Using the *Bridge* Design Pattern for OSGi Service Update

Hans Werner Pohl Jens Gerlach
{hans,jens}@first.fraunhofer.de

Fraunhofer Institute for
Computer Architecture and Software Technology (FIRST)
Berlin Germany

Abstract

In the OSGi framework, components cooperate by sharing service objects. The suggested way to replace a service by a newer version consists of updating its containing components which requires a temporary shutdown of the component. Special care must be taken to avoid dangling references to old service instances.

As this appears to be an overly expensive strategy, we describe the use of the well-known *Bridge* design pattern to decouple service replacement from component updates. Instead of registering services only references to instances of automatically generated bridge classes are registered. This solves not only the problem of dangling references but also avoids stopping and starting dependent bundles.

1 Introduction

The Open Service Gateway Initiative (OSGi)[1] is a consortium that has specified a Java^{TM} component framework[2] for delivering, activating, and replacing services over wide-area networks to local-area networks and devices. Since OSGi server implementations can be quite small, it is a good candidate when looking for supporting runtime evolution in embedded systems.

OSGi components interact by sharing so-called *service objects*. Services are registered by one component and referenced by others. A major problem of OSGi component exchange at runtime is the replacement of service objects. Shutting down all dependent components and restarting the replaced sub-system, as recommended by OSGi, seems overly expensive.

In this paper, we describe the use of the well-known *Bridge* design pattern to decouple service replacement from bundle updates. Instead of registering services only references to instances of automatically generated *bridge* classes are registered. Our approach fits smoothly into the OSGi component framework as clients of services are not affected by our approach.

Our discussion includes the issues of *state transfer* and *synchronization* during service update. We also present and discuss two approaches to generate the necessary *Bridge* classes. The first one relies on generating bridge classes at design time whereas the second approach uses Java^{TM} proxies to provide a generic bridge class.

2 Bundles and Services in OSGi

In OSGi, components are referred to as *bundles*. A bundles consists of a JAR file that contains a manifest file, Java^{TM} class files and other resources. One of these classes must implement the interface `BundleActivator` of the OSGi framework (see Listing 1).

```
interface BundleActivator
{
   void start(BundleContext context);
   void stop (BundleContext context);
}
```

Listing 1: The `BundleActivator` interface

A bundle is deployed to an OSGi server. When a deployed bundle is started the method `start` of its activator classes is called. Note the type `BundleContext` of the `start` and `stop` methods. The Bundles communicate with the OSGi server through the `context` argument.

Bundles cooperate by accessing *services* that are offered by other bundles. The following subsections explain in the context of OSGi, what is a service, and how bundles cooperate through them.

2.1 Services in OSGi

A service is nothing more than an object of a class that implements one or more *service interfaces*. Listing 2 gives a simple example of a class `FooBar` that implements two service interfaces `IFoo` and `IBar`.

```
public interface IFoo {
   public void foo();
}

public interface IBar {
   public int bar(int i);
}

public class FooBar implements IFoo,IBar {
   public void foo()     {...}
   public int bar(int i) {...}
}
```

Listing 2: A service class implementing two service interfaces

2.1.1 Registration of Services

A bundle offers a service by *registering* it at the OSGi server. An example of registering a service is shown in Listing 3. Note that in order to register a service object the *names* of service interfaces must be supplied.

```
String[] names = {"IFoo","IBar"};
ServiceRegistration reg =
   context.registerService(names, new FooBar(),...);
```

Listing 3: Registering a service object

The registering bundle obtains an object of type `ServiceRegistration` upon successful registration. The `ServiceRegistration` object can be used to unregister a service object—see Section 2.2. However, under normal circumstance bundles do not have to unregister their services since the OSGi specification[2] states that all registered services are *automatically unregistered* when a bundle is stopped.

