Beanome: A Component Model for the OSGi Framework

Humberto Cervantes and Richard S. Hall
Laboratoire LSR Imag, 220 rue de la Chimie
Domaine Universitaire, BP 53, 38041 Grenoble, Cedex 9 France

{humberto.cervantes,richard.hall}@imag.fr

Abstract:
The Open Services Gateway Initiative (OSGi) specification defines a service-oriented framework for use in residential gateways. In this context, the OSGi framework acts as a gateway from the Internet to consumer devices attached to the residence's home-area network. Service providers use the gateway to deliver products and services to end-users, such as home security or health care monitoring. As more powerful consumer devices are introduced, the devices themselves become mechanisms for delivering services to the framework. This level of sophistication results in complex applications and the need for sophisticated mechanisms to simplify creating them. This paper introduces Beanome, a lightweight layer implemented on top of the OSGi framework, that implements a simple component model and framework for creating OSGi applications. Beanome is not a component model for consumer devices per se, but is a component model for the OSGi framework, which itself is used to create applications that access and provide user-level and management-level interfaces to consumer devices.

Keywords: OSGi, lightweight, component model, component framework

1 Introduction

For software developers, consumer device technology provides an increasingly interesting environment in which they can deliver products and services. Consumer devices are much more prolific than personal computers and, perhaps more importantly, are more integrated into everyday life. As a result, software developers benefit from unprecedented levels of access to the user by targeting their products and services towards consumer devices. The consumer device software market is fueled by four factors:

- The increasing power of consumer devices, such as mobile phones or personal digital assistants (PDAs).
- The transformation of “traditional”consumer devices, such as refrigerators and washing machines, into “smart”consumer devices.
- The widespread availability of broadband Internet connectivity.
- The growing prevalence of home-area networks.

The Open Services Gateway Initiative (OSGi) is helping developers realize the potential of the consumer device market; OSGi is an “independent, non-profit corporation working to define and promote open specifications for the delivery of managed broadband services to networks in homes, cars, and other environments.” [4] Specifically, the OSGi specification defines a service platform [5] that includes a minimal component-like model and a small framework for managing the components, including a packaging and delivery format.

The OSGi framework facilitates dynamic installation and management of simple components called *bundles*. These dynamically loadable components interact with each other and form “applications” by providing and using services. A service is a Java interface with defined semantics and potentially multiple implementations. Services are packaged along with their associated resources and deployed via the internet into the OSGi home services gateway. In

this scenario, consumer devices become delivery mechanisms, capable to deliver bundles along with their associated services.

The services gateway acts as the access mechanism or conduit between service providers and the end-user. A typical application scenario is that of a home security company that monitors the end-user's home for security concerns using a combination of hardware (e.g., sensors) and software (e.g., sensor monitors, software control panels for the end-users and the service provider). While OSGi creates a good foundation on which to build such services, it is still fairly low-level; the low abstraction level increases the complexity of offering complex applications on top of the OSGi framework.

This paper investigates the limitations of the OSGi Service Platform specification with respect to using it to build complex applications. Beanome, a lightweight component model and framework implemented on top of the OSGi platform, is introduced as a means to improve support for complex OSGi applications. The paper begins with an overview of the OSGi Service Platform and is followed by a discussion of the limitations of OSGi. Then the Beanome component model, framework, and mapping to OSGi are presented. Current status and future work follow, along with related work and the conclusion.

2 The OSGi Service Platform

The OSGi Service Platform is comprised of two parts, the OSGi framework and the OSGi standard services. For the purposes of this paper, the framework is the most important since it defines what a service is, whereas the standard services define specific services and their specified functionality. As such, the standard services, which include HTTP, logging, device access, and user management among others, are not discussed further. It is sufficient for the purposes of this paper to understand that services in OSGi are simply Java interface definitions that have precise and specified semantics; the semantics of a service interface are defined by the creator of the service interface and any object that implements a service interface is assumed to obey its contract.

