Comparing JavaBeans and OSGi Towards an Integration of Two Complementary Component Models.

Humberto Cervantes, Jean-Marie Favre
Laboratoire LSR Imag, 220 rue de la Chimie
Domaine Universitaire, BP 53, 38041 Grenoble, Cedex 9 France
Humberto.Cervantes@imag.fr, Jean-Marie.Favre@imag.fr

Abstract

In today's software engineering practices, building applications from components is the ongoing trend. What can be noticed however is that there really is not a clear consensus about the definition of components, and instead in the literature we find many definitions of what components are. This renders a comparison between component models difficult. However, it is possible to compare different component models over a list of characteristics that several authors agree that they should be found in a component model. In this article, Sun's JavaBeans and the Open Services Gateway Initiative's framework (OSGi) are compared. These are two technologies that target very different types of applications. Their study reveals, however, that both technologies cover, at different levels, a set of important features that characterize components. It also reveals that these component models are in some ways complementary. The paper concludes by giving a proposal to integrate these two technologies to obtain a more complete component model.

1. Introduction

Component Based Software Engineering (CBSE) is one recent trend in the domain of Software Engineering (SE). One major reason why this paradigm has emerged is the need to build software by assembling reusable units, or *components*, as opposed to building whole applications from scratch.

Of the many component technologies that exist today, we are more particularly interested in two Java-based ones: Sun's JavaBeans [4] targeted towards the visual assembly of non-distributed applications and the Open Services Gateway Initiative (OSGi) [5] which is targeted towards the deployment of services in platforms such as home gateways.

These two technologies, which target very different kinds of applications, might initially seem to have nothing in common, however, a closer look reveals that some of their characteristics are complementary. The goal of this paper is 1) to do a comparison of both component technologies and 2) to study a possible way of integrating them.

Comparing component technologies can be a diffIcult task mainly because there is not a clear definition of what exactly components are. In order to establish a framework for comparison, we have collected from a series of articles [1,2,3], a list of relevant features of components. According to these sources, components:

- Have clear and explicit boundary (Well specified interface and explicit dependencies).
- Can be customized
- Can be assembled
- Are reusable
- Are units of substitution
- Are units of delivery and deployment
- Have certified properties

These features characterize a *component model* that is, according to [2], "the set of component types, their interfaces, and, additionally, a specification of the allowable patterns of interaction among component types." Finally, it is important to also take into account the *component framework* which "provides a variety of runtime services to support and enforce the component model."

The rest of this article is structured as follows: section 2 describes the JavaBeans component model, section 3 describes OSGi. Section 4 makes a summary and compares both component models, finally, section 5 concludes by describing a possible way to integrate both models.

2. JavaBeans

2.1 Overview

The JavaBeans component model specification [4] appeared in 1996 and introduced several concepts that were designed to ease the task of visually assembling applications out of components called *JavaBeans*.

JavaBeans components are standard Java classes that either follow certain coding conventions to express the main features of the component or that are accompanied by a class (called *BeanInfo)* that provides this information. Within a JavaBean we can find methods, properties, event sources, and event sinks. Normally these classes are serializable, the reason for this being that any serialized instance can become a *prototype* [6] for other instances. A prototype is the equivalent of a class in the sense that it allows instances to be created from it. Prototypes are also considered to be JavaBean components.

The JavaBeans specification distinguishes two different moments in the life cycle of a JavaBean instance: "design-time" and "run-time." Design-time, which normally takes place within a builder tool, occurs when JavaBeans instances are configured and inter-connected, forming an *assembly* which can then be stored. During run-time, the assembly is executed as a standard program.

Packaging is considered in the specification, however, what is described is how to package the JavaBeans components, not their assemblies. JavaBeans are packaged as JAR files, a special file format that includes among other things a *manifest* which is a special file where information about contained JavaBeans is stored.

2.2 Component Features

We will now describe in more detail how the features of components that were listed in the introduction are expressed in the JavaBeans component model.

Clear and Explicit boundary:

The boundary of a JavaBean is clearly defined since the naming patterns used to write its interface give a good definition of the properties, methods, event sources and event sinks that the JavaBean implements.

