Understanding and Building Java Beans Components

Author: Ted Faison, Faison Computing Inc.

Type: Preconference Tutorial
Track: JBuilder

Short Description

An introduction to the Java Beans specification and Java Bean development. Examples are shown using Borland JBuilder as an example of an Application Builder tool to develop beans.

Abstract

Rapid application development (RAD) is the goal of all developers. The initial JDK 1.0.2 did not support RAD programming environments, lacking sophistication in the areas like application builder support and component-software packaging. Java Beans changes all that, poising Java to become the standard development language for the next decade. The Beans spec allows tool builders to create and reuse Java components in a RAD application-builder framework. Beans leverage many of the design patterns that have recently gained much attention, such as the Event model, property introspection, persistence and others. This paper gives an overview of Java Beans and shows how they can be used to accelerate software development.

Overview

Java Beans represent that latest attempt at building a better component model. Why do we need yet another model? We already have ActiveX for Microsoft Windows platforms, OpenDoc for the Macintosh, SOM/DSOM for OS/2 and others for just about every major platform. These existing models suffer from two main problems: they are platform-dependent and they are too complicated to use from the programmer's perspective. The fact that OLE is still considered an arcane system, even after several years from its release, is testament to the difficulties programmers encounter when approaching OLE in general and ActiveX in particular.

The Java Beans spec was written to address platform neutrality and programmer learning-curve-burnout from the start. The neutrality is achieved by using Java as the building language. To keep Java Beans simple, the designers made conscious design tradeoffs at every level, to provide as much flexibility and power as possible without creating a monster. One important decision that enormously simplifies a programmer's job was to incorporate support for object persistence directly at the language level. Java is the only major language to offer this feature. When you develop a Java Bean, it is persistent almost by default.

Java Beans are designed to be easy to use and easy to customize. Delphi and Visual Basic programmers are already familiar with Property Inspectors, which allow you to change the properties of an object. Beans offer the same type of support. Additional changes can be made to a Bean by adding customized event handlers. If changing properties or adding methods is still insufficient to achieve the desired goal, an entirely new class can be derived from an existing Bean and customized to your heart's content.

But no component is completely useful by itself. It is the interconnection of components that allows you to build real systems. The Java Beans spec was written to facilitate the interconnection of Beans, using a new technique known as the *Event/Listener model*. Beans can be interconnected with little effort using hand-

written code, but the preferred approach is to use a third party Application Builder, such as Borland JBuilder. Using a tool not only makes the interconnection process simple and fast, but also robust. This doesn't preclude bugs, but certainly reduces the occurrence of silly errors. Java Beans can also be connected to CORBA compliant ORBs, using so-called *bridges*, which are special components that translate Bean-jargon to CORBA-speak. Or, if you're really into CORBA, you can develop CORBA-compliant Beans using the Java IDL compiler. The Java motto tells it all: *Write Once, Run Everywhere!*

What Java Beans are not

From the above discussion it might be tempting to deduce that Beans are the successors to ActiveX, OpenDoc and all other current component models. Not so. Beans are designed to be used *with* and *in addition to* existing component models. To prove this point, the JavaSoft folks and a number of third party vendors offer technologies to allow Java Beans to connect with ActiveX, OpenDoc or other component types. There are also special wrappers you can use to package a Bean as an ActiveX, OpenDoc or other component.

Beans are also not designed for real-time applications. Being Java components, they run in the interpreted environment of the Java Virtual Machine, and are necessarily slower than their ActiveX or OpenDoc counterparts. It would generally be a bad choice to use Beans to implement the kernel of a real-time operating system, or to create device drivers.

Moreover, Beans are designed to be small, but not extremely small. It wouldn't be very efficient to create Beans for scalars, with incarnations like `BooleanBean` or `CharBean`. Beans are designed to encapsulate the level of functionality typically associated with class objects, which have several properties and possibly several event handlers that can be modified.

Anatomy of a Java Bean

Java Beans are essentially black-box entities that have built-in Application Builder support. A Bean is normally customized when used in an application. The way it can be customized involves 3 fundamental categories: Properties, Methods and Events (PME). Beans are often described as having a PME model. If a Bean can't be customized using PME changes, then only option is to use it as the ancestor for an entirely new class.

Properties

Properties have become popular with the advent of Visual Basic and Delphi, and even OLE automation. Properties are attributes you can change with scripting languages like Visual Basic or JavaScript, using assignment-statement semantics, such as:

```
myObject.Color = red;
```

or

```
color = myObject.Color;
```

Underneath the pretty syntax, accessing a property involves the invocation of *getter* or *setter* functions, collectively called *accessors*. Java properties aren't declared explicitly. They are defined by the signatures of their accessors. Given a property of type `SomeProperty`, the getter function will have the signature

```
SomeProperty getMyProperty( );
```

and the setter function will have the signature

```
void setMyProperty(SomeProperty theProperty);
```

The class `myBean` might have accessors that look like this for a property named `Color`:

```
public class MyBean {

  private Color color;

  public Color getForegroundColor() {return color;}
  public void setForegroundColor(Color theColor) {color = theColor;}

}
```

Because accessing properties involves method calls, side effects are possible. This is good, because assigning new values to a property can trigger other code that might be required to execute when the property is changed. For example the following code increments a counter every time the `Color` property of `MyBean` is changed:

```
public class MyBean {

  private Color color;
  private int colorChanges;

  public void setColor(Color theColor) {
    color = theColor;
    colorChanges++;
  }
}
```

Side effects can be much more extensive. For example, a grid control might have a property called `ColumnCount`. Assigning a new value to it would cause the grid to internally change the column count, reorganize its cell data in some way, then repaint the entire control with the new number of columns.

Properties are a natural and easy way to change the state of a Bean. Properties let you change attributes with simple assignments, and relieve you from having to know the number and order of parameters in a method call, worrying about values returned by a method call, or dealing with exceptions. Properties also lend themselves eminently to manipulation in an Application Builder. Property Sheets can be used, as in JBuilder, to allow developers to change basic attributes of a Bean without the need for recompilation or linkage of the component. Not only are properties simple to use, but setting them can cause changes to a Bean that are visible immediately at design time.

Indexed Properties

Some properties are not simple types, but occur as arrays. For example, a grid might have an array of `Column` objects, each of which stores attributes like the column `Title`, the `TitleFont`, the `TitleColor`, etc. To access a `Column` requires supplying the index of the desired column. Properties that are accessed through an array index are called *Indexed Properties*. The index must always be an integer. Assume the `Column` and `Grid` classes were defined like this:

```
public class Column {

  private String title;

  public String getTitle() {return title;}
  public void setTitle(String theTitle) {title = theTitle;}
}

public class Grid {

  private Column columns[];

  // accessors to individual Columns
  public Column getter(int theIndex) {return columns [theIndex];}
  public void setter(int theIndex, Column theColumn) {
    columns[theIndex] = theColumn;
  }
}
```

You could read and write the `Title` for column 3 with a Java-aware scripting language like this:

```
grid.Column[3] = “New Title”;
String title = grid.Column[3].Title();
```

Indexes are checked at runtime, and a `java.lang.ArrayOutOfBoundsException` will be thrown is necessary.

Bound Properties

Because objects are generally interconnected, it is sometimes necessary for property changes in one object to be signaled to other objects. Say one object A encapsulates the `User Preferences` of your application. Another object B might handle the painting of a window. If the user changes the selected background color for your application through object A, then object B needs to notified so it can update its background as well.

A *Bound Property* in Java is a property that signals interested objects when it is changed. The object that the property belongs to is called the property-change event source. The signal sent to interested objects is called an event. The objects receiving the notifications are called property-change event listeners. In order for an object to receiver property-change notifications, it must implement the `PropertyChangeListener` interface, like this:

```
class MyListener implements java.beans.PropertyChangeListener {

    // this method is called when a property is changed
    void propertyChange(PropertyChangeEvent evt);

    // ...
}
```

The listener must then register itself with the event source by calling the method

```
public void addPropertyChangeListener(PropertyChangeListener theListener);
```

To stop receiving property-change notifications, objects can call the following method of the event source:

```
public void removePropertyChangeListener(PropertyChangeListener theListener);
```

For more information about events, sources and listeners, see the section below entitled *The Event Listener Model*.

Keep in mind that you don't declare individual properties as `bound`, per se, using some new Java keyword. A bound property looks like any other property. When a property is changed, the setter method sends the property-change event if the property is bounded, otherwise it doesn't. To see which properties of a class are bound, you have to actually look at the source code for each property setter function.

Another important thing is that the property-change notification object carries information telling you which property was changed, its previous value and its new value. The event is fired by the property setter method, by calling the `propertyChange` method for each registered listener. Because there can be more than one listener for a property change, you need a container to keep track of all the listeners. When the time comes to fire the event, you iterate over the container and send the event to each listener in it.

Or you can do it the easy way. Because managing collections of listeners is such a common task, the JavaSoft folks implemented an ad hoc class called `PropertyChangeSupport` to handle all the details of managing collections of listeners. An object with bound properties instantiates a `PropertyChangeSupport` object, and delegates all property-change notifications to it, like this:

```
public class MyBean {

  private Color color;
  private java.beans.PropertyChangeSupport propertyChangeListeners =
                                new java.beans.PropertyChangeSupport(this);

  public Color getColor() {return color;}
  public void setColor(Color theColor) {
    propertyChangeListeners.firePropertyChange("Color", color, theColor);
```

```
    color = theColor;
  }

  public void
  addPropertyChangeListener(java.beans.PropertyChangeListener theListener) {
    propertyChangeListeners.addPropertyChangeListener(theListener);
  }

  public void
  removePropertyChangeListener(java.beans.PropertyChangeListener theListener) {
    propertyChangeListeners.removePropertyChangeListener(theListener);
  }
}
```

The events sent to each listener are objects of class `PropertyChangeEvent`, which looks something like this:

```
public class PropertyChangeEvent extends java.util.EventObject {

    private String propertyName;
    private Object newValue;
    private Object oldValue;
    private Object propagationId;

    public PropertyChangeEvent(Object source,
                               String propertyName,
                              Object oldValue,
                               Object newValue) {…}

    public String getPropertyName() {
       return propertyName;
    }

    public Object getNewValue() {
       return newValue;
    }

    public Object getOldValue() {
       return oldValue;
    }

    public Object getPropagationId() {
       return propagationId;
    }

}
```

The `propertyName` is a `String` that indicates the printable name of the property. The name doesn't have to match the property's name exactly. You could use an abbreviation or even a different word, as long as the listeners understand it.

