A Comparison of Component Integration between JavaBeans and PCTE

Fred Long
University of Wales
Aberystwhth, UK

Robert C. Seacord
Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA

1 Introduction

One of the latest approaches to software development is “component-based integration” [1] in which an application is composed by integrating a number of, usually small, software components. A number of component-based integration technologies have appeared, for example CORBA, ActiveX, JavaBeans. These technologies offer great promise.

However, only a few years ago there were other integration technologies that were also promising great things but, in the event, have turned out not to be commercial successes. For example, repository-based integration and broadcast message systems.

To gain some experience of the new technologies, we compared the implementation of a small component integration application using JavaBeans with how it might have been done using PCTE. This paper describes the experiment we carried out and reports our findings and conclusions. Section 2 provides background to JavaBeans and PCTE. Section 3 describes the application that we experimented with, reports on its implementation in Java, and discusses how it might have been implemented on PCTE. Section 4 compares the two approaches, and section 5 summarizes our findings.

2 Background

2.1 JavaBeans

JavaBeans [3] is the latest thing on the component integration scene. JavaBeans is the name given to a set of APIs and naming standards that enable Java components to be manipulated in a uniform way. JavaBean internals may be examined using the Java Introspector class. This class provides a standard way for tools to learn about the properties, events, and methods supported by a target so that tools can manage the process of integrating a number of JavaBeans in a uniform way. The platform independence of Java leads naturally to the cross-platform integration of JavaBeans.

The three most important features of a JavaBean are the set of properties it exposes, the set of methods it allows other components to call, and the set of events it fires.

Properties are named attributes associated with a bean that can be read or written by calling appropriate methods on the bean. For example, a bean might have a “foreground” property that represents its foreground color. This property might be read by calling a `getForeground` method that returns `Color` and updated by calling a `setForeground` method that accepts `Color` as an argument.

The methods a JavaBean exports are just normal Java methods that can be called from other components or from a scripting environment. By default all of a bean’s public methods are exported, but a bean can choose to export only a subset of its public methods.

Events provide a way for one component to notify other components that something interesting has happened. Under the JDK 1.1 AWT [5] event model an event listener object can be registered with an event source. When the event source detects that something interesting happens it calls an appropriate method on the event listener object.

The BeanBox is a very simple test container bundled with the Beans Development Kit (BDK). The BeanBox is similar in concept to Microsoft Visual Basic, SEI Serpent [2], or other GUI builders that allow a user to graphically compose the user interface of an application and programmatically specify the behavior.

Beans can be added to an application in the BeanBox by selecting the bean's name or icon in the “Tool Box” window and clicking on a location in the composition window. Public properties of the Bean can be edited using the “Property Sheet” window. For each editable property the “Property Sheet” window shows the name of the property and its current value.

The BeanBox allows events from the currently selected bean to be connected with a target event handling method on any other bean. The “edit” menu on the central composition window has an “events” menu that has a sub-menu for all the different kinds of events that the currently selected bean fires. These events are grouped and named according to their `EventListener` interfaces.

If you select one of the events from this menu, the BeanBox draws a “rubber band” line from the source bean. You can then click on the border of the target bean to which you want the event delivered. The BeanBox then checks for methods in the target bean that *accept the same EventObject argument* that is fired by the source event. The user is then offered a dialog box to chose which matching method should be called when the source event is fired. After a method is selected, the BeanBox generates, compiles, and loads an event adaptor class to connect the source bean's event to the target event handler method.

2.2 PCTE

The Portable Common Tool Environment (PCTE) [7] is an example of the repository-based integration technologies developed in the 1980's. PCTE was developed as a tool support interface specifically to provide a tool integration infrastructure in software engineering environments. PCTE is defined as a functional interface to an entity-relationship-attribute type database. As well as the tool support facilities themselves, PCTE provides extensive support for the definition and manipulation of the database schema that is necessary for any application built on PCTE.

Despite extensive support for PCTE, especially from Europe, and international standardization, PCTE has not been commercially successful.