The dependencies of a JavaBean are, however, not described thoroughly. These dependencies can be divided in two different categories: the first one concerns the dependencies towards the Java runtime and imported packages, which are not described at all, and the second category includes the dependencies of a JavaBean towards other JavaBeans, classes or resources. The latter dependencies can be optionally described in the manifest of the JAR file but only if they take place within the same JAR file that contains the JavaBean (Fig.1).

More specifically, there are two ways in which a JavaBean can depend on another JavaBean:

a) A JavaBean contains an instance of another JavaBean, effectively creating a containment relationship. This occurs when there is a Beans.instantiate() statement inside of the code of the JavaBean.
b) A JavaBean contains a reference to another JavaBean, thus creating an association relationship. This reference is set when the assembly of JavaBeans is created.

```
Name : somepackage/CustomizedBean.ser
Java-Bean : True
Depends-On : somepackage/myIcon.gif
Depends-On : somepackage/CustomizedBean.class
Depends-On : anotherpackage/AnotherBean.class
```

Figure 1.

Customizability:

Customizing a JavaBean means changing the values of its properties. Properties in the JavaBeans component model are values that can be read and changed through *getter* and *setter* methods (this makes reference to the naming pattern that these methods follow which is that to get a property the method should be called get<PropertyName> and to set the value of a property, the method should be called set<PropertyName>).

Customization in JavaBeans can be done either at the component level, if the properties of a prototype are changed, or at the instance level, if within an assembly the properties of a particular instance are set. Customization is an activity that normally takes place during design-time.

To be able to customize a JavaBean, information about its characteristics must be obtained. This information can be obtained in two different ways. The first one, called *introspection*, automatically discovers features of a JavaBean from the getter and setter methods it exposes. This information can then be used for example to build a configuration panel that allows properties to be set in an interactive way. The second mechanism consists in using the information provided by a companion class to the JavaBean called a *BeanInfo*. These companion classes can also provide methods that allow properties to be set in more particular ways, eventually by displaying complete graphical editors for the properties. Companion classes to JavaBeans are identified by naming conventions (the name of the JavaBean plus the BeanInfo suffix).

Can be assembled:

As we have described before, JavaBeans are designed to be instantiated and then assembled, so this

characteristic is fundamental in this component model. Assembly is meant to be realized by a third party (typically a builder tool) that will call the methods of the JavaBeans by passing them either values of properties or references to other JavaBeans. We recall however that the assembly is not meant to become a JavaBean itself, it is the responsibility of the third party to store it so that it can be rebuilt later. Currently the *Long Term Persistence Schema* for JavaBeans allows assemblies of Java Beans to be stored in a standard XML format [7].

It must be noted that at the implementation level, assembly and customization in the JavaBeans component model might be confused. The reason for this is that both activities are done by calling getter and setter methods on the JavaBeans.

Reusability:

JavaBeans are meant to be reusable components, since their instances can be used to build many different kinds of assemblies. The limitation, however, is that assemblies cannot become themselves JavaBeans to allow for further reuse.

Units of substitution:

Substitution makes reference to the possibility of replacement of one component by another. We have previously described that relationships between JavaBeans can be of two different kinds: containment and association. Substituting a component that is contained is different than substituting one that is associated.

When a JavaBean contains another one, normally their relationship is buried inside of its source code. In Figure 2 there is an example of a contained bean. Inside of the constructor of the MyBean class, an instance of a ContainedBean component is created. Substitution in this case can only take place if there are changes at the physical level, for example a change in the classpath or in the JAR that exports the JavaBeans, the consequence of this will be that when MyBean is instantiated, the substituted ContainedBean will be loaded. It is important to notice that this type of substitution implies that the bean that substitutes must have exactly the same name that the one it replaces, meaning essentially that it is a different version.

The second possibility of substitution can occur between two JavaBeans that are associated. In figure 2 we can see this situation occuring for the AssociatedBean. This association normally will be done by a third party that will give a reference of an AssociatedBean to MyBean. Substitution can take place if the third party gives a reference to any subclass of AssociatedBean. Another difference with respect to containment is that this kind of substitutions can take place at any time during execution, while the first one only takes place during instantiation.

```
class MyBean implements Serializable
{
  private ContainedBean cb;
  transient private AssociatedBean ab;

  MyBean()
  {
        cb=(ContainedBean)Beans.instantiate(
                              "mybeans.ContainedBean");
  }

  public void setAssociatedBean(AssociatedBean newbean)
  {
        ab=newbean;
  }

  public AssociatedBean getAssociatedBean()
  {
        return ab;
  }
}
```

Figure 2.