The `propagationId` is currently marked for future use by JDK1.1. Its purpose is to prevent circularities in property change notifications. A circularity occurs when two objects are direct or indirect property-change listeners of each other. Changing a property in A causes a notification to be sent to B, which in response changes one of its own properties. This causes B to send a notification to A. The ensuing infinite notification loop would immediately deadlock the system. The `propagationId` field allows you to tag a notification with a number that the listener can look at. If the listener sends a property notification as a consequence of receiving a property-change notification, it must use the first notification's `propagationId` in the new notification. This way the original property-change source can determine that the notification is in response to its first notification and avoid an infinite loop. Keep in mind that this whole scenario of use of the `propagationId` field is yet to be implemented.

The following example shows a class that listens for changes in the property named `Color`. The listener doesn't check the ID of the event source, because it only installed itself as a listener on one source.

```
class MyListener implements java.beans.PropertyChangeListener {

    java.awt.Color backgroundColor = null;

    public void
    propertyChange(java.beans.PropertyChangeEvent theEvent) {

      // see if the background color changed
      if (theEvent.getPropertyName() == "Color") {

        // get the new color
        Color oldColor = (Color) theEvent.getOldValue();
        Color newColor = (Color) theEvent.getNewValue();
        if (oldColor == newColor) return;

        backgroundColor = newColor;

        // the background changed, so repaint ourselves
        // ...
      }
    }
```

Bound and Constrained Properties

When a property is bound, any changes to it will trigger a notification event to registered listeners. There are occasions when changes to a property need be validated by other objects before being accepted. Such bound properties are said to be *constrained*, because one of the property-change listeners can veto the change. Say you have a tab control object, showing the page numbers available in a document. By clicking the page number you set a `pageNumber` property to a new page. Assume the document object with the actual pages is a multi-user object. Users can add and delete pages at any time. Before you switch the tab control to a new page, you must have the document validate that the page exists. If the page was deleted, the listener throws a `PropertyVetoException`, and the tab control avoids going to a non-existent page, and possibly even removes the tab for that page.

Here's how it all works. First you must have an object that contains the constrained property. When I say *constrained*, it is implied that the property is also bound. There is no such thing as a constrained unbound property. The property will have the usual getter and setter methods, whose names follow the Java property design patterns described for bound properties. Calling the setter function causes a `PropertyChangeEvent` to be broadcast to all registered listeners. The order of delivery is not specified by the Beans spec, so be careful your listeners are not relying on a certain order to work correctly. Because the setter method can cause a `PropertyVetoException`, you can identify constrained properties by the signature of their setter method. For a property name `PageNumber`, the setter would look like this:

```
public int getPageNumber();
public void setPageNumber(int thePageNumber) throws PropertyVetoException;
```

When a listener receives a `PropertyChangeEvent`, keep in mind that the reason the event was sent was not to tell the listener that the property changed, but only to find out if all the listeners allow the property to be changed. If listener A received the notification and changed it internal state to reflect the event, it is possible for a subsequent object in the listener chain to veto the change. Listener A would now have an incorrect state.

Listeners for constrained properties must implement the `VetoableChangeListener` like this:

```
class MyVetoableListener implements java.beans.VetoableChangeListener {

    // this method is called to validate a property change
    public void vetoableChange(java.beans.PropertyChangeEvent theEvent)
       throws java.beans.PropertyVetoException {}

    // ...
}
```

When the constrained property is changed, it is up to the setter function to fire the `PropertyChangeEvent` to the listeners, by calling their `vetoableChange` method. As with bound properties, the best way to handle listeners is to use a `PropertyChangeSupport` object, like this:

```
public class MyBean {

  private Color color;
  private java.beans.VetoableChangeSupport vetoableChangeListeners =
           new java.beans.VetoableChangeSupport(this);

  public Color getColor() {return color;}
  public void setColor(Color theColor)
         throws java.beans.PropertyVetoException {
    vetoableChangeListeners.fireVetoableChange("Color", color, theColor);

    // if we get here, it means no listener vetoed the change

    // change the property
    color = theColor;
  }

  public void
  addVetoableChangeListener(java.beans.VetoableChangeListener theListener) {
    vetoableChangeListeners.addVetoableChangeListener(theListener);
  }

  public void
  removeVetoableChangeListener(java.beans.VetoableChangeListener theListener) {
    vetoableChangeListeners.removeVetoableChangeListener(theListener);
  }
}
```

Methods

Methods contain a Bean's code, and are equivalent to C++ member functions. The previous section showed `public` methods for accessing property values, but methods can have other access control modifiers. The Java language defines the following modifiers: `public`, `protected`, `private` and `package`. The first three are equivalent to their C++ cousins. The last is similar to a C++ `friend`. Declaring a method of type `package` makes it accessible to all the other classes defined in the same package. A method declared without an access control modifier is by default a `package` method.

Java methods are allowed to be overloaded, as in C++, but don't support optional arguments. Java Bean methods are no different from ordinary Java methods, except that they relate to Java Bean objects. Except for methods related to accessing properties, Bean methods are generally not accessible at design time in Application Builders like JBuilder.

Events

When something happens to a Bean, it is called an *event*. Events can occur when a Bean gains or loses the focus, when the mouse is clicked on it, when a key is typed, etc. Which events are significant to a Bean depend entirely on what the Bean is designed to do. Some Beans will support GUI-type events, like mouse and keyboard actions. Other Beans may not be visible at all, and respond to non-GUI-related events. For example, a Database Bean might respond to a `PostRecordUpdate` event when the user changes a record.

The Event-Listener Model

In general, events propagate state-change notifications from one place to another. Some events are generated by the system, others originate from Java objects or Beans. System events describe actions that originate outside your program, such as mouse and keyboard activities. Events can also be generated by Java Beans in your own program. The originator is called the *source* object, and the receiver is called the *listener* object. It takes both objects for an event to be transmitted. In order for a listener to receive events from a source, it must register itself as a listener with that source. Listeners must be derived from an `EventListener` interface. Java comes with a number of built-in listeners for common things like mouse and keyboard events. The following figure shows the class hierarchy of these built-in interfaces:

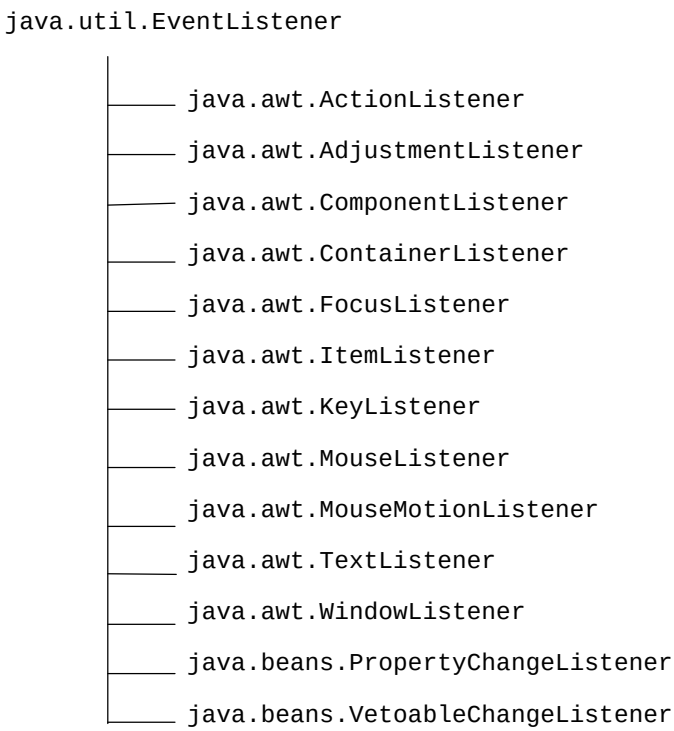

Figure 1 - The built-in Java event handler interfaces.

Objects that are sources of events are not required to be derived from any special class or interface. When an event occurs, the source invokes a method on the listener. The event itself is encoded in an object, which must inherit from `java.util.EventObject`. This arrangement entails that the source know something about the listener, because it invokes one of the listener's methods. A separate event class must be created for each type of event you will deal with. As of this writing, Java comes a variety of built-in event classes, as shown in the following class hierarchy.

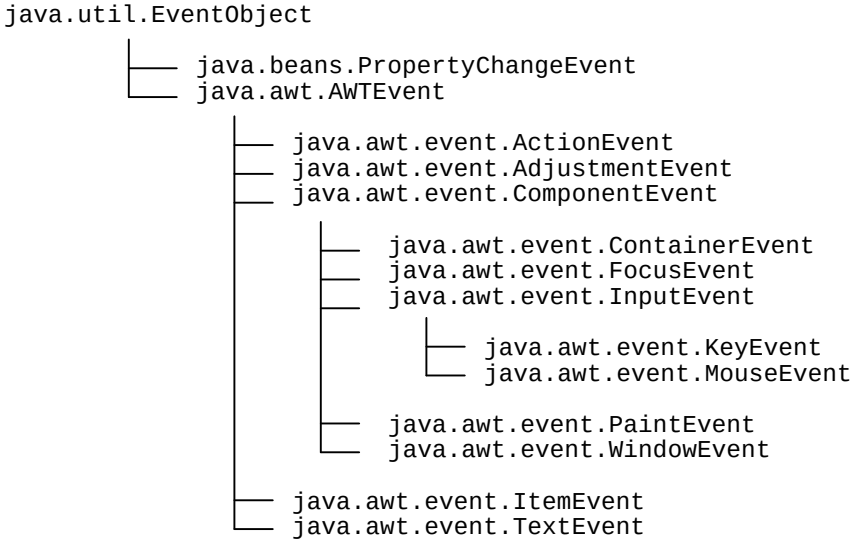

Figure 2 - The built-in Java event classes.

The BDK also contains some sample code showing the use of user-created events in the `beans\demo\sun\demo\quote` directory.

Hooking up the source and listener

OK. You have 3 objects so far: an event source, an event listener and an event object. Now what? To connect the source and listener together, you need to register the listener with the source. You do so using by calling an `addEvent` method. The method name should follow the design pattern for events, i.e. should be the word 'add' followed by the event name. For an event class called `MyEvent`, the registering method should be named `addMyEvent`. To unregister a listener, the `removeEvent` method should be called. Following the standard design pattern, to unregister a `MyEvent` listener, you call the method `removeMyEvent`. As mentioned earlier, to send an event, the source must know something about the listener. If an object is capable of sending `MyEvent` objects, it not only knows about the `MyEvent` class, but also about the `MyEvent` listener class, because the source invokes a specific method on the listener.

Say you created an LED Bean that needs to show the state of a button. When you press the button, the LED is supposed to turn on. When you release the button, the LED turns off. You can get this functionality using two events sent from the button to the LED. The first step is to create event classes that carry information about what happened. So you create the class `ButtonPressedEvent`. By convention, event class names end with "Event". You wind up with a hierarchy like this:

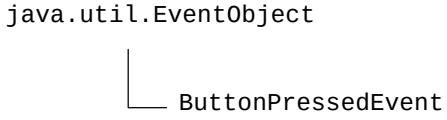

Figure 3 - A simple event class.

