Listeners Considered Harmful: The "Whiteboard" Pattern

Technical Whitepaper

Revision 2.1
April 2019

Introduction

The purpose of this whitepaper is to explain a pattern that was found to be very successful in the usage of the OSGi specifications. The dynamic nature of the OSGi service model requires extra effort from the programmer to track the changes. The traditional model with listeners was found to be overly complicated and error prone. This paper analyzes the issues and proposes an easier and inherently more reliable model.

Background

The Listener Pattern

Java 1.0 contained a poorly designed model for handling events for user interface components in AWT. A major problem with this model is the event storms that can happen as well as overall inefficiency. When a new component model was created, the event model was redesigned to fit the new component model (beans). Cornerstones of this event model are the following parties:

1 The *event source* – an object that can generate an event.

2 The *event object* – an object that carries the information about the event.

3 The *event listener* – an object that receives the event.

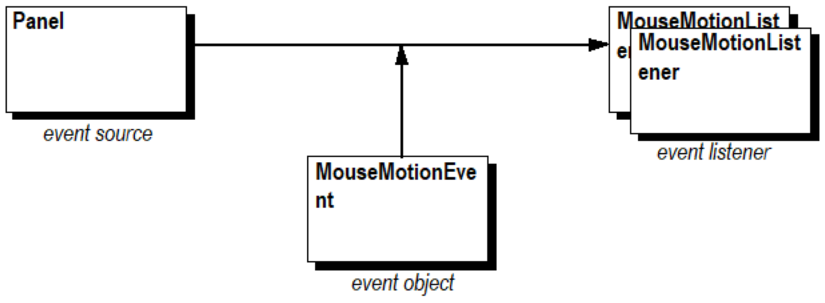

Figure 1: Example actors in the event model of Java

For example, to receive notification about the motion of a mouse, an object implementing the MouseMotionListener interface must be registered with an event source, such as an AWT Panel. The event source provides notification in the form of a MouseMotionEvent object sent to all listeners registered with the event source.

Some listener interfaces have many methods and are thus non-trivial to implement. To reduce the implementation effort required for a listener, default implementations of the listener interface are often provided and are called *adapters*.

One of the novel aspects of the listener pattern was the fact that event sources can support multiple listeners. Prior to the listener pattern, it was common to have APIs for which a single target object could be set with a callback. The listener pattern implicitly assumes that any number of listeners can be registered and that having a single listener is the exception. This was a huge improvement by making programmers aware that they are not in full control and that they must share the VM and application objects with others.

The listener pattern allows an introspective application to connect different components with events and listeners. It quickly became successful, and applications like the BeanBox use the listener pattern to dynamically create applications from a graphic UI.

However, the listener pattern has drawbacks. Presently, there are more than 130 event, adapter, and listener classes in Java. This creates an overhead in class files, affecting start-up time and program size, and also increases runtime overhead. In most cases, an event source has only a single listener registered. Still, the event source must take the overhead of correctly delivering an event to many listeners. This requires at least one extra List to hold the list of listeners and creates demand for “monster” classes like AWTEventMulticaster that try to maintain type safety of the pattern while still reaping the benefits of reusability. These problems are not important for desktop or server applications. When memory and CPU speed is abundant, a 100KB class file overhead for listener, adapter, and events classes is not a concern. In contrast, embedded environments, one of the targets of the OSGi specification, *are* sensitive to these overheads.

Another issue that is not obvious is the dependency that is created between the event source and the listener. Designs must correctly manage this dependency. If the event source goes away, the listener must clean up any references it holds. If the listener goes away, the event source should remove it from the list of listeners. These are the so-called life cycle issues. They usually are not a concern for traditional Java applications. The initialization phase of a traditional application, where listeners are added, is easy to implement correctly and even easier to verify (it just does not work if done wrong). However, the removal phase is much more difficult to verify and is often not handled at all. In workstation environments, where an application is started by the user, the life cycle management of listeners is a non-issue. Most applications assume that the cleanup is done when exiting or that no clean up is necessary. Again, no such assumptions can be made for embedded applications that run continuously and can be extremely dynamic. For example, the tutorial concerning how to write a MouseMotionListener[1] does not mention the fact that listeners should have a life cycle. **Error! Reference source not found.** contains an article that discusses these *loiterers* and offers some solutions.