Units of delivery and deployment:

In the JavaBeans component model, the unit of delivery is the JAR file, not the JavaBean itself. Since a JAR file can contain one or more JavaBeans, what is distributed is a collection of components.

Deployment is not specifically treated in the JavaBeans specification, however, to deploy an application built out of JavaBean components, the collection of JAR files that contain the JavaBeans must be installed in an accessible location so that the JavaBeans can be loaded when the assemblies are recreated.

Certified Properties.

A component with certified properties will have a predictable behavior. In the JavaBean component model, however, there is no way to specify the behavior of a particular component.

2.3 Component Framework

In the JavaBeans component model, there is a minimal component framework provided by a class called java.beans.Beans. The basic services provided by this class include instantiation and resource loading. An extension to the original specification [10] further introduced the concept of *BeanContexts* as a way to provide services to the JavaBean instances. BeanContexts

allow JavaBeans instances to be organized in a hierarchical structure and act as service providers for those instances.

2.4 Summary

We have described the JavaBeans component model as a model that distinguishes two different moments in the lifecylce of components: design-time and run-time. Since this component model is targeted towards the visual assembly of applications, it makes a strong emphasis on configuration aspects. Packaging is treated altough only in a limited way.

The JavaBeans component model covers te set of features that characterize components with the exception of certified properties. It is important to notice that not all of the characteristics are covered by the same entity, some belong to the JavaBean while others belong to the JAR files that export them.

3. OSGi

3.1 Overview

The Open Services Gateway Initiative created the specification of the OSGi services platform to ease the deployment and management of services in a coordinated way [5].

OSGi defines a non-distributed *framework* where units of deployment called *bundles* are installed and managed. A bundle is installed from a JAR file and is identified by a unique number and the location of the file from where it is installed. It must be noted that in the framework there cannot be more than one bundle installed from the same location. Management of the bundles includes starting, stopping, updating and removing them. Every bundle inside the framework has a state associated with it which can take the following values: INSTALLED, RESOLVED, STARTING, ACTIVE, STOPPING or UNINSTALLED. The state of the bundles that are installed in the framework is persistent, meaning that it is restored after the framework is shutdown and restarted.

A JAR file exports a single bundle and contains a *manifest* file where information about the bundle is stored. This information includes the location of a class called the *activator*. This class plays a very important role, since it is called when the installed bundle is started or stopped and also receives a reference to the framework that allows the bundle to interact with it. The manifest also contains other information such as package dependencies, used and provided services and general information about the bundle.

Bundles contain *services* which are, according to the specification, the components from which applications are built. The services that a bundle contains can be registered or unregistered in a framework *registry*. Every service is registered with its name (a Java interface) and a set of properties of type <value,pair>. A bundle can send a request to the framework to obtain a service by providing a filter in an LDAP syntax, this query might eventually return a set of candidates.

A very important aspect of the paradigm of OSGi is that services may appear or disappear at any time during the execution of an application, as a consequence of the management of the bundles. This means that an application, as a set of bundles connected through services, is in constant evolution. This also means that there is not a static assembly described anywhere, services are connected or disconnected dynamically, and the code inside of the Bundles must be prepared to handle this situation.

3.2 Component Features

We will now study the way the features of component models are expressed in OSGi.

Clear boundary and Dependencies:

The boundary of a bundle is clearly defined since a bundle is completely contained within its JAR file. The dependencies of the bundle are divided into three different types: package dependencies, service dependencies and runtime dependencies.

A bundle can import or export source code in the form of a package that may have a version number. This mechanism allows the bundles to access or to give access to the code of the service interfaces, so that it is not necessary to include this code in every bundle. It is also used to load or share library code. These dependencies are clearly specified in the manifest of the bundle since they must be resolved in order to allow the bundle to be started, hence the difference between the INSTALLED and the RESOLVED state.

Bundle to service dependencies are not handled by the framework, they can be declared in the manifest file but only for informative purposes. There is also no way to express service to service dependencies.

Concerning runtime dependencies, they can be described in the manifest for information purposes, moreover, the OSGi framework allows a bundle to query

it for the version of the Java runtime environment where it is being executed so that eventually it can act depending on this information.