You might implement the class something like this:

```
public class ButtonPressedEvent extends EventObject {
  public ButtonPressedEvent(java.awt.Component theSender) {
    super(theSender);
  }
}
```

This simple class doesn't really do anything, expect create a new type that you can use to identify `ButtonPressed` events to the listener. To make the LED turn off, you have two options: you can create a new `ButtonReleasedEvent`, or you can change your original event class to carry information about the state of the button. Let's take the latter approach. The class will need a member to hold the button state. Because an event transmits information about something that has already occurred, it is important to protect its fields from alteration. Making them `private` or `protected` usually suffices. I'll call the new class `ButtonClickedEvent`, and implement like this:

```
public class ButtonClickedEvent extends EventObject {

  protected boolean pressed;
public boolean isPressed() {return pressed;}

  public ButtonClickedEvent(java.awt.Component theSender,
                            boolean stateIsPressed) {
     super(theSender);
```

```
    pressed = stateIsPressed;
  }
}
```

Using this new class, if we receive a `ButtonClickedEvent` object, we can call its `isPressed` method to see if the button was pressed or released. Now that we have an event to send, we need an LED Bean that knows how to handle it. The class must understand `ButtonClickedEvents`, so first I'll create a `ButtonClickedListener` interface like this:

```
interface ButtonClickedListener extends java.util.EventListener {
  void buttonClicked(ButtonClickedEvent theEvent);
}
```

Any listener of `ButtonClickedEvents` will be required to implement the `ButtonClickedListener` interface. For this quick example, the listener is the LED bean. I'll implement the LED like this:

```
class LED extends java.awt.Component implements ButtonClickedListener {

  public void buttonClicked(ButtonClickedEvent theEvent) {
    if (theEvent.isPressed() )
      // paint the LED on
    else
      // paint the LED off
  }
}
```

We're still missing a button that knows how to fire `ButtonClickEvents` to registered `LED` object. I'll create it as a class called `MyButton` and implement it like this:

```
class MyButton extends java.awt.Component {

  public boolean mouseDown(java.awt.Event evt, int x, int y) {
       sendButtonClickEvent(new ButtonClickedEvent(this, true) );
      return true;
  }

  public boolean mouseUp(java.awt.Event evt, int x, int y) {
       sendButtonClickEvent(new ButtonClickedEvent(this, false) );
      return true;
  }

  public void sendButtonClickEvent(ButtonClickedEvent theEvent) {
       // notify all listeners
       Vector initialListeners;
       synchronized(this) {
         initialListeners = (Vector) buttonListeners.clone();
       }
       for (int i = 0; i < initialListeners.size(); i++) {
         ButtonClickedListener nextGuy =
            (ButtonClickedListener)(initialListeners.elementAt(i));
         nextGuy.buttonClicked(theEvent);
       }
  }

  protected Vector buttonListeners = new Vector();

  public synchronized void
  addButtonClickedListener(ButtonClickedListener theListener) {
    buttonListeners.addElement(theListener);
  }

  public synchronized void
  removeButtonClickedListener(ButtonClickedListener theListener) {
    buttonListeners.removeElement(theListener);
```

```
  }
}
```

The two `add` and `remove` methods near the end of the class manage the registering and unregistering of listeners. An internal field of type `Vector` is used to manage the collection of listeners. Class `MyButton` is implemented as a multi-cast object, because it supports multiple listeners. The class would be simpler if it only allowed one listener, because a simple field of type `ButtonClickListener` would suffice. If `ButtonClickedEvent` were a uni-cast event, attempts to register more than one listener would cause you to throw a `TooManyListeneresException` in `addButtonClickedListener` if attempts are made to register more than one listener.

Getting back to my example, the interesting stuff is in the method `sendButtonClickEvent`, which takes a `ButtonClickedEvent` as a parameter. The event carries inside itself the new button state. `sendButtonClickEvent` iterates over the registered listeners, calling their `buttonClicked` method. A temporary internal vector is used during the iteration, because the `initialListeners` vector might be changed during the iteration by listeners calling directly or indirectly `addButtonClickListener` or `removeButtonClickListener`.

We're on the homestretch. We have a method to register `ButtonClickedEvents`, so we need someone to call it. The *someone* can be anyone. If you're doing things the hard way (i.e. by hand) you might have the listener register itself. If you're doing things the easy way (i.e. using an Application Builder like JBuilder), the listener might be registered by the parent frame window of the button and LED. For the purpose of this discussion, it doesn't matter who registers the `ButtonClickedListener`. The registration code needs to look something like this:

```
MyButton myButton = new MyButton();
LED led = new LED();
myButton.addButtonClickedListener(led);
```

Event Adaptors

While connecting objects together as in the previous example is not too complicated, the technique has a problem: `MyButton` and `LED` are not completely generic. `LED` works only with `MyButton` and vice-versa. It would be much preferable to have a way to code the two classes so they didn't have that inter-dependency. The Java solution is through event adaptors, which are objects that connect an event source with an event listener. Adaptors make it possible to remove class dependencies from sources and listeners, but at a cost. The dependency isn't really eliminated, just moved into the adaptor. At least the class dependency is in only one place.

When you connect objects together through events using an Application Builder like JBuilder, an adaptor class is generated automatically, and code is created to hook the adaptor to the source and listener. Using the `MyButton` example from above, JBuilder would create an `ActionEvent` to represent the button click, and an `ActionAdaptor` to hook the button up with the parent frame. Remember from Figure 2 that `ActionEvent` is a built-in event. Assuming the parent frame window was called `MyFrame`, the `ActionAdaptor` code would look like this:

```
class MyFrame_myButton_ActionAdapter implements ActionListener{
  MyFrame adaptee;
 MyFrame_myButton_ActionAdapter(MyFrame adaptee) {
    this.adaptee = adaptee;
  }
  public void actionPerformed(java.awt.event.ActionEvent e) {
    adaptee.myButton_actionPerformed(e);
  }
}
```

The controls and the adaptor are created in the parent frame window like this:

```
class MyFrame extends DecoratedFrame {
  MyButton myButton = new MyButton();
  LED led = new LED();

  public MyFrame() {
    // …
    myButton.addActionListener(new MyFrame _myButton_ActionAdapter(this));
  }
}
```

When you click the button, the adaptor calls the frame's `myButton_actionPerformed` method, passing it an `ActionEvent` object. You write code in this handler to set the LED on or off:

```
class MyFrame extends DecoratedFrame {
  // …
  void myButton_actionPerformed(java.awt.event.ActionEvent e) {
    // … turn the led on
  }
}
```

Adaptors can also serve other purposes. Rather than merely passing along an event from a source to a listener, adaptors can multiplex or demultiplex events, serve as intelligent queues, manage event delivery based on priority, and more.

Event Handlers in JBuilder

Event handlers are a powerful and simple way to customize a Bean. Application Builders like JBuilder display the design-time event handlers to the developer as a tabbed page in the Property Inspector. To add an event handler for an event, all you do is type the name of the handler in the Property Inspector. Better yet, just double click the edit field next to the property. JBuilder automatically creates a handler by concatenating the object name with the event name. For an object name `button1`, the `actionPerformed` handler would look like this in the Inspector:

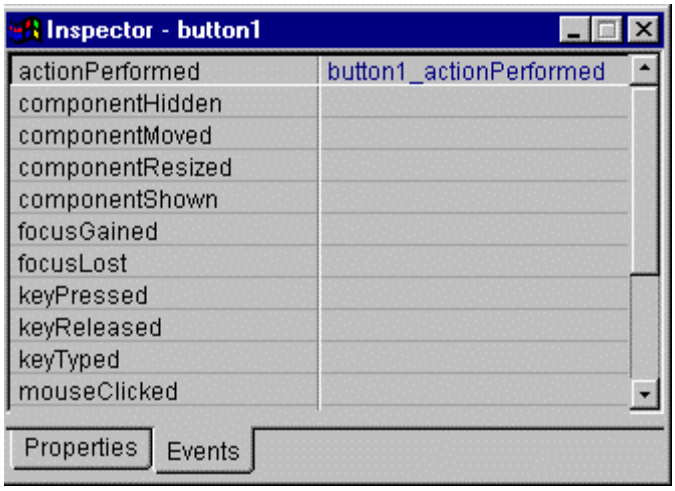

Figure 4 - The Events page for java.awt.Button in JBuilder.

The handler is created as a method of the parent object. Say you have a simple `java.awt.Frame` window called `MyFrame` with a `java.awt.Button` called `button1`, as in Figure 5.

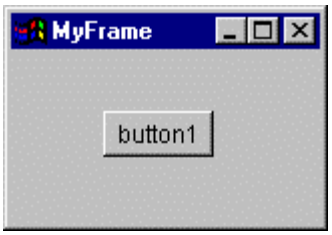

Figure 5 - A simple frame window.

Double clicking on the `actionPerformed` field in the Property Inspector causes JBuilder to create the handler `button1_actionPerformed` as a method of `MyFrame`, like this:

```
class MyFrame extends DecoratedFrame {
  //…
  void button1_actionPerformed(java.awt.event.ActionEvent e) {
    // add your code here...
  }
}
```

The handler is passed an `ActionEvent` parameter, describing the button event that occurred. The class `java.awt.event.ActionEvent` is used heavily in JBuilder applications, and is used anywhere a handler can infer what happened without detailed event help. The class has two methods, `getActionCode` and `getModifiers`. An `ActionCode` is just a string set by the event source. In the example above, `button1` sets the string to just… "button1". `ActionEvent` modifiers encode the state of the modal keyboard keys, such as the SHIFT, CONTROL and ALT keys. The button handler in my example doesn't really need to know what event occurred, because it is obvious: I created the handler to process button clicks, and the handler doesn't care what modal keys are pressed. The only way the handler can get called is through a click on `button1`, so there is no doubt about what event occurred. `ActionEvent` is just a plain vanilla event. It basically tells the listener: your event occurred. The listener should know just what event this is. You could use `ActionEvent's ActionCode` to carry additional information about the event, but JBuilder creates most of the `ActionEvents` you'll be using, so all you'll normally get out of `ActionCode` is the printable name of the event source.

The event handler in `MyFrame` isn't called directly from `button1`: JBuilder always uses adaptors to call handlers. When you create an event handler, JBuilder creates an adaptor through which a method in the parent object is invoked by the event source. This follows the event-delegation model described earlier. Except for the adaptor part (not exactly a trivial addition), the JBuilder event model is similar to the Delphi model.