The OSGi Environment

The requirements for the OSGi environment include the following:

- Small devices
- Collaboration model
- Continuously up and running VM
- Life cycle management

These requirements come at a great cost.

Small devices are usually heavily constrained in persistent storage (often flash memory). Each class file has an overhead of at least 300 to 500 bytes, which adds up surprisingly fast. Use of these classes results in additional overhead. This implies that the number of classes should be kept low. Toward this end, during the early phases of the OSGi specification development, it was decided not to create ad-hoc exceptions or adapter classes.

Small devices are also constrained in performance and dynamic memory. Therefore, during the development of the OSGi specification, attempts were made to minimize the requirements for creating superfluous objects that used memory.

The OSGi collaboration model is implemented with the *service registry*. This registry allows *bundles* (applications) to register services. Services are normal Java objects that are typically defined by a Java interface allowing different implementations to co-exist. The dynamic nature of the registry makes it necessary to track services that can come and go at any time.

The continuously up and running nature of the OSGi environment combined with the life cycle management and service registry requires different programming rules. The requirements of the OSGi environment invalidate the following major hidden assumption in most Java code: *Once you have a reference to an object, the object does not go away*.

In an OSGi environment, the owner of an object can, and at some point, will go away. Services are dynamically added and removed from the registry. Java packages previously available may become unavailable. This highly dynamic environment dictates that original programming patterns must be revisited and reconsidered for this environment. A more reactive mindset is required.

A case where this is necessary is usage of the listener pattern in the OSGi environment. Here, an event listener must take action when a an event source is unregistered. Vice-versa, the event source must monitor the bundle of the event listener and take action when this bundle is stopped. It turned out that managing these dependencies is not trivial.

The Whiteboard Pattern

The whiteboard pattern leverages the OSGi framework's service registry instead of implementing a private registry as required by the listener pattern. Instead of having event listeners track event sources and then register themselves with the event source, the whiteboard pattern has event listeners register themselves as a service with the OSGi framework. When the event source has an event object to deliver, the event source calls all event listeners currently registered in the service registry.

Remarkably, the event source is **not** registered with the framework as a service. This makes bundles using the whiteboard pattern significantly smaller and easier to implement. The inter-bundle dependency between the event source and the event listener is handled by the framework and requires almost no code in the event source and event listener bundles.

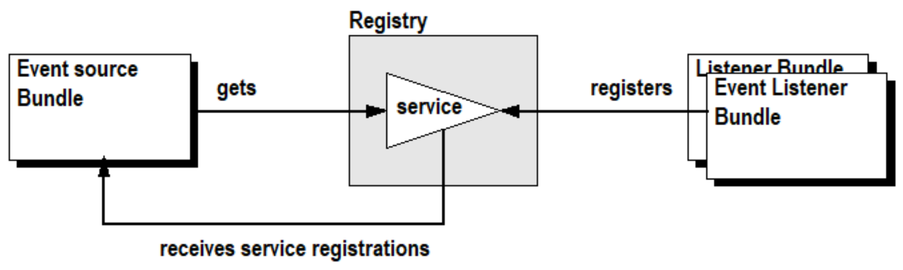

Figure 1: Whiteboard actors in the OSGi framework

Designs must often trade off between complexity in application bundles and server bundles. Servers by their nature are fewer in number and more system oriented. This usually means that it is better to place the complexity in the server and not in the client. However, the whiteboard is one of those rare cases where this tradeoff is not necessary. Both server and application become simpler because they reuse the framework registry. Both application bundles and server bundles can delegate the responsibility for managing the details of inter-bundle dependencies to the framework.