The OSGi framework creates a host environment for managing bundles and the services they provide; a bundle is the physical unit of deployment in OSGi and is also a logical concept used by the framework for organizing its internal state. Concretely, a bundle is a Java JAR file that contains a manifest and some combination of Java class files, native code, and any associated resources. An installed bundle in the framework is uniquely identifiable by either its bundle identifier, a number assigned dynamically by the framework when the bundle is installed, or by its location, which is an arbitrary character string used when installing the bundle. In common practice the location string of a bundle is typically a URL, but the specification does not define it as such; since bundles are uniquely identified by their location string, it is not possible to install two bundles from the same location.

The management mechanisms provided by the framework enable bundle installation, activation, deactivation, update, and removal. Logically, installed bundles have an associated state that at any given time is one of the following values: *installed*, *resolved*, *starting*, *active*, *stopping,* or *uninstalled*. The state of active bundles is persistent across framework activations, meaning that active bundles are returned to the active state when the framework is shutdown and subsequently restarted.

The bundle-to-JAR file mapping is one-to-one and meta-data about the bundle is available in the manifest file of the JAR file; the manifest file corresponds to the entry in the JAR archive named `META-INF/MANIFEST.MF`. The manifest file is simply a set of attribute-value pairs where some attributes are standardized by the OSGi specification. The standard manifest

attributes describe the class path for the bundle, any exported Java packages provided by the bundle, any imported Java packages required by the bundle, native library requirements, and information intended for humans (e.g., bundle name and description).

Besides the aforementioned attributes, the manifest may also contain an attribute that specifies an *activator class* for the bundle. The activator class plays an important role since it is the mechanism by which bundles are able to get a *context* for accessing framework functionality. The activator class is called when the bundle is started or stopped and, in both instances, the bundle is given its specific context for accessing the framework. The context allows the activator or any other class that has access to the context to look up services in the framework's service registry, register services of its own, access other bundles, and install additional bundles. Bundles are not required to have an activator class; it is possible that a bundle is only a library of Java packages and does not need to access the framework.

Services are the main building blocks for creating applications in the OSGi framework. Services are specified as a Java interface and are implemented by a separate Java class and its associated classes and resources, all of which are packaged as a bundle (i.e., a JAR file with a manifest). The service interface definitions are exported by the defining bundle so that other bundles that want to use the service can import them; the service implementations remain private.

A bundle implementation may request services from other bundles, register services to be used by other bundles, or both. A service is registered with the framework by specifying the service's interface name, an instance of a class that implements the service interface, and an optional set of attribute-value pair properties. To find a service, a bundle performs a simple LDAP query on the framework's service registry; it is sufficient to query using only the service interface name, but a query can also include references to the associated service properties. A service query returns zero or more matching services.

3 OSGi Limitations

The scope of the OSGi specification is focused on defining a services gateway; the OSGi specification does not purport to be a sophisticated component model for developing complex applications. As such, the limitations discussed in this section are not shortcomings of OSGi per se, but result from the fact that developers will invariably try to build complex applications with it.

3.1 Dependencies

In OSGi, three types of dependencies are present:

- Bundle-to-package: classes inside of a bundle may need to import code that is external to their JAR file; in this case, the bundle must explicitly import the code (in the form of Java packages). Other bundles can explicitly export Java packages, which are then used to satisfy import requirements. If the framework is unable to fulfill a bundle's import requirements, then the importing bundle cannot be started. Package dependencies are declared in the manifest file.
- Bundle-to-service: classes inside of a bundle may use registered services by querying for them in the framework service registry. Unlike package dependencies, service dependencies are not guaranteed by the framework; this means that a bundle can be started, even if the services that it requires are not available. Service dependencies may be declared in the manifest file, but currently the OSGi specification indicates that this is for information purposes only.

- Service-to-service: this is a special case of a bundle-to-service dependency, where the dependency is between a specific service implementation and an external service, rather than between the bundle as a whole and an external service. In this particular case, a bundle may have several service implementations, some of which have their dependencies satisfied and some of which that do not. Service-to-service dependencies are not covered by the OSGi specification.