Persistence

If you change the properties of a Bean using an Application Builder, you expect those changes to be saved permanently *somehow*. The *somehow* part requires that Beans be persistent. In C++, persistence was supported by a class library, which invariably was vendor-dependent and always quite complicated. In Java, persistence was deemed to be such a valuable feature that the Java folks decided to support it at the language level, a decision that not only makes persistence vendor-independent, but practically inconspicuous in many programs. Beans get their persistence essentially for free from the Java language. Persistence is available to any object created using JDK 1.1 – not just Beans. Even though persistence is not a Beans topic per se, it is so important that I'll discuss it briefly.

Meta Classes

Persistence is based on the ability to serialize things, i.e. to write objects out to a stream one byte at a time, and be able to reconstruct objects by reading bytes from a stream. The concept is simple to understand, and not too hard to implement. Writing Java objects to storage without requiring user code is accomplished with a little help from objects that understand the layout of Java objects. Associated to each Java class and interface is a special object of type `java.lang.Class`, which is essentially a meta-class, i.e. it contains a description of the class. C++ programs require macros to represent class meta-data. Visual C++ programmers are familiar with the macros `DECLARE_DYNAMIC` and `IMPLEMENT_DYNAMIC` to support serialization. Each C++ vendor has a proprietary set of macros to support serialization. Java has had meta-classes from day one, and the JDK 1.1 has added new features to support persistence. The implementation is so unobtrusive that novice Java programmers don't even know it's there!

Any time you create a class, the Java system automatically creates a `Class` object to go with it. The `Class` object is able to create a class from a class name (e.g. `"java.MyBeans.MyFirstBean"`), and can return a list of each field and method used in the class. When you save a class to a stream, the Java system writes information obtained from its `Class` object followed by the object itself. The serialized `Class` information contains enough information for Java to later read the object back in and reconstruct it properly. Java serializes `Class` objects differently from other classes, to avoid infinite recursion.

Serializable classes

Java objects are not persistent by default. If you want persistence, you must derive a class from either the `java.io.Serializable` interface or the `java.io.Externalizable` interface. The former provides transparent serialization support, the latter gives you, the class author, complete control over how your class is serialized. A simple persistent bean can be declared like this:

```
public class MyClass implements java.io.Serializable {
  // …
}
```

By default, all fields of the class will be persistent. You can tell Java not to save a field by marking it with the `transient` keyword. Any fields that are always initialized by the class itself should be marked `transient`, such as:

```
public class MyPanel extends Panel implements Serializable {

  private transient boolean active;
  public MyPanel() {active = false;}
}
```

Once you have a serializable Bean, you can stream it to a file very easily, like this:

```
MyPanel myPanel = new MyPanel();
FileOutputStream myFile = new FileOutputStream("MyPanelFile.ser");
OutputObject output = new OutputObject(myFile);
output.writeObject(myPanel);
output.flush();
```

You can use whatever filename and extension you want, of course, but the Java convention uses the extension `.ser` for serialized beans. If you handle the file using your own code, the extension doesn't matter, but if you plan to package the component in a Java Archive (JAR), then the `.ser` extension is important. The

`OutputObject` object controls the serialization process to a stream. You can just as easily stream your objects to a serial port or to a modem, by changing the object passed to the `ObjectOutput` constructor.

The simple statement

```
output.writeObject(myPanel);
```

packs a lot of power. If `myPanel` contains references to other objects, those objects are also saved. If `myPanel` is part of a network of objects, the entire network is saved. Java guarantees that it will save enough information to restore every non-`transient` field in a class. If a field references another object, the other object is tagged with an internal handle, then Java saves both the object and the handle. The process repeats recursively until all reachable objects have been saved.

Implementing the `Serializable` interface is what is required for a Java class to be persistent. But persistence is not the only consequence of being `Serializable`. What the interface really indicates is that a class is capable of being broken down into a series of bytes, and can be reconstructed from a series of bytes. Just where these bytes go or come from is not important to the class. This is good, because one very important task that has nothing to do with persistence, but everything to do with serialization is passing Java objects over the wire in a Remote Method Invocation (RMI), which is really just a remote procedure call. Using a language like C++, there is no end to the complexity and details of parameter marshaling, IDL interfaces, compilers and what have you. In Java, any object that is serializable interface is auto-magically sendable over the wire. No funny stuff happening under the table. Since Java knows how to take the class apart into bytes and put it together again using standard serialization, persistent objects are also *remotable*.

If you have special requirements for the way individual fields are serialized, you can tag them `transient` and provide `writeObject` and `readObject` methods to write and read the object yourself. Say you have a `UserAccount` class that has a password field. You wouldn't want the password to be written in clear text to a stream. When the times comes to serialize `UserAccount`, you want to encrypt the password field. You'll need to decrypt it on deserialization. The class might initially look something like this:

```
class UserAccount implements java.io.Serializable {
  transient private String password;
  private void writeObject(ObjectOutputStream theStream)
               throws java.io.IOException {}
  private void readObject(java.io.ObjectInputStream theStream)
               throws java.lang.ClassNotFoundException,
                      java.io.IOException {}
}
```

`UserAccount` might read and write the encrypted password something like this:

```
class UserAccount implements Serializable {
  transient private String password;

  // stubs for encryption/decryption methods
  String Decrypt(String theString) {return theString;}
  String Encrypt(String theString) {return theString;}

    private void writeObject(ObjectOutputStream theStream)
                 throws java.io.IOException {
     // write the non-transient fields
     theStream.defaultWriteObject();

    // write the encrypted password
    theStream.writeObject(Encrypt(password) );
    }

    private void readObject(java.io.ObjectInputStream theStream)
              throws java.lang.ClassNotFoundException,
```

```
                    java.io.IOException {
    // Read in the inherited stuff
    theStream.defaultReadObject();

    // read the encrypted password
    String encryptedPassword = null;
    encryptedPassword = (String) theStream.readObject();
    password = Decrypt(encryptedPassword);
  }
}
```

I left out the details of the encryption/decryption logic. The important thing is that `Serializable` objects don't have to do anything (other than implement the `Serializable` interface) to become persistent. If the default persistence support doesn't fit certain fields, you can provide your own read and write functions. You can call `defaultReadObject` and `defaultWriteObject` to handle all the non-transient fields of your class, and never have to worry about reading and writing the ancestors, because Java takes care of that automatically.

Externalizable classes

When you need to take complete control over what and how Java serializes a class, you implement the `Externalizable` interface. Why would you need to take full control over serialization? Say you wanted to write your class capable of reading/writing a compressed version of itself. Maybe the class contains a lot of data, and you want to send it over a modem connection. Using the standard `Serializable` interface doesn't quite give you enough control, so you make the class `Externalizable`.

By having your class implement the `Externalizable` interface, you tell Java that you intend to take charge of serialization. You'll need to add special read and write methods to your class, and the Java I/O classes will call these methods when necessary. To start with, your class might look like this:

```
class MyCompressableBean extends java.awt.Component
                         implements java.io.Externalizable {
  public void writeExternal(ObjectOutput theOutput) {}
  public void readExternal(ObjectInput theInput) {}
}
```

When the Java I/O system needs to serialize your bean, it will recognize that is isn't a regular `Serializable` bean. Discovering that the bean implements `Externalizable`, it then invokes your bean's `writeExternal` or `readExternal` methods. Implementing these methods will generally require more work than the equivalent `writeObject/readObject` methods in the `Serializable` interface, because the Java system doesn't automatically read or write any ancestors of your class. It is up to you to decide whether you want the base classes to be streamed, and how.

Keep in mind that the built-in AWT and JBuilder classes don't implement `Externalizable`, so `MyCompressableBean` can't read or write the ancestor using code like this:

```
class MyCompressableBean extends java.awt.Component
                         implements java.io.Externalizable {

  public void writeExternal(ObjectOutput theOutput) {

    super.writeExternal(theOutput);    // this won't work!

    // write fields of MyCompressableBean
    //...
  }
```

```
  public void readExternal(ObjectInput theInput) {

    super.readExternal(theInput);      // this won't work!

    // read fields of MyCompressableBean
    //...
  }
}
```

You can't call the ancestor's `readObject`/`writeObject` methods, because they take `ObjectInputStream` and `ObjectOutputStream` as parameters, respectively, and you don't have those kinds of streams in `readExternal/writeExternal`. So how do your stream your class? The answer is usually "*the hard way*". Some classes may not need to stream fields for any of their ancestors. In this case the task is simpler, making `MyCompressableBean` look perhaps something like this:

```
class MyCompressableBean extends java.awt.Component
                         implements java.io.Externalizable {

  // assume all the class' data is in this array
  byte[] data = new byte[100];

  // assume these 2 methods handle the compression/decompression
  byte[] compressedArray(byte[] theArray) {
    // pretend we return a compressed array...
    return theArray;
  }

  byte[] decompressedArray(byte[] theArray) {
    // pretend we return a decompressed array...
    return theArray;
  }

  public void writeExternal(ObjectOutput theOutput)
              throws java.io.IOException {
    // write fields of MyCompressableBean
    theOutput.write(compressedArray(data) );
  }

  public void readExternal(ObjectInput theInput)
              throws java.io.IOException {
    // read fields of MyCompressableBean
    byte[] compressedBytes = new byte[100];
    theInput.readFully(compressedBytes);
    data = decompressedArray(compressedBytes);
  }
}
```

In this example, I assume that all the fields for `MyCompressableBean` are stored in the byte array `data`. Reading and writing of this array is performed using the `write` method of `OutputObject` and `readFully` of `InputObject`.

If your class needs to read and write fields from its ancestors, then you have to write your own code to determine the fields of each ancestor, and read and write them yourself. You would use the meta class information supplied by `java.lang.Class.getField` to access fields in each of the ancestors. See the section *Class Meta-Data* below for information on how to do this.

Versioning

Objects invariably evolve with time. You start out with one version, you debug it and ship it with your application. The application is persistent, and saves its state when the user terminates it. When the application

is restarted, it uses the serialized data to resume where the user last left it. Your customer loves it. Until you make some enhancements. You ship the enhancements and boom! When the user tries to start the app, the system crashes. What happened? Simple, you have a versioning problem. When your new app tried to restore itself using the old customer's data, it used data that didn't correlate with the new enhanced classes.

Versioning is the process of stamping serialized data with information that indicates the version of the saved class. Versioning in Java is optional, but always a good idea to support. To version a class, you add a `final` field named `serialVersionUID` to it, like this:

```
class MyClass implements java.io.Serializable {
    // …
    private static final long serialVersionUID = -3665804199014368530L;
}
```

The actual value of `serialVersionUID` is not important in itself. It is a hash code that represents the class.