3 The Application

For this little experiment we chose to implement an example given in [6]. It is a "plumbing" example, with components `WaterSource`, `Pipe` and `Valve`. The `WaterSource` component "drips" one `WaterEventObject` per second into all the components connected to it. A `Valve` component can be in one of two state, "open" or "closed". An open `Valve` passes on any "drips" it receives to all components connected to it. A closed `Valve` does nothing. A `Pipe` component behaves like an open `Valve`. Once these components have been implemented as JavaBeans, they can be instantiated and connected in arbitrary configurations in the BeanBox.

3.1 The Implementation in Java

The Java components were straightforward to code and instantiate as Beans. It was then very easy to connect the components into an arbitrary "plumbing" configuration using the BeanBox. However, the simplicity of the model means that unrealistic configurations may be constructed. For example, both ends of a `Pipe` component may be connected to the same `Valve` component, or two `Valve` components may be connected together directly. Furthermore, any number of components may be connected to any other component.

To explore our ability to constrain the JavaBean integration model, the application was modified to see how easy it would be to prevent some unrealistic configurations. Specifically, we wanted to 1) prevent connections between components of the same type and 2) limit the number of connections allowed by each component.

The first problem was solved by making the "couplings" (events and event listeners in Java terminology) different for the `Pipe` and `Valve` components. This meant that two `Valve` or `Pipe` components could not be connected together directly. It is also possible to provide different couplings for either end of a component.

For this solution to work, it was necessary to define two different event types due to the adeptness of the BeanBox at generating adaptors to map events from one bean to an event handling method on another. The BeanBox generates adaptors for any event handler that accepts a subset of the argument types generated by the event. By defining `Pipe` and `Valve` components so that they accepted different event types we were able to prevent them from being arbitrarily connected in the BeanBox. As a result, the water event changes "polarity" as it passes between each component in the plumbing.

The second problem of limiting the number of connections allowed by each component can be further refined to limiting a) the number of event listeners that can be registered at a time and b) the number of times the bean can be registered as an event listener. The former corresponds to water events generated by the component and the latter to the water events accepted by the component.

As a special case, JavaBeans checks to see if the `add`*`<EventListenerType>`* method throws the `java.util.TooManyListenersException`. If so, it is assumed that the event source is unicast and can only tolerate a single event listener being registered at a time.

So for example:

```
publWaterEventic void addWaterEventListener(WaterEventListener t)
throws java.util.TooManyListenersException;

public void removeWaterEventListener(WaterEventListener t);
```

defines a unicast event source for the `WaterEventListener` interface. We observed that the Version 1.0 BDK BeanBox we used in our experiments still allowed multiple event listeners to be added to the unicast bean, although it was specified as throwing the `TooManyListeners` exception. Throwing the exception, did prevent the `WaterEvent` from propagating to the event listener. However, the logic used to throw the exception could just have easily restricted the number of listeners to an arbitrary number.

We could discover no existing mechanisms to limit the number of times the bean can be registered as an event listener. It would potentially be possible to maintain a hash table with the number of times a bean has been registered as an event listener and to consult this table before registering a bean as an event listener but this mechanism is completely outside the existing JavaBeans component model.

An alternate approach would be to use the Java Core Reflection API [4] to access and modify fields of objects and classes that could be used to limit the number of connections allowed by an application. Again, this concept is not defined as part of the JavaBeans specification these constraints are not enforced by the BeanBox.

3.2 PCTE Mapping

The application was not implemented on PCTE. However, we carried out a paper exercise of designing a PCTE schema for the application.

In this application, the `WaterSource`, `Pipe` and `Valve` components are processes. Hence, they would be represented in the PCTE object management system (OMS) by process objects. Each could have its own type, and there could be many instances of each type in the OMS. The processes need to transfer “data” so the connections between these objects would be represented by message queues. At this point, there is a decision to be made which was not necessary in the JavaBeans implementation. That is, to decide which connections are allowed. For example, can a `Valve` component be connected directly to another `Valve` component? The answers to these questions determines which types of message queue are required. It is also necessary to determine whether the various message queues may have multiple readers or writers. This would constrain whether, for example, the same `Pipe` component could be connected to a `Valve` component at both ends.