Bundle-to-package dependencies are handled fairly well within the OSGi framework, which even provides simple version control support (where backwards compatibility is assumed). Neither bundle-to-service nor service-to-service dependencies are handled by the OSGi framework, even though bundle-to-service dependencies are described in the bundle manifest.

Since bundle-to-service and service-to-service dependencies are implicit in OSGi, they are not externally manageable. Instead, the details of these dependencies are buried inside of their respective bundle and service implementations, which results in OSGi applications that are difficult to deploy and manage. For example, it is not possible to create a service to automatically install all required services for a bundle nor is it possible to allow the user to externally configure service dependencies if he or she wants to use a specific service implementation.

Bundle-to-package and bundle-to-service dependencies are fairly easy to understand, but service-to-service dependencies are more complicated. A service-to-service dependency is not a dependency between two services, rather it is a dependency between a specific service implementation and another service interface. In other words, services themselves do not have dependencies because they are only interfaces and do not specify an implementation. Only implementations of a service can have dependencies and multiple implementations of the same service may have different dependencies.

There are multiple, independent factors that characterize a service-to-service dependency; two important factors are cardinality and dynamism. Cardinality is useful for expressing optionality, such as a zero-to-one dependency, and aggregation, such as a one-to-many dependency. Dynamism indicates whether a dependency tracks changes in the environment at runtime, such as the arrival of new services that satisfy the dependency constraints. An example of a dynamic, zero-to-many dependency occurs in a situation where an application offers a "plugin" mechanism. In this scenario, the application can function without any plugins and it can integrate new plugins as they arrive in the environment.

Notice that bundle-to-bundle dependencies are not present in OSGi, in fact, they are essentially forbidden. This means that it is not possible to depend on the packages or services of a specific bundle and the framework is free to resolve these issues as it sees fit.

3.2 Application-level

Conceptually, an "application" in the OSGi framework is the transitive closure of all dependencies among packages, bundles, and services. As the previous subsection points out, OSGi handles package dependencies, but does not handle dependencies among bundles and services or among services implementations. As a result, OSGi not have an explicit notion of an application nor does it define any application-level services, such as automatic application deployment.

3.3 Complexity

An important aspect of the service-oriented paradigm of OSGi is that services may appear or

disappear at any time. This means that an application, i.e., a set of bundles connected via service dependencies, must be able to handle this dynamic situation. As a result, all bundles must implement complex code to handle binding and unbinding of services as they dynamically arrive and depart, respectively.

4 Beanome

Beanome adds a thin layer on top of the standard OSGi framework to provide functionality similar to that found in existing component models [9] so that applications are built out of assemblies of instances of components [see Figure 1]. Beanome defines a component model and a component framework to support the model. The following subsections describe the characteristics of Beanome components, the framework, and their mapping onto the OSGi framework.

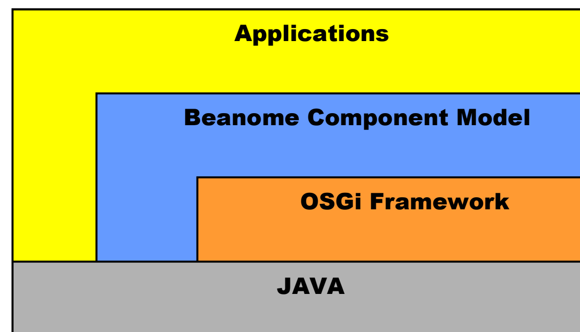

Figure 1. Layers introduced by Beanome.

4.1 Beanome Component Model

The Beanome component model is based on the concept of component types. Component types are identified by a unique name and have an associated version number. The structure of a component type is defined in an XML file, called a component descriptor file, that extends the OSGi manifest. Several components may be described in a single component descriptor file. A component type, as described by its component descriptor, is used as a template for creating component instances.

