defined with a name that does not start with the reserved prefixes *get*, *is, set*, *add*, and *remove,* is considered to be a JavaBean method. The reserved keywords are used

in conjunction with JavaBean properties and events as explained in subsequent sections. As shown in Figure 2, *Circle* is a visual component that can be used as a widget for drawing circular shapes of all sizes on a graphical window using the method called *paint*. Other Java methods defined in this class are not considered JavaBean methods.

```
public class Circle extends java.awt.Component implements java.io.Serializable
                  {
   ...
   // Paint the circle on the screen
    public void paint(Graphics g)
   ...
   // Get the boolean Shown property
   public boolean isShown()
   ...
   // Get the Radius property
   public int getRadius()
   ...
   // Set the Radius property
   public void setRadius(int radius) throws PropertyVetoException
   ...
   // Get the Border Size property
   public int getBorderSize()
   ...
   // Set the Border Size property
   public int setBorderSize(int size)
   ...
   // Add an event listener for change of radius
   public void addRadiusChangeListener(RadiusChangeListener l)
      throws java.beans.TooManyListenersException
   ...
   // Remove an event listener for change of radius
   public void removeRadiusChangeListener(RadiusChangeListener l)
   ...
   // Add an event listener for change of any property
   public void addPropertyChangeListener(PropertyChangeListener l)
   ...
   // Remove an event listener for change of any property
   public void removePropertyChangeListener(PropertyChangeListener l)
   ...
   // Add an event listener for a vetoable change of any property
   public void addVetoableChangeListener(VetoableChangeListener l)
   ...
   // Remove an event listener for a vetoable change of any property
   public void removeVetoableChangeListener(VetoableChangeListener l)
```

Figure 2 Circle – A JavaBean

<u>Events</u>

Events are the occurrence of an incident of interest to components. Events are used in JavaBeans as a publish-subscribe based communication mechanism. Figure 3 graphically represents the JavaBean event architecture, which is based on the Java event model (Englander, 1997). Any object, not necessarily a JavaBean component, registers with beans that generate events that are of interest to it. Each interested object, called an event listener, makes a call to the *subscription management* interface of a JavaBean for registering an interest in events. The JavaBean records all listeners for each event category. Later, when an event occurs, the bean sends the event message, an instance of *java.util.EventObject* or one of its subclass, to all registered listeners of that event as a method call.

A single JavaBean may provide a number of events defined through methods that follow a certain naming pattern: subscription for an event is performed by calling methods of the form *public void add<EventName>Listener(<EventName>Listener l)* and unsubscription is performed using methods of the form *public void remove<EventName>Listener(<EventName>Listener l).* These methods together define the *Subscription Management* port shown in Figure 3. An example

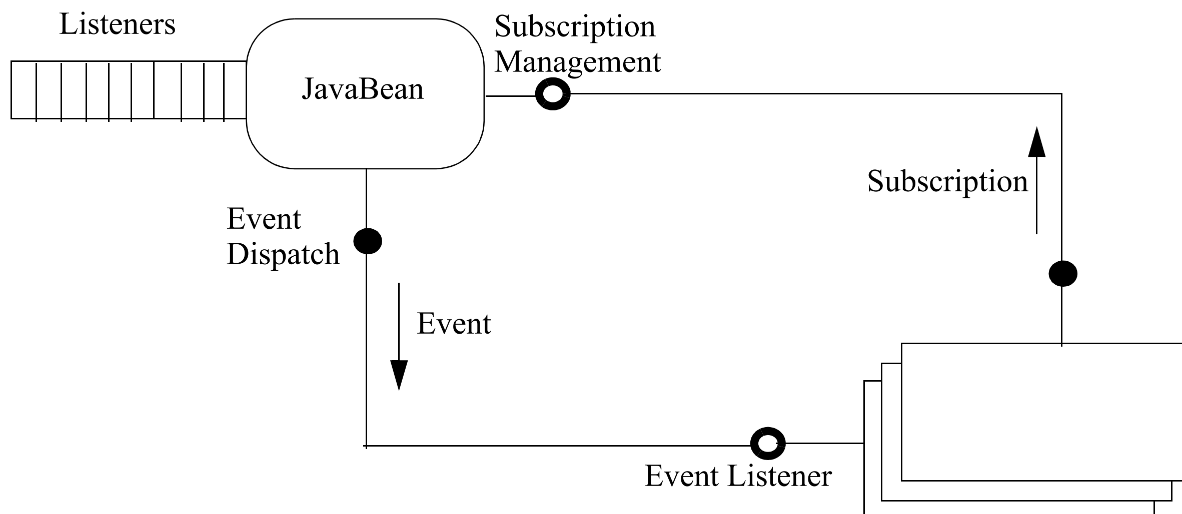

Figure 3 JavaBeans Event Model

of this naming pattern can be seen in Figure 2 where the JavaBean *Circle* defines a *RadiusChange* event.

This subscription management port allows multiple subscribers to be registered for a single event. Sometimes it is necessary to constrain the number of listeners, and a common use is to perform unicast event notification, i.e., notification to only one listener. In order to support this, a bean keeps the number of listeners for the event to 1 or less. While one listener is registered for the event, if another listener tries to register for the same event, the bean can throw a *java.beans.TooManyListenersException* indicating that the requested event subscription cannot be allowed. An example of unicast events is the *radius change* event as shown in Figure 2.