To complete the implementation, the `WaterSource`, `Pipe` and `Valve` processes would have to be coded in an appropriate implementation language, and then instances of the processes would be created in the PCTE OMS and connected together with the appropriate message queues to form the required “plumbing” arrangement. This would be done using the PCTE interface primitives.

If one wanted the same flexibility of connectivity that the initial JavaBeans implementation allowed then one could define all of the process objects to be of the same type and allow the message queues to connect any of these objects, with multiple readers and multiple writers.

4 Comparison

Obviously, the implementation of the components is required in both integration strategies. There is no reason why the implementation effort of the components should be different. The difference comes in achieving the integration. With JavaBeans, the integration is almost for free. Once the components have been instantiated as Beans, the “plumbing” model can be built using the BeanBox very easily. With PCTE, on the other hand, before any integration is possible the schema for the application must be defined. This represents a considerable overhead. However, the schema does give one the opportunity to constrain the “plumbing” models that can be built and so enforce some constraints on the design. Once the schema is correctly defined, PCTE will control how the components may be integrated.

Trying to constrain the integration in the JavaBeans approach requires extra work, and some types of constraint seem to be difficult to impose. However, unconstraining the integration in PCTE is easy. The two technologies provide very different approaches to integration.

The question is, whether integration of components should be controlled. Ousterhout, in his paper [8], argues that component integration “glue” should be accommodating enough to allow flexible integration. Previous experience has shown that placing constraints on components may inhibit integration. For example, the strong typing in the Ada programming language is seen as a barrier to successful reuse.

5 Conclusions

JavaBeans shows great promise as a component technology. Built in introspection and aggressive generation of adaptors by the BeanBox combine to greatly simplify component integration. This ease of integration is enabled by the JavaBeans but provided by the Interactive Development Tool (IDT), in this case the BeanBox. The IDT provides the role of a modular interconnection language, or an aspect language [9] where the aspect is component integration.

As JavaBeans provides a well defined interface model it should be possible to develop other IDT or aspect languages that support different integration rules or provide different integration mechanisms, perhaps using Java RMI to allow individual components to be distributed across a heterogeneous network.

JavaBeans technology continues to improve with innovations such as the JavaBeans Activation Framework and the BeanContext. Commercial success of JavaBeans is likely to depend on other factors, such as market requirements to develop portable software that runs on heterogeneous platforms, performance of the Java Virtual Machine (JVM) and Microsoft’s ability to present a coherent, alternative architecture while under increasing scrutiny from the U.S Justice Department.

References

[1] Brown, A. and Wallnau, K., “Engineering of Component-Based Systems” in Component-Based Software Engineering, IEEE Computer Society Press, Los Alamitos, CA., 1996.

[2] Bass, L., Hardy, E., Hoyt, K., Little, R., Seacord, R., Introduction to the Serpent User Interface Management System, CMU/SEI-88-TR-5, 1988.

[3] “JavaBeans”, Sun Microsystems, Graham Hamilton (Editor), July 24, 1997 Version 1.01.

[4] “The Java Core Reflection PI and Specification”, Sun Microsystems, February 4, 1997.

[5] “JDK 1.1 - AWT Enhancements”, Sun Microsystems.

[6] Using the Beans Development Kit 1.0, April 1997 A Tutorial, Alden DeSoto.

[7] Wakeman, L. and Jowett, J., “PCTE: The Standard for Open Repositories”, Prentice-Hall, 1993.

[8] Ousterhout, J.K., “Scripting: Higher-Level Programming for the 21st Century”, http://www.sunlabs.com/people/john.ousterhout/scripting.html, May 10, 1997.

[9] Aspect-Oriented Programming, Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier and John Irwin, PARC Technical Report, February 97, SPL97-008 P9710042.