The simplest possible Beanome component type consists of an interface and a Java class that implements the interface. In this context, "interface" is abstract and refers not only to a Java interface, but may also refer to a Java class. A component type may provide more than one interface, where each interface is implemented by a different class. When multiple implementations exist, it is possible to bind the separate implementations. Figure 2 is an example of a component descriptor for a simple component that provides two interfaces where each is implemented by a separate Java class and there is a binding between the two implementations.

```
<component name="mycomponents.SimpleComponent" version="1.0.0" runtime="1.3.0" icon="res/icon.gif">
  <provides name="org. beanome. commoninterfaces.Application" implementation="appimpl"/>
  <provides name="javax.swing.JComponent" implementation="uiimpl"/>
  <implementation name="appimpl" definition="fr.imag.bundle.example.ApplicationImpl" type="class"/>
  <implementation name="uiimpl" definition="fr.imag.bundle.example.UserInterfaceImpl" type="class"/>
  <bind source="appimpl" target="uiimpl" definition="setAppReference" type="method"/>
</component>
```

Figure 2: Simple component type that provides two interfaces.

Components types may need to create instances of other component types within their implementation; in this case, the component must define a requires clause for the required component. An interface provided by the required component must also be specified. Any required component dependency must be resolved before the component type can be instantiated. These types of dependencies result in a hierarchy where one component type is built up out of instances of other component types and so on. The top-level of such a hierarchy results in a Beanome application that is itself a component type that may be re-used in another application.

Component types may also have properties associated with them that are used for customization purposes. The types of these associated properties are primitive, such as integer, boolean, and string. These properties are per component type, not per component instance; similar to static members in a Java class.

Sometimes it is inconvenient to define all component interface implementations and/or all interface implementation bindings directly in the component descriptor file; for example, selected interface implementations may depend on property values and the subsequent bindings may depend on the chosen interface implementations. While it is possible to extend the component descriptor format to describe such a complicated scenario, the end result would be a cumbersome and inefficient mini-programming language. Instead, the component descriptor format allows interface implementation selection and implementation binding to be delegated to a Java class. Figure 3 describes a complex component that has provided interfaces, required interfaces, properties, and delegated implementation selection and binding.

```
<component name="mycomponents.ComplexComponent" version="1.0.0" runtime="1.3.0" icon="res/icon.gif">
  <provides name="org. beanome. commoninterfaces.Application" implementation="appimpl"/>
  <provides name="javax.swing.JComponent" implementation="uiimpl"/>
  <requires name="org.beanome.viewers.interfaces.GraphViewer" filter="(&(name=mycomponents.GraphViewerComponent)(version=1.0.0))"/>
  <property name="testmode" type="bool" value="true"/>
  <implementation name="appimpl" definition="fr.imag.bundle.example.ComplexComponentController" type="delegate"/>
  <implementation name="uiimmpl" definition="fr.imag.bundle.example.ComplexComponentController" type="delegate"/>
  <bind source="appimpl" target="uiimpl" definition="fr.imag.bundle.example.ComplexComponentBindingManager" type="delegate"/>
</component>
```

Figure 3. A Complex component which delegates implementation selection and binding.

4.2 Beanome Component Framework

The Beanome component framework provides the necessary runtime functionality to support the Beanome component model. The component framework, called the runtime, introduces two main concepts: component factories and a component registry. The component registry maintains a list of all available component factories that are registered with the runtime. A component factory is associated with a specific component type and implements the *factory*

method pattern [2] for creating component instances. Component factories also provide *finder* methods to lookup shared component instances, where each component instance is identified by a unique, "well-known" name. The runtime provides access to the component registry and to introspection mechanisms on component types and instances.

To create a component instance, the runtime retrieves the component factory for the desired component type from the component registry. The runtime then creates all of the objects that implement the component and that are specified in the component descriptor. For instantiation to be successful, the runtime must also locate factories and create instances for component types that are required (i.e., there is an associated "requires" clause in the component descriptor). If the runtime is unable to find factories for required interfaces, then instantiation will fail.