2.1.2 Accessing Registered Services

Bundles that wish to use services do not directly request service objects. In order to obtain access to a registered service, a bundle queries the OSGi server by providing *names* of service interfaces—see Listing 4.
If a service object has been registered under these interface names, the OSGi server returns an object of type `ServiceReference`. The requesting bundle obtains a “real” reference to the registered service object only through such a `ServiceReference`.

```
ServiceReference ref = context.getServiceReference("IFoo");
IFoo service = (IFoo)context.getService(ref);
service.foo();  // call to a service method
```

Listing 4: Obtaining access to a service object

2.2 Service Update in OSGi

A bundle that has registered a service can unregister it as long as it holds the corresponding `ServiceRegistration` object (see Listing 3). All what the registering bundle has to do is calling the `unregister` method and registering a new service object as shown here.

```
reg.unregister();
reg = context.registerService(names,new NewFooBar(),...);
```

However, this only works if the registering bundles had *anticipated* that it might be necessary to replace a service object.

Even more severe is that this kind of update may lead to dangling reference in bundles that have obtained a `ServiceReference`. One way to avoid dangling service references in client bundles is to use *service listeners* of OSGi. However, solely relying on listeners shifts the burden on the clients of a service.

OSGi recommends that instead of updating individual service objects their registering bundle is updated which implicitely includes that the bundle is stopped and and a new version of the bundle is started. OSGi ensures that all registered services of a bundle are unregistered when the bundle is stopped. It is the task of the new bundle start method to register corresponding service objects.

In the following section we show that this expensive solution can be avoided and that service objects can be individually updated without invalidation of references.

3 Service Update with the *Bridge* Design Pattern

The aim of the *Bridge* design pattern[3] is to "decouple an abstraction from its implementation so that the two can vary independently". The *Bridge* pattern is useful when "you want to avoid a permanent binding between an abstraction and its implementation. This might be the case, for example, when the implementation must be selected or switched at run-time". The price is an additional indirection.

The *Proxy* pattern also incorporates an additional indirection and resembles therefore the *Bridge* pattern. However, they differ at least in the intention. A proxy provides "a surrogate or placeholder for another object to control access to it" [3]. For example, proxies are used to support access to remote objects, to create objects on demand, or to enable protection.

There are several ways to use indirection for service update. They differ in the tradeoff between performance and flexibility. Wrapper functions and interprocedures[4] are used to deal with changing implementations and interfaces and resulting state transfer. A *Delegator* pattern has also been proposed[5], which resembles the *Facade* pattern and also allows to change interfaces. This is achieved by a standardized format of all communications, e.g. XML descriptions. Services can be dynamically plugged into the delegator and depending on the the XML description of the call the delegator decides which service are called and transforms the respective arguments.

Our approach does not rest on dynamically plugable interfaces. Rather, interfaces are "frozen" within bridges and can not be changed there. Nevertheless, it is no problem if a new client wishes to access a new service object through a new interface. The new service implementing both the old and new interfaces and a new bridge with the new interface solve this problem. The old bridge has to redirect requests to the new service object. This way old clients can stay connected with services that offer new interfaces in addition to old ones.

Finally, a remark concerning performance: We do not expect any performance penalties when using the *Bridge* pattern unless the amount of work performed by the service is *very* small.

3.1 Transforming Services Classes

The basics of our *Bridge* pattern transformation is best demonstrated by its application to the example of Listing 2. The service implementing class and the interfaces remain unchanged. Note that the generated bridge only refers to the services interfaces and not to a particular service class.

```
public class FooBar_Bridge implements IFoo,IBar {
   private Object impl;

   public Object getImpl()            { return impl; }
   public void setImpl(Object object) { impl = object; }

   public void foo()      { ((IFoo)impl).foo();}
   public int bar(int i) { return ((IBar)impl).bar(i); }
}
```

Listing 5: The bridge class of our example

Using the generated bridge class is quite simple. Instead of registering a `FooBar` service object, a bundle has to register a `FooBar_Bridge`. Since client bundles refer to a service only through its interface(s) they notice no difference.