Properties

Properties provide access to the state of a JavaBean that can be read and/or changed, and have a name and a type. Properties allow a builder tool to access the state of a JavaBean for customization, as well as an application to modify the behavior of a bean by changing its state. Thus properties are used at design-time as well as run-time. Properties are defined as a set of methods in a JavaBean instead of being defined as public fields. This enforces encapsulation by hiding away the fields defined in the Java class implementing the JavaBean, and instead exposing only the bean's high-level properties.

A naming pattern is used to identify properties of a JavaBean. Since all properties are accessed through method calls, special prefixes *get*, *is,* and *set* are used to identify the methods used for property access. For example as shown in Figure 2, the Circle bean has a property named *BorderSize* of type *int* that is accessed through methods *getBorderSize* and *setBorderSize* for reading and changing the property, respectively. Boolean properties such as *Shown* from Figure 2 are read using the *is* prefix i.e. as *public boolean isShown()*.

JavaBeans support both single- and multi-valued properties. Single-valued properties are accessed through simple accessor methods, whereas a multi-valued property, also called *indexed* property, requires that an index be provided in the accessor methods. For example, *public Color getColor(int index)* would be the signature of a getter for the indexed property *Color*, whereas *public void setColor(int index, Color value)* would be the signature of a setter for the same property. A property can be either read only, write only or read/write, depending on which of the accessor methods are defined in the bean. Moreover, based on how the bean responds to a change in the value of the property, three kinds of properties—*simple*, *bound*, and *constrained*—are possible.

In the case of a simple property, the JavaBean unilaterally changes the property value without alerting other components that depend on the property. Bound properties are used when the bean is required to notify any interested components of the change in the value of one or more properties. Interested components are notified through an event of class *java.beans.Property-ChangeEvent*. Such components may respond to the event in a coordinated manner to maintain a stable application state e.g., by updating their own properties, changing the properties of other beans, or calling a method on a bean. Although a change made to any property can result in an event, those properties for which a change in value causes the bean to generate a PropertyChangeEvent are considered as bound properties. A bean should change the value of its property before firing a PropertyChangeEvent to ensure that components receiving the events do not find an inconsistency between the event and the state of the bean. It is possible to register interest for changes to any property of a bean or to individual properties. This is accomplished by providing two sets of methods for registering listeners, one called *addPropertyChangeListener(Property-ChangeListener p)* for registering interest in all bound properties of the bean, and another of the

form *add<Property Name>Listener(PropertyChangeListener p)* for registering interest in a specific bound property of the bean.

Simple and bound properties do not constrain changes to their values to ensure consistency between beans. Sometimes, it is necessary to do so when combining beans together so that those property values that are unacceptable to listeners monitoring these properties can be rejected. In the case of constrained events, the bean notifies listeners using a *java.beans.VetoableChangeEvent* to request their approval for a change in the value of a property before the change can be applied. If even one such component rejects the change by throwing a *java.beans.PropertyVetoException* while processing the event, the new property value is disallowed. Accordingly, the accessor methods for a constrained property follow a slightly different naming pattern in that the setter method is defined to throw a *java.beans.PropertyVetoException*. A bean persists the changed value only if it receives the approval of all interested components, and then sends a *PropertyChangeEvent* once the property value is changed. This two phase protocol allows for distributed approval to take place without causing transient side effects. It is possible to register interest for changes to any vetoable properties of a bean as well as to a specific property. This is accomplished by providing two sets of methods for registering listeners, one called *addVetoableChangeListener(VetoableChangeListener v)* for registering interest in all constrained properties of the bean, and another of the form *add<Property Name>Listener(VetoableChangeListener v)* for registering interest in a specific constrained property of the bean. Figure 2 shows an example of a specific constrained property called *Radius*. When placed with other widgets on a window, the radius of a circle might need to be acceptable to other widgets present in the same window. The vetoable property *Radius* allows for a check to be

made with the neighbors of a *Circle* object in the window to ensure that a change in its radius does not consume space occupied by another widget.

Persistence

A JavaBean may store information internally in order to perform its behavior. Some of this information may be accessible as properties and other information may be private to the bean. In order to recreate an instance of a bean, it is necessary to record and restore this internal information. This information management behavior is called persistence. JavaBeans provide persistence of their state so that a customized bean can be recreated later with the same set of customized properties. JavaBeans supports persistence through three mechanisms *Serialization*, *Externalization*, and *XML Encoding*.

The first mechanism, Serialization, is based on the use of a standard Java interface namely, *java.io.Serializable*. Implementing this interface in any Java class will generate additional *byte code* for persistence when the class is compiled. This special byte code is capable of recording all the internal fields including the non-primitive fields of the class that implement the Serializable interface. The Java Virtual Machine (JVM) serializes the super classes and subclasses of the bean, if any. A bean can exert more control over the manner in which its declared fields are persisted by implementing special methods *readObject* and *writeObject*. An object can be *serialized* to any stream format such as a disk file or a network connection, and *deserialized* when the stream is read, thus communicating the design information to an application at run-time. Deserialization offers a more flexible and preferable way of instantiating beans compared to regular constructor-based instantiation as the exact set of properties of a customized bean can be restored.