Using a factory to find a component instance is similar to creating a component instance; the runtime performs a lookup on the component type registry to find a component factory then calls its finder method with the "well-known" name of the desired instance. If the instance does not already exist, the component factory creates the instance on-the-fly. The purpose of instance lookup is to enable component instance sharing.

4.3 Mapping to OSGi

The Beanome runtime is implemented as an OSGi bundle. If a component interface implementation requires access to the runtime, it must implement an interface called `RuntimeClient`; this allows the implementation to receive a reference to the runtime when it is instantiated.

Bundles are used to package one or more Beanome component types. These component type bundles must register their component factories in the OSGi service registry. To simplify this process, a generic bundle activator is exported by the Beanome runtime bundle. The generic activator parses the component descriptor file and automatically registers a `ComponentFactoryService` interface for each component type in the file. Factory service registration takes place even if the required dependencies of the component type are not fulfilled, because interface implementation resolution does not take place until instantiation.

When the generic activator registers factories, it uses the standard OSGi mechanism to attach to each factory service a set of properties that include the name of the component type, the type's associated version number, and the set of interfaces that the component provides. The list of provided interfaces is necessary because the runtime or another component instance can request a component factory whose component type provides a given set of interfaces, regardless of its name or version.

If a component type requires an interface implemented by another component (i.e., there is a "requires" clause in the component descriptor), then these dependencies must be resolved upon instantiation. These required dependencies are captured as LDAP query expressions (see figure 3). LDAP query expressions are the standard way to request services in OSGi. The query expressions are written in terms of the properties attached to the component factory services, such as type name or version. An interesting feature of this approach is that implementation resolution need not be precise; in other words, a component type can describe an interface dependency in such a way to allow for multiple possible resolutions, such as different versions of the same component type or different component types altogether.

5 Current Status and Future Work

Currently there is an implementation of the Beanome runtime that provides the following features:

- a generic activator that automatically interprets component descriptor files and registers component factories,
- component type instantiation and retrieval,
- introspection facilities on component types and instances, and
- access to the component type registry.

In addition to these features, several runtime extensions have been prototyped for dealing with dependency conflict resolution, automatic deployment of bundles to resolve missing dependencies, and shared instance persistence; these extensions to the runtime are implemented as OSGi services.

In Beanome, the full spectrum of desirable service-to-service dependency types, as described in subsection 3.1, is not completely supported. The current dependency semantics, which occur between factory services, are one-to-one and static; this means there is only one source and one target of the dependency, the dependency is required for instantiation, and the dependency is not affected by changes in the environment. This definition of a dependency is limited and forces complex dependency handling into the application.

In standard OSGi, an approach for removing complex dependency handling from application code is to define the concept of a binding operation that is invokable when desired services arrive so that it can be set externally. This binding operation is provided so that the bundle classes or service implementations can receive references to the services they need. Using such an approach, it is possible to describe application-level binding operations declaratively in the bundle's meta-data so that binding can be processed externally.

Beanome introduces a new level of dependency not found in standard OSGi, since component instances can use services, instance-to-service dependencies must also be supported. To support these dependencies, the component descriptor format must be extended to include cardinality and dynamism information. These changes to the component descriptor format also require extending the Beanome runtime to properly handle optional dependencies and to invoke appropriate binding operations at runtime.

A prototype of a service binder has been built to support the different types of dependencies mentioned in subsection 3.1. This prototype provides automatic runtime binding support for bundle-to-service and service-to-service dependencies, which are described in an XML file. This prototype has not yet been fully integrated with the Beanome framework so instance-to-services dependencies are not yet covered.

Another important issue that requires further investigation concerns the departure of required services or component types after a component has been instantiated. The current Beanome prototype handles this situation by trying to revoke all component instances and their associated interface implementation instances, but it cannot provide any guarantees that the instances will not continue to be used since direct references to these object may be cached in other objects. Perhaps a more consistent approach would be to shutdown the entire dependency graph, but a more fine-grained solution is preferable.