The registering bundle can now easily replace a service object at run-time as it is shown in Listing 6. The update can even be performed by another bundle.

```
ServiceReference ref = context.getServiceReference("IFoo");
FooBar_Bridge bridge = (FooBar_Bridge)context.getService(ref);
bridge.setImpl(new NewFooBar());
```

Listing 6: The dynamic update by an exchange bundle

With services following the *Bridge* pattern a service can be explicitely *invalidated* through `bridge.setImpl(null)`. This is not possible if (non-bridge) service objects are directly registered since the Java^{TM} garbage collector can free a service object only when no client bundle references it any longer.

3.2 Technical Issues of Service Update

In this section we discuss issues of state transfer (§3.2.1) and synchronization (§3.2.2) when applying the *Bridge* pattern to represent services.

Other important issues are *security* and the treatment of *service factories*. Regarding the relationship of service update and the OSGi security concepts[2], we only mention that for changing the service implementation a bundle must have the same service permissions as for registering the service.

Updating service factories requires both updating the factory and updating the customized service implementation objects. One problem that cannot be solved easily is that with the current OSGi specification service registering bundle has no access to the individual service objects of other bundles.

3.2.1 State Transfer

A service may have state that must be preserved over the the update. In a simple case it may be enough to equip the new service with a constructor having an old service object as single argument. The designer of the new version has to know the old implementation.

However, this tight binding of the new service to the old one may hinder reusability. The introduction of `getState` and `setState` functionality appears to be more appropriate. This resembles the *Memento* design pattern[3]. In contrast to the *Memento* pattern this functionality belongs to the service implementing classes, not to the memento classes (`getState` of the *Memento* pattern corresponds to `setState` here). Moreover, in contrast to the *Memento* pattern it seems to be a good idea to define a generic memento class. This generic implementation can rely on Java^{TM} Properties or XML descriptions as possible candidates.

3.2.2 Synchronization

Even in the case that there are no synchronization issues for the original system we have to ensure the thread safety of the system extended by bridge classes.

First we consider the simple case that all methods of the service object are declared as `synchronized`. If there is no state transfer, we define the `setImpl()` method as follows:

```
   public void setImpl(Object object) {
      synchronized(impl) { impl = object;}
   }
```

If state transfer is an issue it will not be sufficient to only synchronize inside of the `getImpl` and `setImpl` methods because another thread may have changed the state of the service between these calls. One possibility to solve this problem is to synchronize "outside" of the bridge.

```
synchronized(bridge.getImpl()) {
   Object obj = bridge.getImpl();
   // compute newImpl depending on obj
   bridge.setImpl(newImpl);
}
```

Now we consider the case that the service methods have not been declared `synchronized`. The aim is to allow (at least in principle) that service invocations work as they do without bridge. However, an update must not take place while a service is in use and vice versa (mutual exclusion).

Listing 7 presents one solution. The bridge class gets an additional private integer member `useCount` that holds the actual number of users. In case it is not zero the thread invoking the `setImpl` method has to wait. Later on the thread can be waked up by the last thread that invoked `foo`. On the other hand, while performing an update (which may include a state transfer) no `foo` method can enter because the incrementation of `useCount` is also synchronized with the bridge object.

3.3 Generation of Bridge Class

Since we want to deal with unanticipated changes it is of paramount importance that users themselves need not to program the bridge classes. Fortunately, the generation of the bridge classes is straightforward and so (as in Hicks' system[6]) it can easily be automated.

We present two approaches. The first one generates the bridge classes at design time. The second approach use a generic and very flexible bridge implementation. Both rely on Java$^{\text{TM}}$ reflection.

```
int foo() {
   synchronized(this) {useCount++;}
   ((IFoo)ipml).foo();
   synchronized(this) {
      useCount--;
      if (useCount == 0) notifyAll();
   }
}

public synchronized void setImpl(Object object) {
   while(useCount != 0) wait();
   impl = object;
}
```

Listing 7: Synchronization in the general case

3.3.1 Automatic Generation At Design Time

A convenient way to perform the code generation is to use the *reflection* mechanisms of the Java^{TM} platform. No external tools are necessary. There is no problem if the target platform does not fully support reflection—as in the case of the Java^{TM} 2 Micro Edition[7]—since reflection is used at design time only.