In addition to the standard serialization, Java also supports *externalization*, which gives a JavaBean complete control over the amount of and manner in which internal information of the

bean is serialized using standards such as compound documents (Brockschmidt, 1994). A Java class must implement *java.io.Externalizable* interface to use externalization, which in turn requires the class to provide definitions of methods *readExternal* and *writeExternal,* both used with binary object streams.

With Java Development Kit (JDK) 1.4, Java also provides a more resilient persistence mechanism in the form of *java.beans.Encoder*. This mechanism only persists properties of a bean instead of all the internal fields of the bean's Java class. This mechanism is especially used with XML as provided in the *java.beans.XMLEncoder* which records the properties textually instead of the binary formats that are used in serialization and externalization. Since this technique does not depend on any private class information, this technique is more resilient as it can survive changes in the JVM as well as versions of the bean; it is more robust as a partial damage to the API of the bean can still allow remaining properties to be read; and it is more compact as default properties of a bean can be safely removed from the persistent form of the bean. The encoder is responsible for traversing the entire graph of properties used in a bean as well as for structuring and recording the information about each bean in this graph into a single stream. A counterpart of this encoder, the *java.beans.XMLDecoder* is available to deserialize this graph of beans from its textual persistent form. While the binary representation is more useful for inter-process communication, the text form is preferred for archival purposes.

Packaging

JavaBeans use the Java Archive (*Jar*) file mechanism for packaging multiple bean classes, other associated classes, and serialized beans that can be distributed with the application for use in a visual builder tool. The Jar file uses the ZIP format which has the benefits of compression as well as support on a large number of platforms. Jars also contain a manifest file, which is used to

record additional information about the contents of the Jar such as the versions and names of JavaBeans. Jar files are often used to record other information in the form of graphic files, text resources and documentation about the bean, thus providing a manageable means of distributing JavaBeans. At design-time Jar files can be created from the beans used in the design, which can then be deployed at run-time to produce executing instances of the same beans using deserialization mechanisms discussed in the previous section.

Introspection

Introspection is the ability to acquire design information about the bean so that a visual builder tool can assist a designer in using the bean to compose applications or a composite JavaBean. In the process of introspection, a visual builder tool can discover the customization and configuration capabilities of the bean. Introspection involves providing explicit high-level information about the bean that can help a designer use it. Each JavaBean that supports introspection provides a class that implements the *java.beans.BeanInfo* interface. With beans that do not provide or omit certain high-level introspection information through the BeanInfo interface, a builder tool derives this information by using the Java Reflection APIs and applying the JavaBeans naming conventions to discover properties, events, and methods.

As shown in Figure 4, the bean customization and functional behavior is separated from the introspection behavior. Since introspection behavior is only required during design, such a division of behaviors results in light weight beans that can be employed at run-time without any need to manage large amounts of introspection data about the bean. The names of the two classes (the bean and its BeanInfo) are coordinated by a *naming pattern*. According to this naming pattern, the names of these two classes should only differ by the suffix BeanInfo used for the introspection class. Thus a JavaBean by the name *FooBean* will provide at least two classes,

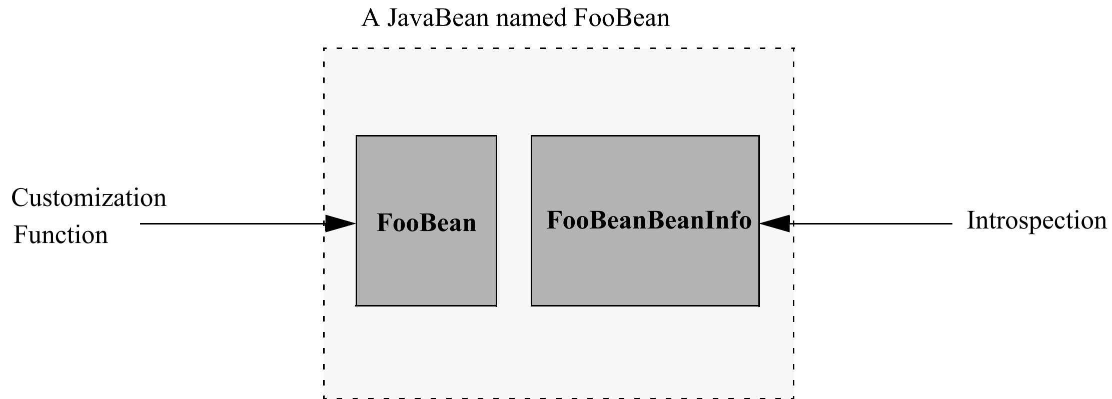

Figure 4 Separating JavaBean behaviors

FooBean and *FooBeanBeanInfo*. The BeanInfo interface provides the following information about the bean:

- A set of descriptors for the events emitted by the bean
- A set of descriptors for the properties supported by the bean
- A set of descriptors for the methods supported by the bean
- An icon for the bean
- A descriptor for the bean specifying the class for the bean as well as the class for the customizer of the bean (JavaBean *customization* is explained in the next section)
- Introspection information about other beans that are relevant to this bean