In C++ programs, you handled object versioning manually. You tagged the stream with an arbitrary value. Each time you made changes to classes that resulted in changes to the serialized data structure, you incremented the version number. I don't have to tell you how easy it is to forget to bump the number after making a change in the heat of a project deadline. Java takes over the process, and computes the version number not using an arbitrary number sequence, but by using a 64 bit hash code computed from the class meta data itself.

When a class is serialized, the Java I/O layer writes the version to the stream. When the class is read back, the `serialVersionUID` from the stream is checked with the value of the class being reconstructed. If the values are different, the system recognizes a versioning problem and throws an `InvalidClassException`.

Restoring a Bean

Once a Java object has been saved in a stream, it can be read back. Reading objects is a bit more complicated than writing, mainly because it involves identifying what type of object is stored in the stream and creating an equivalent object in memory to initialize from the stream. The details are fortunately hidden from application programs. To read a Java object, the following code will suffice:

```
FileInputStream myFile = new FileInputStream("MyPanelFile.saved");
InputObject input = new InputObject(myFile);
MyPanel myPanel = (MyPanel) input.readObject();
```

This example requires an explicit typecast, meaning you must already know the type of the object before you read it in. Any objects that are streamed in with `myPanel` will be properly restored an initialized.

Application Builder Support

One of the most interesting features of Java Beans is the degree to which they support Application Builders. The idea of RAD development is centered on the use of tools to make your work easier and faster. It then helps if Beans have a way of helping the tool do its job. What kinds of information does an Application Builder need? Basically, it needs a way to obtain from each Bean at design-time a description of the Properties, Methods and Events that are available. Older component systems, like OLE, are not able to give this kind of information, because COM interfaces possess no meta-data. To get any kind of information about an OLE object, you must have access to its `.tlb` Type Library file. The creation of Type Libraries is awkward, requiring the use of a stand-alone Type Library compiler. To read the file, you need a tool that understands the binary structure of `.tlb` files. If you have the executable file (DLL) of an OLE component, but not its `.tlb`

file, you're out of luck, because you can't generate the `.tlb` using a DLL alone. This is not a small problem, especially since most of the OLE components out there are distributed without Type Libraries. The decision to include Application Builder support in each and every Bean means the OLE situation will never occur with Beans.

Introspection

Application Builder support was designed to make it simple to, implement for most Beans, requiring essentially no work on the Bean developer's part. The folks at JavaSoft didn't want Application Builder support to create a big overhead requirement for every Bean. Keep in mind that one of the basic principles behind the Bean philosophy is simplicity (which usually means small footprints). On the other hand, there are certain Beans that have really complicated properties, requiring elaborate Property Sheets. These Beans must have a way to show their own custom Property Sheets, bypassing the default property display mechanism. To solve the problem, Beans use two techniques to support Application Builders: a low-level one, based on a process called *Reflection*, and a high level one, based on explicit specification. The process of an Application Builder obtaining meta-data from a Bean at design-time is called *Introspection*. Introspection is carried out using either Reflection or explicit specification, through `BeanInfo` objects.

Reflection

When an Application Builder is handed an arbitrary Bean, it must be able to somehow inspect the Bean and determine its Properties, Methods and Events. First the Builder checks to see if a `BeanInfo` object is available for the Bean. If not, the tool then analyzes the names of the Bean's methods and checks to see if any names follow certain naming conventions or design patterns. This is the Reflection mechanism.

Design Patterns for Properties

Properties come in different types and each type has its own design pattern. Simple properties must have accessor functions whose signatures match the design pattern:

```
public <PropertyType> get<PropertyName>();
public void set< PropertyName >(<PropertyType> theProperty);
```

For example, a property named `MyColor` would be identified by the accessor functions:

```
public Color getMyColor();
public void setMyColor(Color theColor);
```

Note that the `<PropertyType>` and `<PropertyName>` don't have to be the same. The name of the property is just a string displayed in the property sheet. The type is the Java class that represents the property internally. If only a getter accessor is found, then a property is assumed to be read-only, and similarly for the setter accessor.

Boolean properties occur so often that they were given the special design pattern.

```
public boolean is<PropertyName>();
public void set<PropertyName>(boolean theValue);
```

A boolean property called `Active` would have the accessors:

```
public boolean isActive();
```

```
public void setActive(boolean theValue);
```

If `MyBean` had the `Active` property, you would access it like this:

```
MyBean myBean = new MyBean();
if myBean.isActive()
  // do something…
myBean.setActive(true);
```

Indexed properties are another special case. To access an indexed property you need to specify an index, so the following design pattern is used:

```
public <PropertyType> get<PropertyName>(int theIndex);
public void set<PropertyName>(int theIndex, <PropertyType> theProperty);
```

To access the `Column` information for a grid Bean, the accessors might be:

```
public GridColumn getColumn(int theColumn);
public void setColumn(int theColumn, GridColumn theColumnObject);
```

Design Patterns for Methods

There are no design patterns for methods. Any methods that don't follow a known design pattern are considered just…methods, meaning they are construed to not be Properties or Events. Methods are not usually displayed in Property Sheets at design-time by Application Builders, because they are available to be called only at run-time. An Application Builder might display a Bean's methods in the context of a debug session, perhaps in a `Watch` or `Evaluate` window.

Design Patterns for Events

Beans can generate two kinds of events: unicast and multicast. The unicast variety requires only one listener. If an attempt is made to register other listeners, the Bean must throw a `java.util.TooManyListenersException`. Events use the following design pattern:

```
public void add<EventListenerType>(<EventListenerType> theListener);
public void remove<EventListenerType>(<EventListenerType> theListener);
```

where the `<EventListenerType>` must implement the `java.util.EventListener` interface. By default, events are assumed to be multicast, unless the `add` accessor is declared to throw a `java.util.TooManyListenersException`. A couple of examples might be useful. Here is a unicast event called `Update`:

```
public void addUpdate(Update theListener)
             throws java.util.TooManyListenersException;
public void removeUpdate(Update theListener);
```

Here is a multicast event called `Click`:

```
public void addClick(Click theListener);
public void removeClick(Click theListener);
```

Explicit Specification

You can make your Application Builder bypass the Reflection process altogether, if necessary. Say you have a bean that requires an entire program (such as a Wizard) to be customized. A simple property editor panel is just not sufficient. You can hook your wizard into your Bean by creating a `BeanInfo` class. If you have a Bean called `MyBean`, then you need to create a class called `MyBeanBeanInfo` that implements `BeanInfo`, like this:

```
public class MyBeanBeanInfo implements BeanInfo {…}
```

`BeanInfo` provides access to all kinds of information through its methods. By overriding these methods, you can expose whatever information you want for your bean. The information includes the following:

- `BeanDescriptor`, which provides basic information about your Bean. A very important item that a `BeanDescriptor` can return is a `Customizer`. The folks over at JavaSoft have put a great deal of effort into ensuring that Java Beans could be arbitrarily customized, without imposing unreasonable burdens on simple Beans. A `Customizer` allows you to specify a complete GUI component, such as a `java.awt.Panel`, to customize a Bean.

- An array of `PropertyDescriptor` objects, giving a complete description of each property available in your Bean and in all of its ancestor classes.

- An array of `MethodDescriptor` objects, giving a complete description of each method available in your Bean and in all of its ancestor classes.

- An array of `EventDescriptor`, giving a complete description of each event available in your Bean and in all of its ancestor classes.

Providing a `BeanInfo` object doesn't in itself disable the Reflection mechanism. A well-behaved Application Builder (one that follows the Beans spec) is expected to check for `BeanInfo` objects before launching into Reflection. The Beans spec can't prevent a tool from using Reflection in the presence of a `BeanInfo` object, or force a tool to use Reflection. The Beans spec provides only the objects necessary to locate information – it doesn't force one behavior or another onto Application Builders. Obviously vendors are well-advised to follow conventions, but the Beans spec is not designed to force them, or even check that they do.

Class meta-data

Among the new features added to Java since JDK 1.0.2 is a change in the special class `java.lang.Class`. The new class has been extended with methods to return detailed information about a given class, so builder tools can query `java.lang.Class` and use the information to populate forms showing the user a class' constructors, methods and fields. The addition was made not only for the benefit of Application Builders, but also as part of the Java language support for serialization. The following code excerpt shows some of the interesting features of `java.lang.Class`:

```
public final class Class implements java.io.Serializable {

    public Field[] getFields() throws SecurityException {…}
    public Method[] getMethods() throws SecurityException {…}
    public Constructor[] getConstructors() throws SecurityException {…}

    public Field[] getDeclaredFields() throws SecurityException {…}
    public Method[] getDeclaredMethods() throws SecurityException {…}
    public Constructor[] getDeclaredConstructors() throws SecurityException {…}
}
```

The first 3 methods return information about public items of a given class (or interface) and all its ancestors. For example `getMethods()` returns an array of `java.lang.reflect.Method` objects for each `public` method declared in a class.

The second 3 methods return information about all items (`public`, `protected`, `package` and `private`) of a given class or interface, ignoring any ancestors. For example `getDeclaredFields()` returns an array of `Field` objects for every field in a given class, ignoring fields inherited from ancestors.

The meta-class information returned by `java.lang.Class` is encapsulated in the new classes `java.lang.reflect.Method`, `java.lang.reflect.Field` and `java.lang.reflect.Constructor`. These classes allow complete visibility of field types, method return types, parameter types and so on. Application Builders can use `java.lang.Class` to get a complete description of everything that makes up a class. For example, the following code displays in a list box the fields of `java.awt.Button`:

```
ButtonControl myButton = new ButtonControl();
Class myButtonClass = myButton.getClass();  // get the object's
                                            // java.lang.Class object
java.lang.reflect.Field[] myButtonFields = myButtonClass.getDeclaredFields();

for (int i = 0; i < myButtonFields.length; i++)
  listOfFields.add("Field ["  +
                   myButtonFields[i].getName() +
                   "] type = [" +
                   myButtonFields[i].getType() + "]");
```

The first 5 lines in the list box are:

```
Field [orientation] type = [int]
Field [imageFirst] type = [boolean]
Field [image] type = [class java.awt.Image]
Field [imageName] type = [class java.lang.String]
Field [label] type = [class java.lang.String]
```

The following code snippet fills a list box with a `ButtonControl`'s methods, their return types, and their parameter lists:

```
ButtonControl myButton = new ButtonControl();
Class myButtonClass = myButton.getClass();
java.lang.reflect.Method[] myButtonMethods = myButtonClass.getDeclaredMethods();

for (int i = 0; i < myButtonMethods.length; i++) {
  list1.add("Method " +
             myButtonMethods[i].getName() +
             " return type = " +
             myButtonMethods[i].getReturnType() );
  Class[] parameters = myButtonMethods[i].getParameterTypes();
  if (parameters.length == 0)
    list1.add("Parameter list: none");
  else {
    list1.add("  Parameter list = ");
    for (int j = 0; j < parameters.length; j++)
      list1.add("    " + parameters[j].getName() );
  }
  list1.add(""); // add a blank line after each method
}
```

The following shows the first few lines of output from the program:

```
Method setOrientation return type = void
```

```
 Parameter list =
  int

Method getOrientation return type = int
 Parameter list: none

Method setImageFirst return type = void
 Parameter list =
  boolean
```

Meta data is a wonderful thing to have at run-time. C++ programmers were forced to use macros to get it. Not only do macros differ across vendor implementations, but often make code very hard to read and even harder to debug. I wish I had a penny for every programmer that accidentally single-stepped into a Visual C++ message map macro, only to waste hours looking up useless macro details in the documentation. Java gives you standard meta-data right out of the box, making it much simpler to support commonly used features like serialization. Keep in mind that meta-data support was added to the core JDK, so meta-data is available for any Java object (not just Beans).

