Shell Programming—JavaBeans Style

By Jim Knutson

Java is being used with increasing frequency on UNIX systems for smart Web forms, backend servlets, GUI frontends to applications, and Web-enabling enterprise applications. This article compares and contrasts Sun's JavaBeans component model with shell programming to show how easy it can be to build up Java applets and applications with very little knowledge of programming.

Shell programming is a rudimentary form of component programming. Many of the features available in today's component models have been in use since the early '70s when UNIX® first appeared. This article will cover concepts and terminology related to component programming with the JavaBeans component model and show their similarity to shell programming. The natural progression from shell to component programming is learning how components can be snapped together easily.

Component Programming in Shell Code

Shell programming makes it possible to compose multiple commands (components) together to perform a sequence of actions. The JavaBeans component model does the same.

Suppose we wanted to offer a menu-driven interface to allow users to talk to each other. A first-pass shell program might look similar to the following:

```
#!/bin/sh
who | sed 's/ .*//'
echo "Enter user: "
read user
talk $user
```

Note that the `who`, `sed`, and `talk` commands are being glued together using a scripting language (the scripting language of the Bourne shell). The JavaBeans component model supports two methods of connecting bean components: scripting and event connections. JavaBeans does not include scripting language support, but several bean-builder tools support the use of scripting to compose beans together. Most of these bean-builder tools currently support only a single scripting language (usually Java™), but IBM's BeanExtender technology offers support for multiple scripting languages, including Java and NetRexx, with the capability of adding other scripting languages.

In the shell script, the components perform these functions: data access, data filtering, and presentation. Though not required, beans typically separate those functions into separate components, which reduces the runtime requirements when presentation is not needed. It also allows for different styles of presentation components to plug in and display the information. Similar to UNIX commands and filters, bean components work best when designed to do one thing and do it well.

Jim Knutson

A third consideration is how component execution can be altered. Shell commands take arguments to affect their execution. JavaBeans offers a similar function through properties. However, properties offer a much richer function for controlling components than simple arguments. Any component interested in determining when the value of a property in another component changes can listen for the change. It can veto that change as well.

In shell programming two of the components are exchanging data using the standard I/O paradigm of a single data producer and single consumer. JavaBeans has similar concepts for I/O, but the typical way to exchange data is through Events. When a bean component has altered its state or wishes to make data available to another component, it generates an event with the data in the event or available for access from a property (which could also generate an event when its value changes). This event can be distributed to an arbitrary number of consumers.

In shell programming, it is difficult to capture the state of a component so that it can be restarted from that state at a later time. In JavaBeans, it is not only easy to do, but quite natural. Anyone who has dealt with the state management of the UNIX accounting system will likely see the advantages.

Finally, the way input is obtained from the user differs. Nothing in the JavaBean component model predefines the method of obtaining input. Suffice it to say that a component could perform the function of gathering input.

The Beans Component Alternative

How would we use beans to compose something that performed a function similar to the above? Assume a bean, called `WhosOn`, knows how to obtain a list of users on a system, and another bean, called `TalkToMe`, knows how to talk to a user. Using the BeanExtender technology from IBM, we would find those two components and drop them onto the visual assembly surface. We also need a way to present the list of users on the system; to do that, we use a presentation component related to `WhosOn` called `WhosOnViewer`.

The `WhosOn` component is designed so that it constantly monitors the system and generates a `WhosOnEvent` every time someone logs on or logs off the system. The `WhosOnEvent` has an array of users currently on the system as part of its data. The `WhosOn` component has a corresponding presentation component called `WhosOnViewer`, which consumes `WhosOnEvents`. It is a simple matter of point and click using BeanExtender's event connection view to connect the source of an event to any number of event sinks. See Figure 1.

The close relationship of events, event producers, and event consumers brings type safety into connecting components together. You can be relatively assured that a consumer of an event will know exactly what to do with the event data given to it.