A default implementation of this interface is provided in the Java API class *java.beans.SimpleBeanInfo*, which defines a default behavior for each of the methods in the BeanInfo interface. This allows a developer of the JavaBean to avoid implementing those methods in the interface that are not relevant while specifying relevant introspection behavior in a class that extends the *SimpleBeanInfo*. Figure 5 gives an example of a BeanInfo implementation for the Circle JavaBean introduced in Figure 2. More information about the JavaBeans API for

```
public class CircleBeanInfo extends java.beans.SimpleBeanInfo {
   public PropertyDescriptor[] getPropertyDescriptors() {
      PropertyDescriptor[] result = null;
        try
        {
         PropertyDescriptor pd1 = new PropertyDescriptor("radius", int.class);
         PropertyDescriptor pd2 = new PropertyDescriptor("borderSize",
                  int.class);
         result = {pd1, pd2};
      }
      catch (Exception e)
      {
         System.err.println("CircleBeanInfo: unexpected exception " + e);
      }
      return result;
   }
   public EventSetDescriptor[] getPropertyDescriptors() {
      EventSetDescriptor[] result = null;
      try
      {
         EventSetDescriptor ed1 = new EventSetDescriptor(Circle.class,
            "radiusChange", RadiusChangeListener.class, "radiusChanged");
         result = {ed1};
      }
      catch (Exception e)
      {
         System.err.println("CircleBeanInfo: unexpected exception " + e);
      }
      return result;
   }
}
```

Figure 5 Circle JavaBean Introspection

introspection can be found in several elaborate references on this topic (Hamilton, 97; Englander, 97).

Visual builders use a JavaBeans utility, *java.beans.Introspector*, to locate the introspection information about a bean. The Introspector then locates the BeanInfo class for the given bean using the BeanInfo naming pattern. The Introspector then constructs the BeanInfo using the beans introspection class, and filling in the missing aspects of the introspection through reflection-based

analysis of the bean. Thus, introspection allows users of a bean to obtain high-level information for using the bean.

Customization

A visual builder is used to design an application that uses JavaBeans. The introspection facilities of a JavaBean discussed in the previous section enable the builder to understand the structure of beans. A builder uses this information to create suitable tools for manipulating the properties of beans as well as for providing visual techniques for connecting various beans and defining their interaction. For simple properties such as numbers and text, simple text editors are sufficient tools for specifying the values of properties. One of the immediate benefits of using a visual builder is the ability to specify sophisticated properties of a bean, such as colors and font types, more naturally using intuitive techniques present in rich *property editors* such as a color palette. A bean can also provide its own property customization user interface to give it complete control over how a designer provides values for its properties.

Property editors are used only at design time to capture properties of a bean. The property editors to be used for manipulating each property can be specified in the introspection information about that property. Since it is not necessary for the bean to provide introspection information about all its properties, nor to provide a custom editor for any of the properties, the development tool automatically selects an editor for properties based on the type of the property. Custom property editors can be developed and packaged with the beans, and specified for use in the introspection information. A custom property editor implements the *java.beans.PropertyEditor* interface, which ensures consistent behaviors across all property editors, both tool-provided and custom.

Property editors are used for editing one bean property at a time. On the other hand, a customizer may be provided for customizing the entire bean. A bean customizer should implement the *java.beans.Customizer* interface. When a customizer is provided for a bean, it takes over all aspects of customization of the bean, which could be a super set of the properties exposed by the bean. Introspection of a bean reveals the customizer to be used for a bean. More details about customization may be found in (Englander, 97).

Distributed JavaBeans

When an application consists of components distributed over a network, serialization is used in combination with Java Remote Method Invocation (RMI) (Sun Microsystems, 2001) to transport methods, events and JavaBean instances over the network. Java RMI is a protocol that allows methods of a Java object to be remotely invoked from another Java virtual machine, possibly residing on a different host across the network. RMI is used in conjunction with Java Naming and Directory Interface (JNDI) to lookup registered instances of JavaBeans that are hosted remotely. Once the required instance of a JavaBean is located using JNDI, a client of that instance can make a method call using RMI. Parameters passed in the method are serialized using the serialization mechanism described earlier in this section. The method is executed remotely, and results of the execution as well as any exceptions are returned to the caller. RMI enables object-oriented invocations of methods without the need to implement low-level data and network communication logic.

ENTERPRISE JAVABEANS

JavaBeans are commonly used in graphical application development, and this has led to the development of other component models. Another prominent Java component model is Enterprise JavaBeans (EJB), which is targeted at the development and deployment of component-based

distributed business applications (DeMichiel, 2002). EJB derives its name from JavaBeans although EJBs are used for a very different purpose. An EJB is an “enterprise” component, i.e., a component used in large, distributed systems that can be concurrently accessed by a large number of users over a network. The demands of a distributed system used at an enterprise-scale requires that special attention be paid to the architecture of the components.

The EJB component model was developed to address the need for providing data access and information processing to large groups of users. In an enterprise application, data is stored in large databases, with a large number of users simultaneously accessing the data. For business-critical applications, high availability and security are extremely important. Moreover, such applications consist of components that are distributed over a network, further requiring management of resources such as processors, memory, and network communication. The EJB component model aims at simplifying the task of developing, deploying, and maintaining such applications. In the EJB component model a number of lower-level infrastructure services are provided by EJB middleware called an EJB *container* in which the EJB components are deployed. It should be noted that, unlike JavaBeans, EJBs are not visual components, and cannot be visually manipulated by users. Moreover, EJBs do not require customization of behavior, but instead require configuration of resources required by the EJB. The goal of EJB—to simplify distributed application component development—is achieved by providing a number of built-in services in the EJB container.