As an added benefit, the use of the OSGi framework's service registry provides more than just life cycle management. The service registry has additional benefits for the programmer:

- Debugging. The event listeners are visible in the registry and any framework support tool can inspect the registry. This makes inter-bundle dependencies more visible.
- Security. OSGi framework ServicePermissions can control access to an event source because event listeners must have permission to register the event listener interface.
- Properties. The registry supports properties that can be used by the server to select a subset of all listeners. This mechanism can be used for configuration management.

An Example

An LCD Display Bundle

The primary problem in OSGi environments is the handling of inter-bundle dependencies: one bundle referencing an object owned or created by another bundle. This problem can be explained using the following simple example.

The example consists of an LCD Display service that cycles through a number of screens. The content of the screens is provided by ContentProviders. This is a simple interface that can be implemented by any bundles that wants to be displayed on the LCD. ContentProviders are queried for their content when a new screen is needed. In this example, a Display is responsible for cycling through all the ContentProviders and a Clock implementing ContentProvider delivers the current time. System.out is used instead of a real display to keep the example as simple as possible.

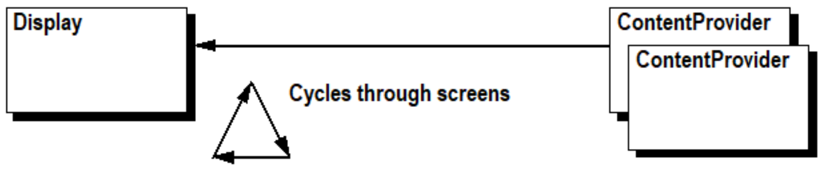

Figure 2: Actors in the example LCD Display application

The focus of this example is to show how the dependency between the Display and the ContentProviders are handled. In the OSGi environment, the Display and the ContentProviders *both* can come and go at any moment.

This example will be implemented in the following sections by first using the listener pattern followed by the whiteboard pattern.

Coding Rules

Code size comparisons between different approaches are subjective. It is easy to bloat one approach and minimize the other approach by combining classes and expressions. In the following approaches to the example, the same coding rules are followed. Different responsibilities (interfaces) are implemented in their own class. No expressions are combined and one statement is written per line.

Listeners Implementation

The listener approach uses a ContentProvider (ClockManager) that tracks the Display object. The DisplayManager registers each ContentProvider and cycles through all the active ContentProviders.

The Display and ContentProvider interfaces must be implemented by any object wanting to act as a Display or a ContentProvider, respectively.

```
package org.osgi.example.display;
public interface Display {
     void addContentProvider( ContentProvider p );
     void removeContentProvider( ContentProvider p );
}
package org.osgi.example.display;
public interface ContentProvider {
     String getContent();
}
```

Figure 3: Interfaces with the listener pattern

Next is the implementation of the ClockManager. The ClockManager tracks Display services in the registry with a DisplayTracker. The DisplayTracker extends the ServiceTracker. This utility class is designed to simplify monitoring services.

When a new Display service is registered with the OSGi framework, a Clock object is created and registered with the Display service as a ContentProvider. These objects are unregistered when the bundle is stopped or when the Display service is unregistered.[1]

The ClockManager class implements the BundleActivator to keep the code simple.

```
package org.osgi.example.listener.clock;
import org.osgi.example.display.*;
import org.osgi.framework.*;
import org.osgi.util.tracker.*;
import java.util.*;

class Clock implements ContentProvider {
     private final Display display;
     Clock( Display display ) {
           this.display = display;
           display.addContentProvider( this );
     }
     void dispose() {
           display.removeContentProvider(this);
     }
     public String getContent() { return new Date().toString(); }
```