Property Editors

Many Beans have properties that require special handling, meaning they can't be set by simply typing some text or selecting a value from a list. For example a fancy grid Bean with a `layoutOptions` property might want to display a picture of a grid. Clicking on specific areas of the grid might allow the user to turn on/off layout features, such as the row lines, the column lines, the column headers, etc. You would probably implement this editor as a `java.awt.Panel`, with embedded bitmaps and other controls. The Java Beans spec allows you to hook your own Property Editors into the system by defining an editor class for your special properties. An Application Builder locates the editor by using a simple design pattern: for a property `MyProperty`, it looks for a class named `MyPropertyEditor`. Actually, things are slightly more complicated. To keep application builders and vendors all on the same page, the JavaSoft folks created a standard class to locate property editors: `java.beans.PropertyEditorManager`. The Beans spec allows you to *register* editors for your own custom properties. The registry they are put into is not some hairy operating-system-level monster like the Windows Registry. It's simply a `java.util.HashTable` that stores key-value associations. The *key* is the class you want to register an editor for. The *value* part is the editor class. The hash table is a field of `PropertyEditorManager`. The Beans `PropertyEditorManager` tries to locate an editor for a class in three steps:

1. It checks the `PropertyEditorManager` registry. You register editors by calling the method

   ```
   registerEditor(Class targetType, Class editorClass)
   ```

   of class `java.beans.PropertyEditorManager`. Calling the method a second time for a given editor unregisters the editor.

2. It searches for a class whose name is the same as the class, and ends in "Editor". For a class `Color`, it searches for the class `ColorEditor`.

3. It searches for a class whose name is the same as the bean that owns the property, and ends in "Editor". For example, assume you have a bean `MyBean` with a property of type `Longitude`. The Beans `PropertyEditorManager` will search for the class `MyBeanEditor`. The search path defaults to `sun.beans.editors`, but you can change it by passing an array of strings to `java.beans.PropertyEditorManager.setEditorSearchPath`.

The BDK comes with built-in editors for all the standard types. They form the following hierarchy:

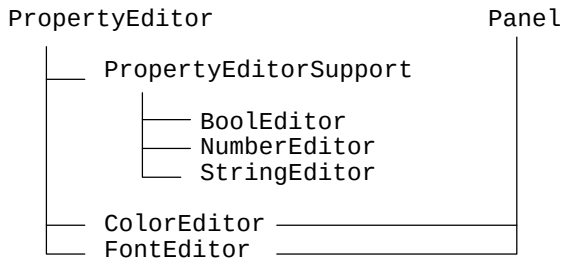

Figure 6 - The built-in property editors.

Editors come in 3 basic flavors: String, choice and custom. A String editor lets you type in a value for the property, using an Edit control. The following figure shows a string editor for the title property of a Frame, as shown in the JBuilder Property Inspector:

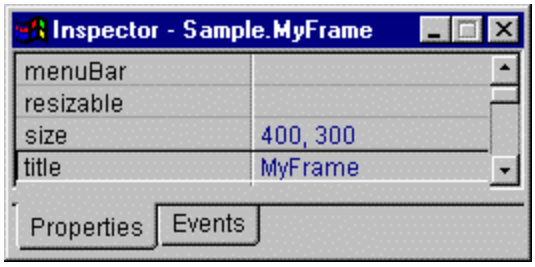

Figure 7 - A string editor in JBuilder.

A choice editor presents the user with a list of options to choose from. Boolean values are simple properties edited using this type of editor, which looks like this in JBuilder:

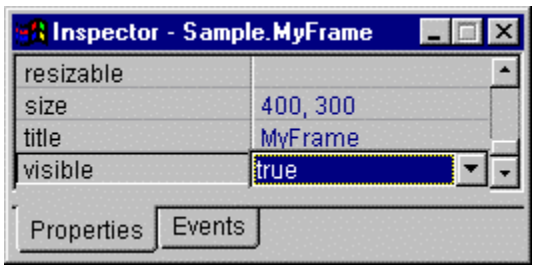

Figure 8 - A choice editor in JBuilder.

A custom editor uses its own panel, so it doesn't show up in the Property Inspector window. A custom editor lets you display anything your bean needs. A custom editor is indicated in the JBuilder Property Inspector by an ellipsis button, like this:

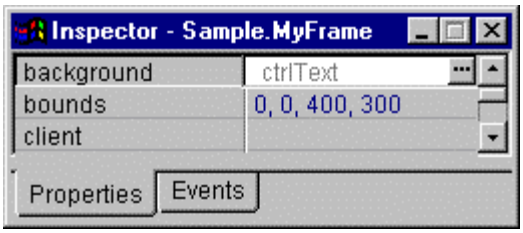

Figure 9 - The ellipsis button, indicating a custom property editor.

Clicking the ellipsis button opens the full property editor. The `ColorEditor` is an example of a custom editor. Under JBuilder, the editor looks like this:

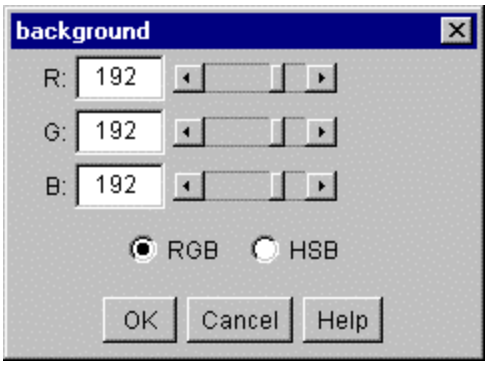

Figure 10 - The ColorEditor displayed under JBuilder.

Customizing a Java Bean

In C++, the only way to customize a class was to derive a new class from it. This is a fine model, but often there are cases when you need to change a class in a minimal way. Creating an entirely new class is a lot of work – in fact too much work for something as trivial as changing the text color of a button from black to red. Java builds on the lessons learned with Visual Basic and Delphi to allow trivial changes to be accomplished through persistent properties. To create a button with red text bean, you just take the standard one, change its color and save it. Changing properties is certainly the easiest way to customize a Bean, and is often done within the context of an Application Builder, with a property editor. Figure 11 shows the property inspector for a JBuilder button.

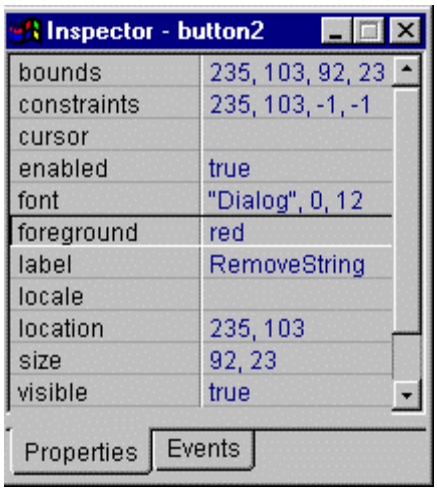

Figure 11 - Changing a simple bean property using properties.

The next level in customization that doesn't require deriving new classes is through event handlers. Beans use the good old event-delegation model -- a fancy expression for a simple concept. In C++, if you have some class to which you want to add an event-handler you are forced to derived a new class to handle the event. As with properties, deriving classes is fine and well if you intend to add a significant amount of functionality, but for simple changes it is just too much work. Say you want your red button to be used to insert the text in an Edit control into a list box. The hard way is to derive a `RedButton` class from `java.awt.Button` and add a `mouseDown` handler, something like this:

```
public class RedButton extends java.awt.Button {
  public boolean mouseDown(Event evt, int x, int y) {
    //…do something
  }
}
```

The easy way is to use delegation. Instead of deriving classes, you delegate event processing to another class. Stated differently, why do the work in one class if there is another class that can do it for you?

Creating a new Java Bean

If you develop applications using Java Beans, probably the most central activity will be creating new Beans. Often you will be able to get by using built-in beans and changing their properties. You can build all sorts of dialog boxes using standard JBuilder beans. When you can't get the needed features this way, it's coding time. You have two options for creating beans: starting from scratch and starting from an existing bean. Starting from scratch is quite a bit more work, since you can't leverage any other class' code. It is by far more desirable to start from existing bean, if at all possible. The whole idea of object-oriented programming is reuse. Reusing code minimizes your effort and lets you start from a base or (supposedly) debugged code. Fortunately, JBuilder comes to your aid when creating new beans. With its built-in wizards, it takes most of the cut-and-paste grunt-work out of the process.

Creating a bean from scratch

There aren't too many applications out there that can be built entirely using existing beans, or classes derived from them. It doesn't take a lot of imagination to think of a situation. Take the good old `Employee` class that

lurks the pages of practically every book on OOP. What bean would you create an `Employee` from? Probably none, so you create `Employee` from scratch. Exactly what *from scratch* means will depend on what your `Employee` does. If your want it to show up as a displayable object on forms, then you will probably derive it from `java.awt.Component`. At the very least, you derive it from nothing, which by default makes `java.lang.Object` the ancestor.