Characteristics

All Enterprise JavaBeans extend *javax.ejb.EJBObject* and provide implementations of all the abstract methods defined in this class in a *template method pattern* (Gamma, Helm, Johnson & Vlissides, 1996). A standard set of facilities are required in enterprise applications to ensure

efficient usage of resources as well as scalability. These facilities, provided in the EJB container that supports the deployment and execution of EJBs, arranged sequentially in order of usage from deployment- to run-time, are:

- *Deployment* – Classes of an EJB component are packaged in the form of an *EAR* (enterprise archive) file, building on the Jar file format, which contains a *manifest* for describing the contents of the EAR file, and *deployment descriptors* that define the configuration of facilities required by the component.
- *Memory and instance management* – The EJB container performs memory management for EJB components, over and above JVM garbage collection, in order to reduce overheads involved in creating and initializing instances of components, as well as to keep an optimum number of instances available to serve average and peak demands.
- *Thread management* – The EJB container manages the runtime environment of EJB components including the Java threads available for execution.
- *Communication management* – The EJB container provides services over the network to EJB clients, and dispatches calls received to EJB components.
- *Security* – The EJB container enforces security by controlling access to restricted components as well as by performing authentication of clients requiring access to restricted components.
- *Location* – An EJB container enables the distribution of components across the network, with run-time binding of services with their clients, thus enabling load-balancing and failure management.
- *Messaging* – Asynchronous communication between components is supported through message queues that store and dispatch messages with varying levels of delivery guarantees to network destinations.

- *Invocation* – The EJB container invokes available instances of EJB components for services requested by EJB clients.
- *Persistence* – The EJB container ensures durability of component state by entrusting it to external storage systems such as a database management system.
- *Transaction management* – The EJB container supports grouping together multiple database accesses corresponding to a single transaction by defining and managing transaction boundaries.

Internal Architecture

Figure 6 shows the containment relations between the various constituents of the EJB architecture. The EJB architecture consists of three kinds of EJB components namely, session beans, entity beans and message-driven beans, and the EJB container. Session beans provide business logic to clients; entity beans provide an object-oriented representation of persistent application data; and message-driven beans process asynchronous messages. The EJB container is responsible for providing infrastructure services identified in the previous section whereas EJB components provide application services. The EJB container provides two entry RMI-based ports

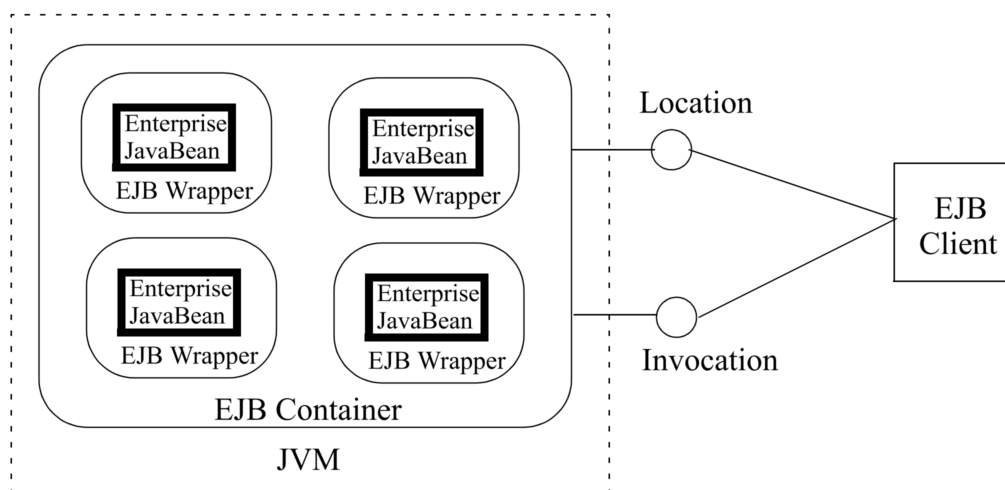

Figure 6 Enterprise JavaBeans Container Architecture

to EJB clients, one each for locating an EJB and for invoking a method on the EJB. In the EJB architecture, a third constituent, an EJB *wrapper*, plays an important though invisible role. The EJB client cannot access an EJB instance directly, but can only do so through the EJB wrapper. The EJB wrapper is a *connector* (Mehta, Medvidovic & Phadke, 2000) between an EJB and its container generated from the deployment descriptor of the bean. This wrapper provides the deployed instances of an EJB with their required services. The EJB wrappers can be generated either within the containers when the bean is deployed, or using special tools when the EJBs are packaged for deployment.