[1] The code size could be slightly reduced by implementing the ServiceTrackerCustomizer interface on the ClockManager rather than extending ServiceTracker. However, this is not done because it only works for a single listener type. When an application uses the listener pattern, there are usually several listener types to be implemented. This problem led to inner classes being introduced in Java 1.1.

```
}

class DisplayTracker extends ServiceTracker<Display, Clock> {
     DisplayTracker( BundleContext context ) {
           super( context, Display.class, null );
     }
     public Clock addingService( ServiceReference<Display> ref ) {
           Display display = context.getService(ref);
           return new Clock( display );
     }
     public void removedService( ServiceReference<Display> ref, Clock clock ) {
           clock.dispose();
           context.ungetService( ref );
     }
}

public class ClockManager implements BundleActivator {
     private DisplayTracker tracker;
     public void start( BundleContext context ) {
           tracker = new DisplayTracker( context );
           tracker.open();
     }
     public void stop( BundleContext context ) {
           tracker.close();
     }
}
```

Figure 4: ClockManager source using the listener pattern

Next is the implementation of the DisplayManager. The Display service *must* be implemented using a ServiceFactory. A ServiceFactory allows the DisplayManager to monitor the bundles registering ContentProvider services, and take action when such a bundle *gets* and *ungets* this Display service. It is provably impossible to write such a service correctly without using a ServiceFactory.

A single bundle can register more than one ContentProvider. For each bundle, a ContentProviderRegistration object is created that tracks the multiple ContentProvider registrations from that bundle. This allows the DisplayManager to properly remove all ContentProviders registered by a bundle when that bundle releases the Display service, for example, when the bundle is stopped.