6 Related work

Beanome is not creating a component model and framework for execution directly on consumer devices, instead it is creating component concepts specifically for the OSGi framework, which in turn focuses on the consumer device domain. Given this goal of Beanome, it is difficult to compare it to other component models and framework for OSGi, since no others are known to exist. Instead of direct comparisons, it is necessary to compare Beanome to existing component models.

Beanome concepts are similar to those found in other component models. The concept of component factories can be found in models such as EJB [7] and CCM [3]. Beanome component types have a similar structure to CCM component types. The differences to CCM are that CCM allows a component type to provide the same interface multiple times using different names and that CCM includes the notion of event sources and sinks. The Beanome introspection mechanisms leverage the standard JavaBeans [8] introspection technique for type-level introspection, but Beanome extends the introspection capabilities to include instance-level introspection. Both COM and CCM also allow type- and instance-level introspection, similar to Beanome.

Another similarity to existing component models is that there is a separation between units of delivery and components types. With respect to Beanome, the unit of delivery is the OSGi bundle. For COM the unit of delivery is a dynamic linked library or the executable file itself. EJB and CCM also have their own delivery units.

A related project is ROBOCOP [10] whose aim is to define "an open, component-based partial architecture for the middleware layer in high volume embedded appliances that enables robust and reliable operation, upgrading and extension, and component trading." ROBOCOP and Beanome share several concepts, which are inspired from COM ideas. ROBOCOP's executable components are comparable to bundles and ROBOCOP's services and service instances are comparable to Beanome's component types and component instances. ROBOCOP, however, is not built on top of OSGi, nor does it address dynamic dependencies.

Another related research project is PECOS [6], which aims to enable component-based software development for embedded systems. It provides a component model that extends industrial component models to address the needs of embedded systems and also provides an "ultra-light" component environment where component-based applications are installed, tested, and tuned. The PECOS component model has been defined to reflect an architectural style built of components, ports, and connectors. PECOS is specifically defined to be used on field devices, while Beanome is not, but it is possible to consider Beanome for such cases when running the OSGi framework on limited resource Java virtual machines, like PersonalJava™.

7 Conclusions

The OSGi framework provides a strong foundation on which to build service-oriented applications that target the consumer device and residential gateway domains. The OSGi framework benefits from the precise focus of its specification and the result is a framework that is both conceptually and physically lightweight. A side-effect of this precise focus is that complexity arises when building sophisticated applications on top of the framework because OSGi is rather low-level.

Beanome, presented in this paper, adds a simple component model and framework on top of

the OSGi. The Beanome component model is implemented using standard OSGi mechanisms and does not require any proprietary extensions. In Beanome, OSGi bundles become delivery mechanism for component types, which can be used to instantiate complex application graphs. The benefits of Beanome include: a hierarchical component model, an "application" concept for deploying complex dependency graphs into the OSGi framework, and an extensible component runtime using OSGi services.

Currently, a prototype exists that illustrates all of the concepts described in this paper, except for instance-to-service dynamic dependencies. Future work will continue to concentrate on dynamic dependencies, separation of the runtime into independent OSGi services, and defining and improving services for extending the runtime, such as a persistence service and a dependency conflict service. For further information on Beanome, visit http://www-adele.imag.fr/BEANOME.

8 References

[1] Box, D., "Essential COM." Addison Wesley, January 1998.

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

[3] Object Management Group, "CORBA Components: Joint Revised Submission," August 1999.

[4] Open Services Gateway Initiative, http://www.osgi.org, 2002

[5] Open Services Gateway Initiative, "OSGI Service Platform," Release 2, October 2001.

[6] Nierstrasz, O., et al., "A Component Model for Field Devices," 1st Proceedings of the International IFIP/ACM Working Conference on Component Deployment, June 2002.

[7] Sun Microsystems, "Enterprise JavaBeans Specification," Version 2.0, 2001.

[8] Sun Microsystems, "JavaBeans Specification," Version 1.01 edition, 1997.

[9] 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

[10] ITEA, ROBOCOP : Robust Open Component Based Software Architecture for Configurable Devices Project – Framework concepts. Public Document V1.0, May 2002