Figure 7 shows the EJB invocation model where EJB clients communicate with the EJB through interfaces rather than directly invoking the implementation class. This separation of interfaces from implementation results in greater encapsulation of the business logic inside the EJB. EJBs have two interaction ports, i.e., access interfaces: *Home Interface* and *Remote Interface*. The Home Interface provides methods for creating new instances of the EJB as well as for locating required EJB instances. In the case of a session bean, the Home Interface is only used to create instances of the EJB, whereas in a message-driven bean this port is not used. In the case of an

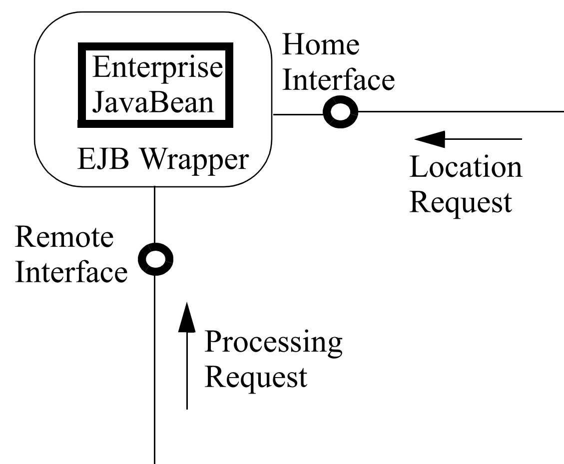

Figure 7 Enterprise JavaBeans Invocation Model

entity bean, the home interface is used to load specific instances of an entity from a persistent data store, such as a relational database, based on the properties of those instances.

The Remote Interface is used by an EJB client for performing business services using a specific EJB instance. A message-driven bean does not require a remote interface since it does not expose any business service except the standard message consumption service. Session and entity beans may expose specific methods in the remote interface to meet application needs.

EJB wrappers generated in an EJB container intercept calls made to the home and remote interfaces and pass the calls to the EJB for processing after fulfilling security, transaction, and instance management. Certain calls can directly be processed by the wrapper including those that involve *container-managed persistence* where the persistence is directly managed by the container based on the deployment descriptor of an entity bean. Others are delegated to the implementations provided in the EJB by its designer.

Comparison with JavaBeans

JavaBeans and Enterprise JavaBeans have emerged as the primary component models of architecture-based software development, each in its own niche. JavaBeans are used for developing desktop applications that provide rich graphical interfaces to users. Enterprise JavaBeans, on the other hand, are used in *back office* applications as distributed components providing business logic, data processing, and persistent data access. JavaBeans serve to simplify development of applications through the use of visual builders that can provide visual means to designers for composing applications. Enterprise JavaBeans, on the other hand, are not necessarily aimed at supporting a visual development environment, although wizard-like tools are often provided to "jump start" their development.

JavaBeans do not require inheritance from a specific class, although visual components are expected to extend *java.awt.Component*. On the contrary, EJBs are always expected to extend *javax.ejb.EJBObject*. JavaBeans can execute in standard JVM environment, whereas EJBs require a container typically provided in a J2EE (Java2 Enterprise Edition) (Shannon, 2002) server. JavaBeans require customization before they can be integrated into an application; the customized instances are recorded as serialized instances which can then be recreated in the application. EJBs on the other hand are customized by configuring the deployment descriptor and repackaging the beans. In both cases, certain customization can be performed even at run-time. Both types of components require standard-based packaging of classes to ensure manageable deployment units, and inter-operability with various vendor implementations of JVM and EJB containers.

JavaBeans support an architectural style of loose component integration and de-coupled interaction in the form of events, whereas EJB interaction takes the form of asynchronous messages and synchronous method calls. JavaBeans provide two ports for interactions: subscription management and event dispatch. EJBs also provide two ports but with different purposes: home interface for location and remote interface for business services.

JAVABEANS IN OTHER JAVA TECHNOLOGIES

JavaBeans define a generic component model that can be used in different domains where Java is used. Since JavaBeans is suited for use in a client environment as explained previously, most uses of JavaBeans and derivative technologies employing them have emerged for the client environment. We present a brief overview of these technologies in the remainder of this section.

Java Abstract Windowing Toolkit (AWT)

Java AWT is part of the standard API for producing GUI applications for Java (Sun Microsystems, 2002a). Java AWT was originally introduced in Java version 1.0.2, and subse-

quently re-engineered to use JavaBeans. AWT beans satisfy the goal of JavaBeans – reusable software components that can be manipulated visually in a builder tool. The *beanified* version of AWT presents GUI widgets (such as buttons, text input controls, check boxes) as JavaBeans replete with properties, methods, and events. AWT widgets can be customized for use in an application using a bean customizer, properties can be edited, and events can be managed both at design-time and run-time. An example of the use of constrained events in AWT is when various beans are placed on a *panel;* if a single widget is resized, then every widget in the panel is alerted to a resulting modification in its size and/or location; if any widget disapproves such a change, the change in the original widget is disallowed. AWT also uses the same event model as employed in JavaBeans to communicate the occurrence of user actions on the GUI widgets as events to the application class handling them.

Java Database Connectivity (JDBC)

JavaBeans can be persisted on file systems using any of the three standard persistence mechanisms, Serialization, Externalization and XML Encoding. However, the standard mechanisms do not offer the same levels of durability and integrity required in many business systems. Such requirements can be met by storing data in a database this is supported in Java with the help of JDBC technology. JDBC provides classes for establishing connections, performing queries, and manipulating result sets of data returned from the database (Ellis & Ho, 2002). JDBC version 2.0 introduced JavaBeans to the existing API by adding events and properties to the low-level data access model.

Two JDBC interfaces primarily employ JavaBean mechanisms *javax.sql.RowSet* and *javax.sql.PooledConnection*. A RowSet can be configured at design-time and used at run-time to retrieve and persist information to and from the database. The RowSet is responsible for creating

its own database connections and performing data access using lower-level JDBC APIs based on specified configuration. It generates a RowSetEvent to indicate the occurrence of three kinds of events: *cursorMoved*, *rowChanged*, and *rowSetChanged*. A PooledConnection is used to make intermittent access to the database more efficient by pooling the use of a lower-level database connection. A ConnectionEvent can be emitted to indicate the occurrence of two kinds of events: *connectionClosed* and *connectionErrorOccurred*. The additions of JavaBeans mechanisms to JDBC enables database connectivity to be used at the level of components manipulated through visual builder tools.