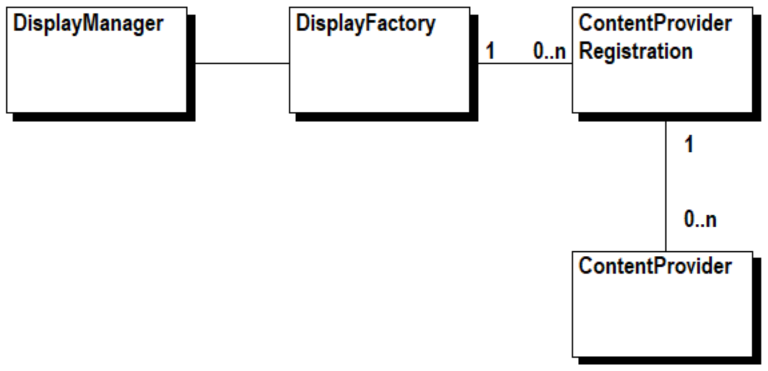

Figure 5: Multiple ContentProvider services per bundle

To keep the code simple, the DisplayManager implements BundleActivator and Runnable. The run() method implements the core display cycle as an infinite loop, cycling through all the registered ContentProviders at 5 second intervals.

```
package org.osgi.example.listener.display;
import org.osgi.framework.*;
import java.util.*;
import org.osgi.example.display.*;

class ContentProviderRegistration implements Display {
     private final List<ContentProvider> providers;
     private final DisplayManager    manager;

     ContentProviderRegistration( DisplayManager manager ) {
          this.manager = manager;
          providers = new ArrayList<>();
     }
     public synchronized void addContentProvider( ContentProvider p ) {
          providers.add ( p );
          manager.addContentProvider(p);
     }
     public synchronized void removeContentProvider( ContentProvider p ) {
          if ( providers.remove( p ) ) {
               manager.removeContentProvider( p );
          }
     }
     void synchronized dispose() {
          for ( ContentProvider p : providers ) {
               manager.removeContentProvider( p );
          }
          providers.clear();
     }
}

class DisplayFactory implements ServiceFactory<Display> {
     private final DisplayManager    manager;
     DisplayFactory( DisplayManager manager ) {
          this.manager = manager;
     }
     public Display getService( Bundle b, ServiceRegistration<Display> r ) {
          return new ContentProviderRegistration( manager );
     }
     public void ungetService( Bundle b, ServiceRegistration<Display> r, Display s ) {
          ContentProviderRegistration cpr = (ContentProviderRegistration) s;
          cpr.dispose();
     }
}

public class DisplayManager implements BundleActivator, Runnable {
     private volatile Thread thread;
     private ServiceRegistration<Display> registration;
     private final List<ContentProvider> providers = new ArrayList<>();
     public void start( BundleContext context ) {
          DisplayFactory factory= new DisplayFactory( this );
          registration = context.registerService(
               Display.class,
               factory,
               null
          );
          thread = new Thread( this, "DisplayManager Listener" );
          thread.start();
     }
     public void stop( BundleContext context ) {
          thread = null;
     }
     public void  run() {
          Thread     current = Thread.currentThread();
          int        n = 0;
          while ( current == thread ) {
               synchronized (providers) {
                    if ( !providers.isEmpty() ) {
                         if ( n >= providers.size() )
                              n = 0;
                         ContentProvider cp = providers.get(n++);
                         System.out.println( "LISTENER: " + cp.getContent() );
               }
               }
```

```
                try { Thread.sleep( 5000 ); } catch( InterruptedException e ) {}
            }
        }
        void addContentProvider( ContentProvider p ) {
            synchronized (providers) {
                providers.add (p);
            }
        }
        void removeContentProvider( ContentProvider p ) {
            synchronized (providers) {
                providers.remove (p);
            }
        }
}
```

Figure 6: DisplayManager source with the listener pattern

Whiteboard Implementation

Instead of tracking Display objects and then registering a ContentProvider with the Display, the whiteboard approach only registers a ContentProvider as a service with the OSGi framework. This ContentProvider is tracked by the DisplayManager. The DisplayManager is **not** registered with the framework. This makes the code significantly smaller and easier to implement.

Another advantage of the whiteboard approach is the requirement for only a single service interface: the ContentProvider.

```
package org.osgi.example.display;
public interface ContentProvider {
    String getContent();
}
```

Figure 7: Interface with the whiteboard pattern

When using Declarative Services, no ClockManager implementation is necessary as Service Component Runtime (SCR), the runtime implementation of Declarative Services, handles all the details of managing the service of the Clock component. Registration and unregistration of the Clock's ContentProvider service is completely delegated to the SCR. The Component annotation causes SCR to register the component under the names of any directly implemented interfaces which is ContentProvider in this case.

```
package org.osgi.example.whiteboard.clock;
```

```
import org.osgi.service.component.annotations.*;
import org.osgi.example.display.*;
import java.util.*;

@Component
public class Clock implements ContentProvider {
     public Clock( ) {}
     public String getContent() { return new Date().toString(); }
}
```

Figure 8: Clock source with the whiteboard pattern

The DisplayManager must track all the ContentProvider objects in the registry. Using Declarative Services, this can be handled automatically by SCR. As a result, the DisplayManager implementation becomes significantly smaller. The DisplayManager component does not register a service, so we specify `service={}` in the Component annotation since, by default, the component is registered as a service under the names of any directly implemented interfaces. The Reference annotation on the providers field tell SCR to track and inject all the ContentProvider services, since the field is a List of ContentProvider. Because the providers field is declared volatile, SCR will update the field with a replacement List whenever the list of tracked ContentProvider services changes. This means it is always safe to quickly iterate over a reference to the list since list will not change once injected in the field.