Using reflection, it is not hard to automatically generate constructors of the bridge class which initialize the private `impl` member through the related constructors of the implementation class. Registration of bridge-service then looks almost exactly as without bridge.

3.3.2 Using Dynamic Proxies

By defining a single universal `Bridge` class with Java^{TM} *proxies* there is no need to generate bridge classes at design-time (see Listing 8). This approach requires comprehensive support for run time reflection which is unrealistic for embedded systems. In particular J2ME[7] does not support Java^{TM} proxies. Moreover, there are higher run time costs caused by additional indirections.

The client calls the Java^{TM} proxy, an object of a class with no (in the program visible) name. This object calls the `invoke` method of the `Bridge` object, which finally calls the *method*'s invoke method.

For convenience we define static methods of the `Bridge` class (see Listing 9). The `newInstance` method generates the proxy with invocation handler which are related to the final serving object. The proxy object is registered by the server bundle. The method `setImpl` is used to perform the dynamic update, `getImpl` is defined analogously. As in the case of generated bridge classes there are no changes for clients of a service.

```
class Bridge implements InvocationHandler {
   private Object impl;
   private Bridge(Object obj) { impl = obj; }

   public Object
   invoke(Object proxy, Method method, Object[] args) {
      Object result = null;
      try {
         result = method.invoke(impl,args);
      } catch(Exception e) {...}
      return result;
   }
   // static methods ...
}
```

Listing 8: The bridge invocation handler

We also want to mention another way to define the invocation handler. It is possible to enrich the interfaces by the `setImpl` and `getImpl` methods inside of the `newInstance` method. This requires to check the invoked methods within the `Bridge.invoke` method.

4 Conclusions

OSGi bundles cooperate through sharing of service objects. On a conceptual level there is no tight coupling between a bundle that registers a service and another bundle that uses it because both bundles need only to know the *interfaces* and not the *exact type* of the service object. However, as a client bundle obtains a reference to the service object it is difficult for the registering bundle to replace the service object.

Our approach, using the *Bridge* pattern, provides the separation at the object level, and thus, simplifies the exchange of services. Moreover, since the bridge class of a service class implements the same service interfaces no changes are necessary for clients of a service.

We also have shown that state transfer and synchronization of service update can be treated well with the *Bridge* pattern. In order ease the burden of implementing bridge classes we have presented two solutions. The first solution consists of generating at design time a new bridge class for each service class. This has the advantage that no support for run time reflection is required which generally cannot be expected for embedded systems. In contrast, the second approach relies heavily on Java^{TM} proxies but provides a single powerful bridge class.

```
public static Object newInstance(Object obj) {
   return Proxy.newProxyInstance(
      obj.getClass().getClassLoader(),
      obj.getClass().getInterfaces(),
      new Bridge(obj));
}

public static void setImpl(Object proxy, Object obj) {
   Bridge bridge =
      (Bridge)Proxy.getInvocationHandler((Proxy)proxy);
   bridge.impl = obj;
}
```

Listing 9: Static methods of `Bridge`

References

[1] Open Service Gateway Initiative. *Home Page of the OSGi Consortium.* `http://www.osgi.org`.

[2] Open Services Gateway Initiative. *OSGi Service Platform, Release 2.* IOS Press, 2001.

[3] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[4] M. Segal and O. Frieder. On dynamically updating a computer program: from concept to prototype. *The Journal of Systems Software*, pages 111–128, February 1991.

[5] J. Gorinsek, S. Van Baelen, Y. Berbers, and K. De Vlaminck. Empress: Component based evolution for embedded systems. in ECOOP 2002 Workshop on Unanticipated Software Evolution (USE2002) G. Kniesel, P. Costanza and M. Dimitriev (eds.), http://joint.org/use2002/, June 2002.

[6] M. Hicks. Dynamic software updating. PhD thesis, Department of Computer and Information Science, University of Pensylvania, June 2001.

[7] Sun Microsystems. *Java^{TM} 2 Platform, Micro Edition.* http://java.sun.com/j2me/.