JavaMail

JavaMail is designed to create rich mail client components based on standard mail protocols that can be integrated into applications (Sun Microsystems, 2000). It can be used at design-time to record properties for establishing a connection to the mail store, and at run-time to retrieve information from the mail store. A set of closely related JavaBeans are available from this API: *Message*, *Folder*, *Store*, and *Transport*. Seven sets of events are generated by JavaMail components: *ConnectionEvent*, *FolderEvent*, *MailEvent*, *MessageChangedEvent*, *Message-CountEvent*, *StoreEvent*, and *TransportEvent*. These events are emitted in response to user actions and data received from mail servers.

Java Management Extensions (JMX)

Java Management Extensions API defines an architecture for managing applications and networks (Sun Microsystems, 2002b). Application components and complete applications can be instrumented to obtain information about their functioning, as well as to control their run-time behavior. Components that can be managed need to follow certain naming patterns and implement a *management interface*. An *MBean* (managed bean) is a Java object that implements the

management interface for the component being managed. MBeans can define *meta data* that provide introspection information about the management interface. The MBean instance also needs to register itself with an *MBean server* thus making the bean manageable even from outside the JVM. The management interface contains properties, methods, and notifications available for management in the component. Properties and methods follow the same naming pattern as JavaBeans, while notifications behave identically as JavaBean events.

JAVA AND ARCHITECTURE-BASED SOFTWARE DEVELOPMENT

Software architectures provide high-level abstractions to represent and reason about software systems (Perry & Wolf, 1992). The focus of software architecture is to represent the structure of a system without exposing all the complex details of its implementation. JavaBeans and various Java technologies based on it have developed programmatic means of implementing conceptual software architectures. In this section we provide a brief synopsis of software architectures and discuss how JavaBeans and other Java APIs come together to realize software architectural concepts.

Concepts of Software Architectures

Complementing other approaches to developing large, complex systems, software architectures provide support for composing software systems from coarse-grained *components*. A software component in this sense is an element of a software architecture that performs processing and records state. Another important aspect, particularly magnified by the emergence of the Internet and the growing need for distribution, is *interaction* among components. Component interaction is embodied in the notion of *software connectors*. Components and connectors can be linked together in specific *configurations* to create the architecture of a software system and provide a higher level of abstraction as compared to classes and methods. In

a study of current architecture description languages (ADLs), Medvidovic and Taylor (2000) propose that an ADL describes applications in terms of at least the following: software *components* with their *interfaces*, *connectors*, and *configurations*.

Architectures of a family of systems contain many similarities to each other, and such similarities of organization can be represented as *architectural styles* (Shaw & Garlan, 1996). An architectural style defines a set of rules that describe or constrain the structure of architectures and the way in which their components interact. Architectural styles improve the efficiency of application development since individual applications can be designed using a similar set of rules and, potentially, underlying infrastructure.

JavaBeans and related Java technologies therefore lend support to software architectures in two main ways: support for architectural abstractions, and support for architectural styles.

Support for Architectural Abstractions in JavaBeans and Related Technologies

JavaBeans define a software component model for Java. Every technology described in this article defines a set of *components*: EJB provides session, entity, and message-driven beans; AWT provides GUI widgets; JDBC provides pooled connections and row sets; JavaMail defines Folder, Message, Store, and Transport; and JMX provides MBeans. Moreover, each technology defines *connectors* that can be used for interaction between components: JavaBeans, AWT, JDBC, JavaMail and JMX use properties, methods, and events for interaction; EJBs use RMI, method invocation, and asynchronous messages for communication and coordination; EJB containers provide a number of facilities in the EJB wrapper, which acts as a connector between the EJB and its container. Finally, an architectural configuration is achieved in one of two ways: instantiation and registration.

In instantiation-based configuration, the client of a component instantiates the component that provides services and maintains a reference to it, thus creating a configuration in which components hold references to every component they use. This technique is used in JDBC, AWT, and JavaMail. This technique allows for efficient interaction among components. However, the drawback of the technique is that components are highly coupled: any run-time component additions and removals need to be reflected in all affected components. Furthermore, a configuration created in this manner is only implicitly represented in the dependency information distributed across the components, hampering system understandability.

In registration-based configuration, a component is instantiated by its container to which the instance of the component is registered. The container then offers a lookup service with the help of which the required instance of a component can be located by its client. Once located, the component is referenced by the client, thus setting up the configuration of components in that architecture. This technique is used in EJB as well as JMX. JavaBeans also use registration of listeners as a variation of this technique to locate the clients of their events. The registration-based technique results in more flexible systems than the instantiation-based technique in that run-time changes to a system are handled more naturally via the lookup service. Of course, this flexibility is achieved at the expense of performance, which is impacted by the lookup service.

Support for Architectural Styles in JavaBeans and Related Technologies

JavaBeans support a publish-subscribe architectural style, in which interaction between components takes place through events, and components are producers or consumers of events. EJB defines a distributed component style, whereby two sets of interfaces are exposed, one for locating the components and another for providing business services. Both these styles are extensively used for developing consumer and business applications.