The easy way to create your new beans is via a wizard. JBuilder has one that looks like this:

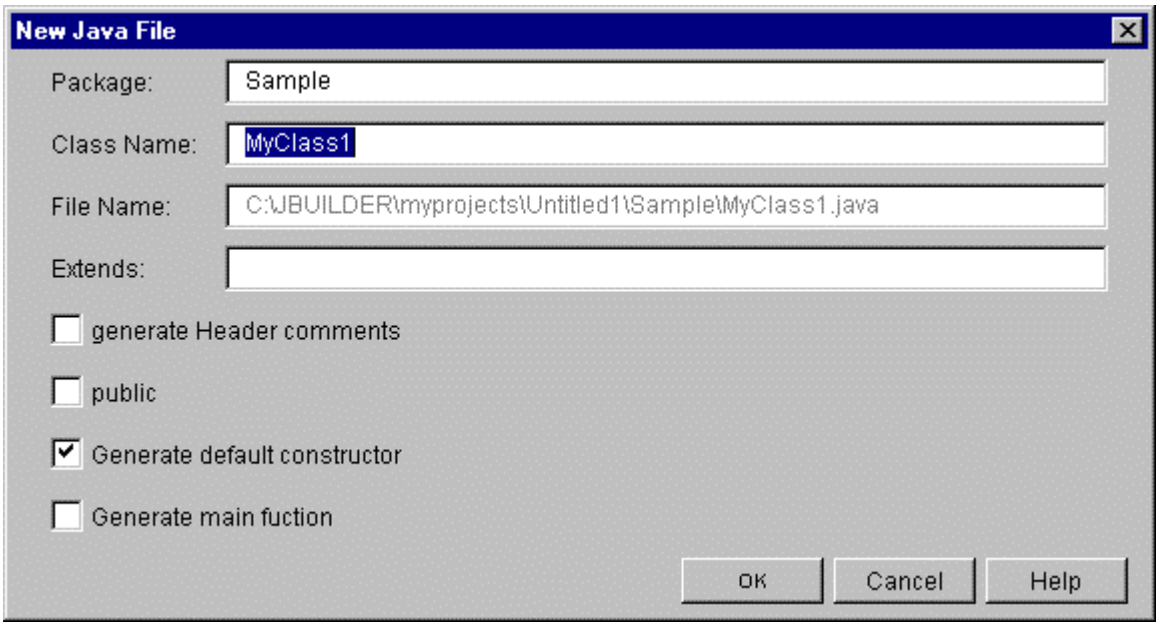

Figure 12 - The JBuilder wizard for creating new beans.

By leaving the **Extends** field blank, the wizard creates a class with `java.lang.Object` as the ancestor, producing something like this:

```
package Sample;

class MyClass1 {
  public MyClass1() {
  }
}
```

Ain't much in there! You can expect to do a lot of work, if you plan to have the bean do much of anything. Most of the time, you'll be adding PME to your beans. That's Properties, Methods and Events. In case you dozed off after the paper's first paragraph, properties are fields that can be accessed through getter and setter functions, and can be shown on property sheets like JBuilder's Property Inspector. Methods are where all the code for your bean resides. Methods don't have any standard features that lend themselves to automatic generation, so you'll spend a considerable amount of time coding them. Events are special methods that have add and remove accessors. Events can be extracted from your bean with an Application Builder, and displayed in a Property Sheet. JBuilder displays the events for a bean on the Events tab of the Property Inspector.

Starting from an existing bean

You don't actually have to use a bean as the ancestor. Any Java object will do, although bean ancestors will obviously have more code to reuse in terms of Application Builder and PME support. The biggest choice to make is which class to derive from. Sometimes the choice is simple, but not always. Say you plan on creating a new clickable image. Do you inherit from regular awt classes like `java.awt.Button` and `java.awt.Panel`, or use beans like `borland.baja.control.ButtonControl` or `borland.baja.control.FancyPanel`? It all depends on what your bean is going to do. Once you've decided, the JBuilder Class Wizard helps you crank out the starting code for your bean. Say you decide to inherit from `borland.baja.control.FancyPanel`. Here's what your initial bean's code would look like:

```
package MyControls;

import borland.baja.controls.*;

class MyClickableImage extends FancyPanel {
  public MyClickableImage() {
  }
}
```

Again, not too much to start with, but remember there's all that code in the ancestor to reuse.

Unleash those wizards

Now you want to override some methods in the ancestor, such as `mouseDown`. JBuilder has all sorts of wizards to help you develop Java code. There is one that shows you all the ancestors of your class, with the methods available in each. All that information is made available thanks to the built-in Java meta data classes. Imagine all the information you get without even a hint of a macro! The JBuilder Override Wizard looks like this:

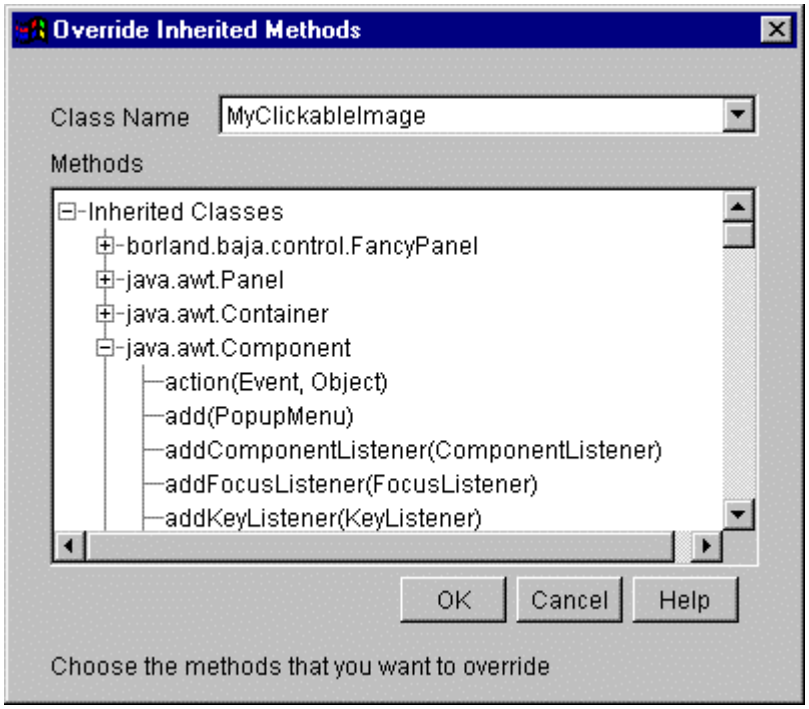

Figure 13 - The JBuilder wizard for overriding methods.

You check the ancestors and find a `mouseDown` event handler in `java.awt.Component`. You select it and presto! You get a new method for `MyClickableImage` that looks like this:

```
public boolean mouseDown(Event parm1, int parm2, int parm3) {
    //TODO: override this java.awt.Component method;
    return super.mouseDown( parm1,  parm2,  parm3);
}
```

Pretty standard stuff, if you're used to working with common C++ IDEs like Visual C++ or Borland C++. You implement the mouse handler, then you need a `Paint` method to dray your image. Back to the JBuilder wizard. You locate a `Paint` method in `borland.baja.control.FancyPanel`. You select it and you get a method looking like this:

```
public void paint(Graphics parm1) {
    //TODO: override this borland.baja.control.FancyPanel method;
    super.paint( parm1);
}
```

The wizard generates as much code as it possibly could, given the information provided. What if you can't find a method to override? No problem: you write a method that doesn't override anything. Nothing magic about it. JBuilder comes with a variety of different wizards, to give the most support it can for the various tasks you'll probably engage in. For example there is a multi-page Applet wizard that looks something like this:

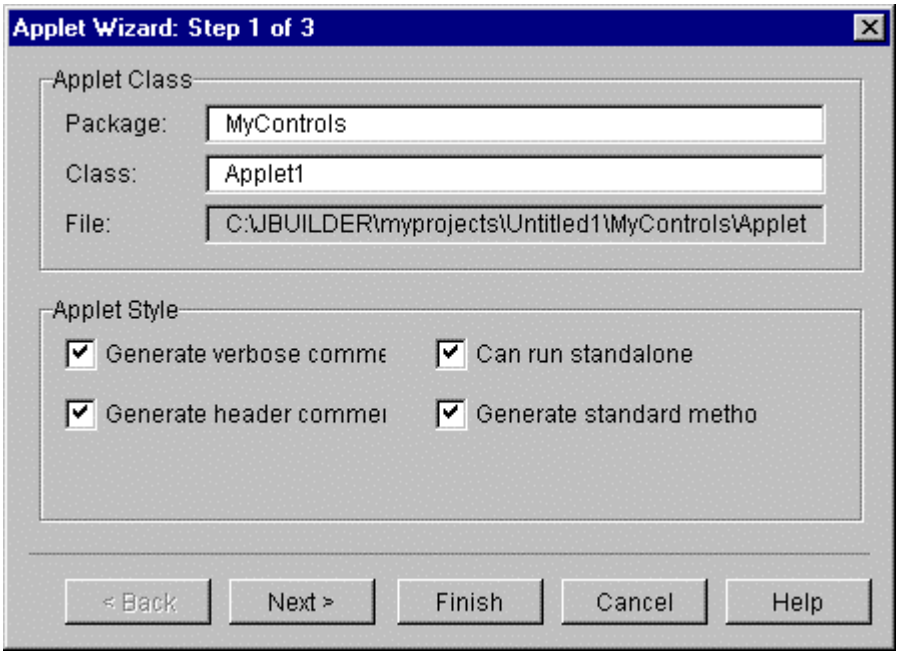

Figure 14 - The first page of the JBuilder Applet wizard.

There is also a multi-page Application Wizard, for creating brand new applications. The first page looks like this:

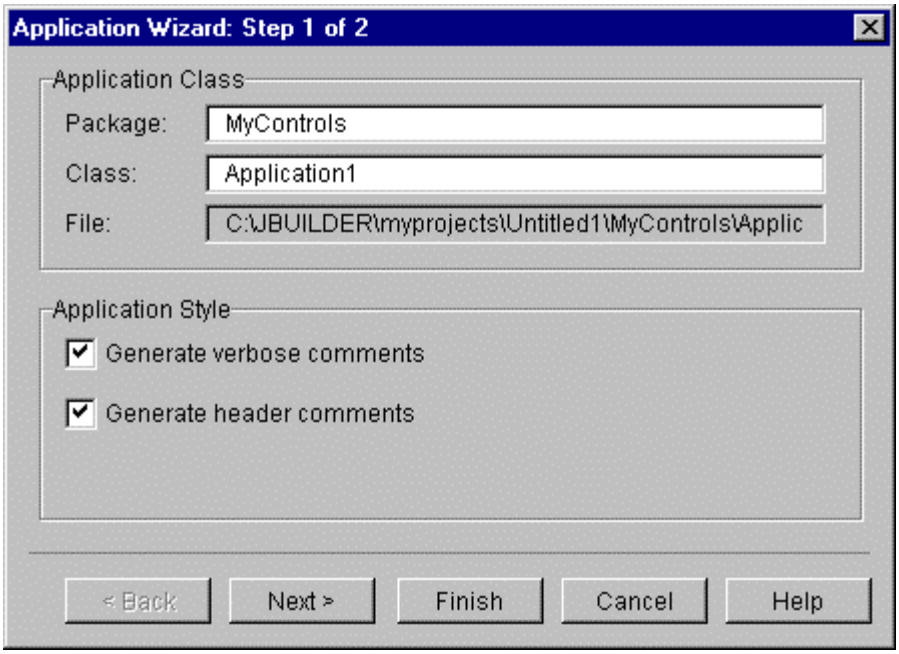

Figure 15 - The first page of the JBuilder Application Wizard.