Some dependencies cannot be expressed in OSGi, in particular, those concerning the standard Java classpath or the /lib/ext directory. These ways of locating code should be avoided.

Customizability:

Services can be customized by their clients if they implement a particular interface. However the changes made upon them have a repercussion on all of the clients of the service, since services are shared. The state of the service can be rendered persistent into the installed Bundle.

There is no standard mechanism to configure bundles other than through services.

Can be assembled:

As we described previously, within the OSGi framework, the applications are in constant evolution. These applications are built from bundles connected through services, but the structure of the application is not a static one since services come and go at different moments during execution.

Through the package dependencies, bundles are connected between themselves in a static way. However, it is not possible to specify that a bundle imports the packages from some particular bundle, what can be expressed is only that a bundle imports packages and the framework is charged to decide the bundle from where they will be imported.

Reusability:

Services by definition are meant to be reusable, since they can be used by many different clients. Any bundle can become client of a service if the service is available at the moment it requests it.

Bundles are reusable units too, since they can be installed over several frameworks.

Units of substitution:

In OSGi, substitution is an important characteristic which takes place during the execution of the application. Substitution can take place either in services or in bundles. A service can be substituted by another one if the latter implements the same interface and if the properties that the client requests for the service are the same in both services. This is the reason why when the framework is queried for a particular service, it can return a collection of services instead of a unique answer. If two services implement the same interface and the properties requested in the filter are found on both services, they can be considered as being equivalent, and it is up to the requesting client to choose the one it will use.

Bundles are also units of substitution, since a bundle can be changed by another one that exports equivalent services and packages. Bundles can be updated, and this means that they are replaced by a different version of themselves.

Units of delivery and deployment:

The JAR files that contain the bundles are the units of delivery in OSGi. The framework allows these files to be installed from a remote location. These files are considered to be units of deployment too, since there is not a standard way to distribute and deploy several bundles at once as a high level unit, like an assembly of bundles.

Certified Properties.

In OSGi, services are characterized by a set of properties but apart from this there is no way to certify that a service will comply with a particular behaviour.

3.3 Component Framework

The OSGi framework provides methods that allow the bundles to request and to register services. It also allows bundles to register themselves as listeners to different types of events : service events (registering / unregistering), bundle events (change in the state of a bundle) and framework events (start/stop). The framework also provides bundles with methods that allow them to manage the framework, altough this may be restricted by setting permissions. Finally, the framework allows the bundles to save their state by providing them access to a file.

3.4 Summary

We have described OSGi which can be considered as a component model since the features of components are covered either by the bundles or by the services. It must be noted that OSGi applications are not described statically and they evolve along with the services that are registered or unregistered in the framework.

Packaging, deployment and delivery are essential aspects of OSGi, and the framework provides powerful mechanisms to manage the bundles that are installed on it.

4. Comparing OSGi and JavaBeans

4.1 Differences

The first important difference between OSGi and JavaBeans, is that OSGi explicitly defines a framework that manages both the units of distribution that are the bundles and the services they export. In JavaBeans, it is mostly the Java runtime and the Beans class that are the framework for the components, so JARs are installed following the conventions for standard Java classes. This, however, has limitations since versioning is treated poorly in Java. Another clear advantage of OSGi is that the framework provides powerful mechanisms to allow for management of the Bundles, for example to update or install from a remote location, something that is not available for the JavaBeans.

Among the differences, we notice that in OSGi assemblies are not static, connections between bundles occur depending on the availability of services.

The concept of service as an object that can be obtained by querying the framework through a set of properties has no exact equivalent in the JavaBeans component model, although something that approaches it was introduced in the extension to the original specification [10] where the concept of BeanContexts was introduced. BeanContexts allow JavaBeans instances that are registered within a BundleContext to ask it for services. In this case the context acts in a similar way to the OSGi framework, however, requests and registration are done in a simpler way. From this point of view JavaBeans instances could be compared to Bundles, although Bundles exist as singletons in the OSGi framework, and the latter is the only context where all the Bundles reside.