It has also been shown that JavaBeans can be adapted to support more elaborate styles such as the C2 architectural style (Taylor et al., 1996). C2 is a novel architectural style that is highly flexible, and lends itself to a variety of application domains as well as to dynamically changing architectures of systems. The C2 style consists of components and connectors that interact through messages. Rosenblum and Natarajan (2000) show how a seamless integration can be achieved between the creation of individual components and the architecture-based construction of a system to satisfy the system's requirements. They provide a C2-style aware visual builder tool for JavaBeans that can be used to develop the architecture of an application, and then design and implement the architecture of the application. Components created using the tool are JavaBeans, and their interaction takes place through JavaBean events which form the connectors in this style. The configurations of and interactions among the JavaBeans in Rosenblum and Natarajan's system adhere to the topological rules of the C2 style.

SUMMARY AND CONCLUSIONS

Component-based development of software systems has shown a lot of promise in alleviating the many problems experienced by traditional software development approaches. Similarly, the Java programming language has demonstrated a number of characteristics that make it particularly useful in developing large, complex software systems distributed across the Internet. The natural solution, then, is to provide a component-based solution for the Java setting. JavaBeans are just such a solution. In tandem with a suite of accompanying technologies including EJB, AWT, JDBC, Java Mail, and JMX, JavaBeans are capable of effectively addressing numerous challenges common in today's software development.

This article has provided an overview of JavaBeans and the suite of component-based development technologies anchored around Java. The article has also related these technologies to

an important recent direction in software engineering: software architectures. While the software development marketplace is very dynamic and new techniques and technologies are frequently introduced, this article has tried to focus on the principles underlying Javanese and its companions. These principles reflect some of the best software development practices and are likely to persist through future technological advances.

CROSS-REFERENCES

BIBLIOGRAPHY

Brockschmidt, K. (1994). Inside OLE 2. Redmond: Microsoft Press.

DeMichiel, L. (Ed.). (2002). Enterprise JavaBeans™Specification, v 2.1 Proposed Final Draft. Santa Clara: Sun Microsystems, Inc.

Ellis, J. & Ho, L. (Eds.).(2002). JDBC™ 3.0 Specification. Palo Alto: Sun Microsystems, Inc.

Englander, R. (1997). Developing JavaBeans. Sebastopol: O'Reilly & Associates.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). Design Patterns: Elements of Reusable Object-Oriented Design, Reading: Addison-Wesley.

Geary, D. M. (1999). Graphic Java 2: Mastering the JFC, Volume II, Palo Alto: Sun Microsystems, Inc.

Hamilton, K. (Ed.). (1997). JavaBeans™: Version 1.01-A. Mountain View: Sun Microsystems, Inc.

Microsoft Corp. (2001). Overview of the .NET Framework. Available at:
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpovrintroductiontonetframeworksdk.asp
(Date of access: October 28, 2002)

Medvidovic, N. and Taylor R. N. (2000). A classification and comparison framework for software architecture description languages. IEEE Transactions on Software Engineering. 26(1).

Mehta, N. R., Medvidovic, N. & Phadke, S. (2000). Towards a Taxonomy of Software Connectors. Proceedings of 21st International Conference on Software Engineering. Los Angeles: ACM Press.

Orfali, R., Harkey, D. & Edwards, J. (1996). The Essential Distributed Objects Survivable Guide. New York: John Wiley & Sons, Inc.

Perry, D. E., & Wolf, A. L. (1992). Foundations for the Study of Software Architectures. ACM SIGSOFT Software Engineering Notes. New York: ACM Press.

Rosenblum, D. & Natarajan, R. (2000). Supporting Architectural Concerns in Component Interoperability Standards. IEE Proceedings on Software. IEE: London.

Sessions, R. (1997). COM and DCOM: Microsoft's Vision for Distributed Objects. New York: John Wiley & Sons, Inc.

Shannon, B. (Ed.). (2002). Java2™ Enterprise Edition Specification, v1.4 Proposed Final Draft. Santa Clara: Sun Microsystems, Inc.

Shaw, M. & Garlan, D. (1996). Software Architecture: Perspectives on an Emerging Discipline. Upper Saddle River: Prentice-Hall.

Sun Microsystems, Inc. (2000). JavaMail™ API Design Specification Version 1.2. Palo Alto: Author.

Sun Microsystems, Inc. (2001). Java Remote Method Invocation Specification. Available at: http://java.sun.com/j2se/1.4/docs/guide/rmi/spec/rmiTOC.html.

(Date of access: January 19, 2003)

Sun Microsystems, Inc. (2002a). Java™ Foundation Classes: Cross-Platform GUIs & Graphics. Available at: http://java.sun.com/products/jfc/index.html.

(Date of access: November 13, 2002)

Sun Microsystems, Inc. (2002b). Java Management Extensions Instrumentation and Agent Specification, v1.1. Palo Alto: Author.

Taylor, R. N., Medvidovic, N., Anderson, K. M., Whitehead, E. J., Robbins, J. E., Nies, K. A., Oreizy, P., & Dubrow, D. L. (1996). A Component- and Message-Based Architectural Style for GUI Software. IEEE Transaction on Software Engineering, 22(6).