There is yet another wizard to help you implement an interface in your class. It looks a lot like the Override wizard:

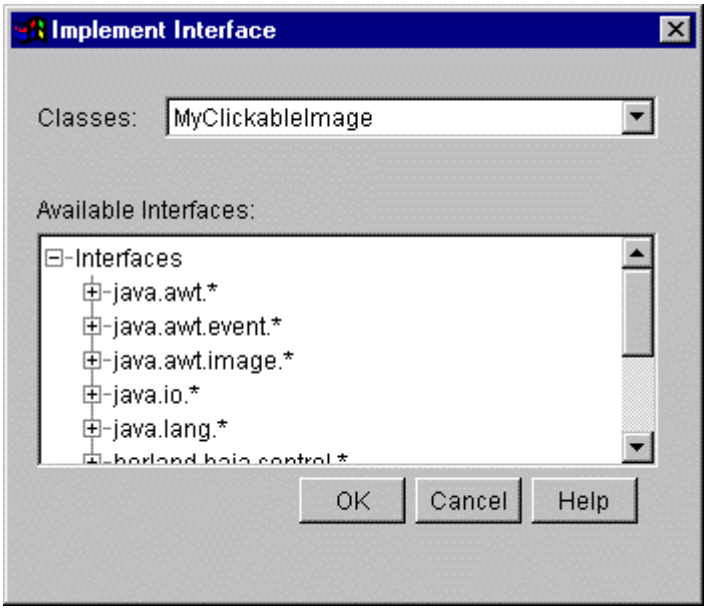

Figure 16 - The JBuilder Interface Wizard.

The idea here is that you search for an interface by expanding the items in the **Available Interfaces** list. When you select an interface, JBuilder adds some code to your Java object, which doesn't necessarily have to be a Java Bean. Here is how class `MyClickableImage` looks after implementing the `java.awt.Shape` interface:

```
class MyClickableImage extends FancyPanel implements Shape {
  public MyClickableImage() {
  }
  public Rectangle getBounds() {
    //TODO: implement this java.awt.Shape method;
  }
}
```

The wizard not only adds the selected interface to the list of ancestors, but also adds stubs for each method inherited from the interface. Not rocket science, but definitely a useful feature that saves you from having to manually lookup all the methods for each interface you implement.

The last wizard is the Project Wizard, a multi-page beast whose first page looks like this:

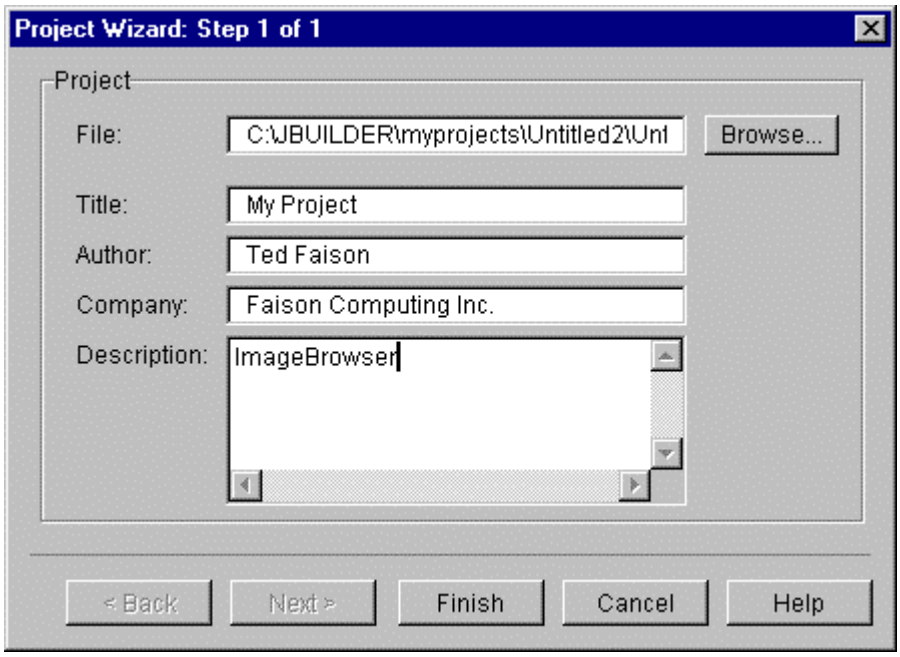

Figure 17 - The first page of JBuilder's Project Wizard.

The project wizard works similarly to the AppWizard of Visual C++ or the AppExpert of Borland C++. Again, nothing absolutely new here. JBuilder hasn't invented everything from scratch. It builds on the lessons learned with from Borland C++ and Delphi.

Comparing Java Beans with ActiveX

If Java Beans is the specification for a new component architecture, it makes sense to see how it compares with other existing component models. Because ActiveX is the one most widely used in the Windows world, I'll discuss briefly how Java Beans and ActiveX compare.

For starters, ActiveX is too complicated! A two-year-old could tell you that much. ActiveX controls are built on top of the Microsoft Component Object Model (COM). If ActiveX is based on a first generation component model, Beans is definitely 2nd generation. COM objects were designed from the start to be hand-programmed. COM objects don't possess any meta-data, and contain little or no information that might help an Application Builder. COM objects don't have functions that return a list of the interfaces it supports. No functions returning a list of the functions supported by a given interface. No one to call to get parameter list descriptions or other valuable information. The OLE approach is to use separate binary files, like Type Library (`.tlb`) files. Because the `.tlb` file must be generated using a special stand-alone compiler, there is the risk you (the component develop) forget or even decide not to create a `.tlb` file for distribution. See how many Type Library files you can find for the Microsoft products that are OLE servers (like Word, Excel et similia)... Moreover, even if you had a Type Library, good luck in writing a quick function to decode its contents.

Okay, so ActiveX is complicated. Do users other than developers care what goes inside a component ? You bet, because where complicated objects go, big ones follow. These days of Web-mania and anemic modem bandwidths, web surfers are very conscious about page download times. If you stick a hairy old ActiveX control on a page, chances are the download time will shoot from seconds to minutes. The truth is that ActiveX controls are big. Just go check the size of your favorite 10 ActiveX controls, and tell me how many are smaller than 100K. You'll probably find a few in the megabyte range. Moreover, the size of an ActiveX DLL may only be part of the control's actual footprint. The control probably uses a slew of other DLLs. It is quite common for ActiveX controls to pull in support DLLs extending the memory footprint into the megabytes. In boxing terms, ActiveX is definitely a heavyweight, but big isn't better in the software business.

In contrast, Java objects in general, and Beans in particular, were designed with small footprint in mind from the very beginning. Beans are frequently in the 5-10 KB size. How is this possible? Simple, Beans get a big boost from the Java runtime environment, which eliminates the need to add trivial stuff like file I/O or system-level routines to each and every Bean. Drawing support, network support, file support… it's all provided at runtime by the Java environment, or by a small number of tiny Java objects. You don't have to statically or dynamically link a zillion libraries with your application. As a simple example, take a look at the sample program `java\demos\MoleculeViewer\XYZApp.class` in the JDK directory. It is a viewing program that displays one of three organic molecules in 3D. Using the mouse, you can rotate the molecules in 3 dimensions. The whole program is about…12KB. You can't much more light-weighted than that! Reminds me of the days of Forth, that good ol' language that compiled down into almost nothing. How big would the `XYZApp` program be if done with ActiveX controls? I rest my case.

You want more? Let's talk security. You download a Web page with embedded ActiveX controls. Your browser pops open a dialog box that looks something like this:

Figure 18 - The Code-Signed Certificate used by downloaded ActiveX controls.

The system is asking you whether you trust an ActiveX control embedded on the Web page. What do you do? How do you know you can trust it? Assuming you would entrust your life to the company that created a given control, what if the control were buggy? Maybe it has a bug that crashes your system on every 57th keystroke. Maybe it contains a virus that waits for you to run your home-banking program. When you do, it silently withdraws some money from your account and sends it to a European bank. Don't laugh: a German hacker group publicly demonstrated such a virus-infected ActiveX control recently

The big problem with ActiveX security is that it is based on the notion of trust and trustworthiness, stuff hard to come by in these days of viruses. Even worse, Microsoft puts the burden of decision of trust on the end user. It would be bad enough if all end users were software techies, but most users don't have a clue about technology. It is scary enough for them to have to boot their computer. I'll bet my grandmother wouldn't tell the above dialog box from a New York Stock Exchange certificate.

The fact is that no one can tell you for sure whether an ActiveX control will cause trouble in your system or not, and I don't care who the publisher is. Java, on the other hand, was designed to deal with security from the *very* beginning. The entire language is built on top of the so-called *sand-box security model*. When you run Java code that was downloaded, a virtual sand-box is placed around the program. Every instruction of that program that deals with sensitive areas, such as files or system calls, is runtime checked by the *Java Security Manager*. This guy makes a drill sergeant look tame. Nothing, and I mean nothing gets done unless the Security Manager approves it. You, the end user, have nothing to do with any of these decisions. If your Java applet originated from outside your system, it is sand-boxed, otherwise it can do anything it wants. I'll let you decide whether you prefer the ActiveX or Java security model…

Next feature: platform independence. This one's a no-brainer. ActiveX was designed for the Windows platform. Period. There are packages that let you run ActiveX code on Macs, but you take a runtime hit, and

runtime performance is really the only areas in which ActiveX outshines Java. At least on the Windows platform, ActiveX controls are fast. Just as fast as any other code running on the machine. Java code is normally not as fast, unless you have a Just-In-Time (JIT) compiler. Using JIT, the system compiles a Java object's bytecode into native machine language when it is loaded. There is a small up-front penalty for the compilation, but after that your object runs at native speed. To be perfectly honest, Java programs generally run quite a bit slower than native-compiled programs. Even with JIT compilers, the overall speed of a Java program is lower than an equivalent ActiveX control. On the other hand, Java is truly platform-independent. Java code runs anywhere there is a Java Virtual Machine (JVM) and you'll have to look pretty hard to find a major platform for which a JVM isn't available.

Conclusion

To wrap this whole discussion up, Java Beans are here and here to stay. Beans are not out to eliminate the competition (read ActiveX, CORBA and company), but to give developers the means to use an advanced component model where runtime performance is not at a premium. You can fallback on native models like ActiveX where speed is essential. Java Beans represent a generation leap in software development, primarily because the spec was written specifically in favor of Application Builders. Beans allow you to develop high quality software quickly, because they bend over backwards to help your favorite Builder make your software development job easier. The Java language was designed from the start to be simple and lightweight, and the Java Beans spec follows that same tradition of elegance.