4.2 Similarities

We will now focus on the similarities of these two component models. One common aspect is that in both component models there are two clearly different entities that possess part of the component characteristics for the model: In JavaBeans these two entities are JavaBeans and JAR files, and in OSGi they are the services and the bundles. We must point out that the existence of this kind of separation is not specific to the component models that we have studied in this paper, we can find a similar entities in .NET [8] (*Assemblies* and *Components)*, in EJB [9] (JARs and EJB) and in other models. It is interesting to notice that in each of the component models, one of the two entities, that is JARs and bundles, is particularly oriented to be a unit for deployment.

4.3 Summary

In the previous section we have discussed and compared JavaBeans and OSGi. We have concluded that both can be considered as component models, since they support what we considered as common features of components.

We must note though that even if in both component models we can find similar aspects, some of them may be better treated in one component model than in the other, the prime example for this is the units of distribution in OSGi that are treated in a much complete way than the JARs loaded through the standard classloader mechanism of Java. In a similar way, assemblies are treated in a more complete way in JavaBeans.

5. Conclusion

In this paper we have studied JavaBeans and OSGi to find if they supported a set of characteristics that are found on component models. We concluded that in both component models we find all of the characteristics that we described on section 1 with the exception of certified properties which are only treated very superficially in OSGi. Among the similarities of both component models, we found that two main entities existed in each of them: one that is more focused to the distribution and deployment and another one that is more oriented on the assembly of components. One of our conclusions is that if we consider the set of characteristics described in section 1, it is not possible to consider each individual entity as being components, but that rather that the features of components are distributed over the two kinds of entities. This can be summarized by saying that JavaBeans alone cannot be considered completely as components if we do not speak of the JARs that are the units in which they are distributed.

A second conclusion is that although in both models we find units of distribution, in the case of OSGi, these units are defined and managed in a more complete way than in JavaBeans. On the other side, some aspects of JavaBeans are not found on OSGi. We think that it is possible to integrate both component models so that the best features of each one are used to build a common component model. To do so we can:

- Use bundles as distribution units that contain JavaBeans.
- Make these bundles export some particular service, such as a BeanFactoryService that allows for the creation of instances of the JavaBeans they export. The OSGi registry would

then be used to lookup for factories of components.

- Give JavaBeans access to the OSGi framework, so that they can query for services in a more complete way than what is available today through the BundleContext.

The benefits of an eventual integration of these two technologies are various. The most immediate advantages are that by doing so we would benefit from the OSGi framework mechanisms that manage the distribution units in a local or remote way. The other advantage is that it would be possible to build applications out of static assemblies of components, something that is currently not specified in OSGi.

We are currently testing this approach and our current results are encouraging. Due to a lack of space we have limited ourselves to just citing the main ideas that we are putting into place to reach the integration of these two component models. More information can be found at the following web address:

http://www-adele.imag.fr/BEANOME

Finally we would like to thank Dr. Richard S. Hall for his reviews and support.

Bibliography

[1] Bass, Buhman, Comella-Dorda, Long, Robert, Seacord, Wallnau. “Volume I: Market Assessment of Component Based Software Engineering.” Technical Report CMU/SEI-2001-TN-007, May 2000

[2] Bachman, Bass, Buhman, Comella-Dorda, Long, Robert, Seacord and Wallnau “Volume II: Technical Concepts of Component Based Software Enginnering”, Technical Report CMU/SEI-2000-TR-008, May 2000

[3] C.Szyperski, Cuno Pfister : Why Objects Are Not Enough, Proceedings, First International Component Users Conference, july 1996

[4] Sun Microsystems : “Java Beans Specification”, version 1.0.1, 1997 http://java.sun.com/beans

[5] OSGi service gateway specification, version 1.0, May 2000, http://www.osgi.org

[6] E. Gamma et al., “Design Patterns - Elements of Reusable Object-Oriented Software”, Addison Wesley, 1995

[7] Sun Microsystems, “Long Term Persistence of Java Beans Components”
http://java.sun.com/products/jfc/tsc/articles/persistence3/index.html

[8] Microsoft, “The .NET Framework”
http://msdn.microsoft.com

[9] Nicholas Kassem and the Enterprise Team, “Designing Enterprise Applications with the Java 2 Platform, Enterprise Edition.”, Version 1.0.1, October 2000

[10] *Sun Microsystems*, “Extensible Runtime Containment and Server Protocol for JavaBeans” Version 1.0 December 3, 1998