The component, as written so far, will give us a real-time view of the users who are logged on. Now we need the ability to choose the users to talk to, then talk to them. Fortunately, the `WhosOnViewer` component generates a `UserSelectedEvent` whenever an item in its view is selected. This time we use NetRexx scripting to take a `UserSelectedEvent`, get the value of the event's user property, and set a property on the `TalkToMe` component before invoking a method to open the connection to the other user. See Figure 2.

This brings us to our next topic: What happens if the person refuses to talk to us?

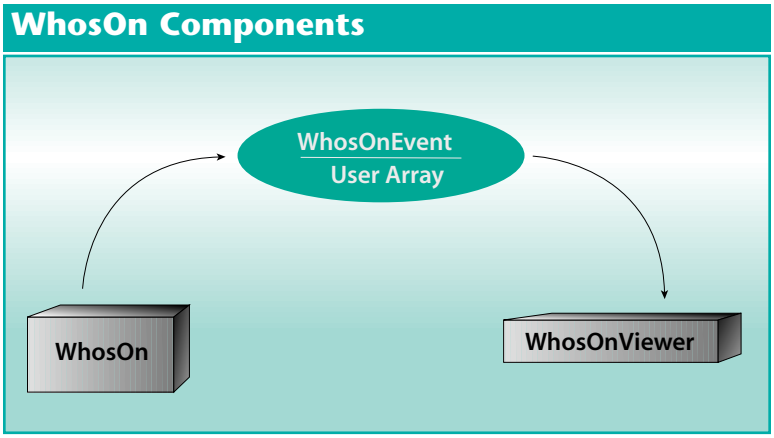

Figure 1. WhosOn components

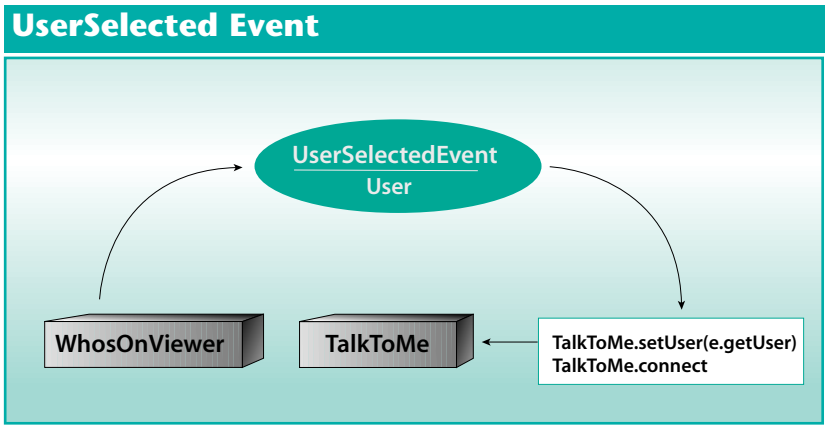

Figure 2. UserSelected Event

In shell programming, components relay failure information through return codes and signals. Java-based components communicate failure information through exceptions, which is similar to signal handling in shell programming.

When using pre-built components, you will most likely encounter this situation when writing script code using a class that throws exceptions. As a component programmer, you must decide whether to handle the exception directly in your script code using a try-catch block or pass it on to someone else. Usually you will need to handle it in your script code.

Some Java language features affect bean component programmers; for example, Java components may be used concurrently. Just as shell programming supports the notion of concurrent programming through the use of background processes, Java supports multiple threads of execution. This is easy to do, and Java programmers should expect their components to run multithreaded. Careful use of synchronization can avoid problems associated with running in a multithreaded environment.

How do environment variables in shell programming translate into beans component programming? Environment variables are often used as a persistent method for defining arguments to shell commands. Java supports a similar notion using "system" properties, which can be made persistent across invocations of Java applets and applications. Examples of these can be found in the Java Development Kit (JDK) library directory.

Bean Husbandry 101

If bean component programming is so easy, particularly with bean-builder tools such as IBM BeanExtender, why hasn't it become even more popular? The simplest answer is time.

The JavaBean specification, which describes how to write beans so that they can be composed, was not available until early this year. In addition, many Java developers do not understand what they need to do to turn their classes and applets into beans that can be composed with other beans.

Lastly, tools support for beans has been, until recently, nearly non-existent. We shall examine one more example to show how easy it is to make bean components, particularly with bean-aware builder technology such as IBM BeanExtender.

A common occurrence in shell programming is a loop that performs a repeated periodic function. The shell code typically looks like the following:

```
while true; do
  <function to perform>
  sleep <some interval>
done
```

Although this works fairly well, it cannot guarantee that the function is performed at specific intervals. It only

guarantees that an interval amount of time has occurred between the time when the last round of function ended and a new round starts. It is easy to build a bean component to provide similar functionality by using IBM BeanExtender.

Figure 3. Defining a new property

Figure 4. Adding a new event

Our new `Ticker` component will generate an event at specific intervals defined by a property that can be set. It runs as a separate thread; therefore, it is derived from the `java.lang.Thread class`. With BeanExtender, we can set the parent class of the component we are building by simply typing in the name of the parent class.

Next, we need to define a settable property to govern the interval between events. The Publish view of BeanExtender lets us define new properties easily by defining a name and type for the property. In this case we will define a property called *delay* with a type of long to represent milliseconds between event generation, shown in Figure 3. We also decide to be good bean citizens and make the property bound and constrained so that other components can be notified of an event change and optionally veto the change.

Next, we decide the delay needs to default to one second, so we select the constructor method, Ticker, in the Publish view for editing, and add the line `delay = 1000`, to the constructor.

To generate an event, we must first define an event to generate. The Publish view lets us create events as easily as we create properties. We generate a new event called `TickerEvent`. We decide that we do not need to supply any data items with the event and the method, which all sinks of the `TickerEvent` will need to implement to handle the event, will be called `tickerTocked`, shown in Figure 4. Defining the event allowed us to do the following:

- Generate the `TickerEvent` class
- Generate the `TickerEventListener` interface, which is implemented by components that want to be `TickerEvent` sinks
- Add a `fireTickerTocked` method to our Ticker bean component to handle the distribution of the event to everyone wanting to hear about it

The final task to complete our component is to write the code that actually generates the event. To do this, we publish a new public `void run()` method with the code shown in Figure 5.

This method overrides the run method of the `Thread` parent class and defines the function for the thread to perform. In this case, wait until the appropriate time to generate a new event and then fire it.

```
while(true) {
  try {  sleep(delay);  }
  catch (java.lang.InterruptedException e) { }
  fireTickerTocked(new TickerEvent(this));
}
```

Figure 5. Code to generate event

Conclusion

As you can see, with the appropriate tools and a little knowledge, it is easy to create a bean component within a short time. In this example, we created a new fully functional bean component supporting properties and events with six lines of code and a couple minutes of work. This new component can then be used to compose with other components to provide higher level function. One example of this is to take a few progress bar beans arranged vertically, along with some script code and a Ticker bean, to implement a visual representation of an egg timer. Another possible use for the Ticker bean would be to drive a ticker tape component.

BeanExtender is a technology preview available from IBM providing support for beans programming and deployment. It continues to mature and pieces of the technology are expected to appear in several IBM products.

Evaluation copies of BeanExtender can be found on the IBM VisualAge™ WebRunner home page at: *http://www.taligent.com/Products/webrunner/webhome.html*.
Additional information regarding BeanExtender can be found on the IBM BeanExtender Technology Web page at: *http://www.software.ibm.com/ad/javabeans/beanextender/*.

Jim Knutson, *IBM Corporation, 11400 Burnet Road, Austin, TX 78758. Mr. Knutson is the technical lead and a development manager of the BeanExtender development group. He has worked at several companies, including the Microelectronics and Computer Technology Corporation (MCC) before joining IBM in 1993. He has a BA in Computer Science and 20 years of computer industry experience.*