As in the Listener example, the DisplayManager implements the display cycle in a thread that calls the run() method.

```
package org.osgi.example.whiteboard.display;
import org.osgi.service.component.annotations.*;
import org.osgi.example.display.*;

@Component(service = {})
public class DisplayManager implements Runnable {
     private volatile Thread thread;

     @Reference
     private volatile List<ContentProvider> providers;

     @Activate
     void activate() {
          thread = new Thread( this, "DisplayManager Component" );
          thread.start();
     }
     @Deactivate
```

```
        void deactivate() {
              thread = null;
        }
        public void  run() {
              Thread     current = Thread.currentThread();
              int        n = 0;
              while ( current == thread ) {
                    List<ContentProvider> providers = this.providers;
                    if ( !providers.isEmpty() ) {
                          if ( n >= providers.size() )
                                n = 0;
                          ContentProvider cp = providers.get(n++);
                          System.out.println( "COMPONENT: "  + cp.getContent() );
                    }
                    try { Thread.sleep( 5000 ); } catch( InterruptedException e ) {}
              }
        }
  }
```

Figure 9: DisplayManager source with the whiteboard pattern

Comparison

The following table provides a line-by-line code comparison of the implementation with the listener versus the implementation with the whiteboard using Declarative Services.

Table 1: Listeners versus Whiteboard

Listener Pattern	Whiteboard Pattern

public interface Display {
  void addContentProvider(ContentProvider p);
  void removeContentProvider(ContentProvider p);
}	

public interface ContentProvider {
  String getContent();
}
	public interface ContentProvider {
	  String getContent();
	}

class Clock implements ContentProvider {
  private final Display   display;
  Clock(Display display) {
    this.display = display;
    display.addContentProvider(this);
  }
  void dispose() {
    display.removeContentProvider(this);
  }
  public String getContent() { return new Date().toString(); }
}
	@Component
	public class Clock implements ContentProvider {
	
	  public Clock( ) {}
	
	  public String getContent() { return new Date().toString(); }
	}

Listener Pattern	Whiteboard Pattern

class DisplayTracker extends ServiceTracker<Display, Clock> {
  DisplayTracker(BundleContext context) {
    super(context, Display.class, null);
  }
  public Display addingService(ServiceReference<Display> ref) {
    Display display = context.getService(ref);
    return new Clock(display);
  }
  public void removedService(ServiceReference<Display> ref, Clock clock) {
    clock.dispose();
    context.ungetService(ref);
  }
}	

public class ClockManager implements BundleActivator {
  private DisplayTracker   tracker;
  public void start(BundleContext context) {
    tracker = new DisplayTracker(this, context);
    tracker.open();
  }
  public void stop(BundleContext context) {
    tracker.close();
  }
}	

class ContentProviderRegistration implements Display {
  private final List< ContentProvider> providers;
  private final DisplayManager   manager;
 	
  ContentProviderRegistration(DisplayManager manager) {
    this.manager = manager;
    providers = new ArrayList<>();
  }
  public synchronized void addContentProvider(ContentProvider p) {
    providers.add(p);
    manager.addContentProvider(p);
  }
  public synchronized void removeContentProvider(ContentProvider p) {
    if (providers.remove(p)) {
      manager.removeContentProvider(p);
    }
  }
  void synchronized dispose() {
    for (ContentProvider p : providers) {
      manager.removeContentProvider(p);
    }
    providers.clear();
  }
}	

class DisplayFactory implements ServiceFactory<Display> {
  private final DisplayManager   manager;
  DisplayFactory(DisplayManager manager) {
    this.manager = manager;
  }
  public Display getService(Bundle b, ServiceRegistration<Display> r) {
    return new ContentProviderRegistration(manager);
  }
  public void ungetService(Bundle b, ServiceRegistration<Display> r, Display s) {
    ContentProviderRegistration cpr = (ContentProviderRegistration) s;
    cpr.dispose();
  }
}	

Listener Pattern

```
public class DisplayManager implements BundleActivator, Runnable {
  private volatile Thread                    thread;
  private ServiceRegistration<Display> registration;
  private final List<ContentProvider>     providers = new ArrayList<>();

  public void start( BundleContext context ) {
      DisplayFactory factory = new DisplayFactory( this );
      registration = context.registerService( Display.class, factory, null );
      thread = new Thread( this, "DisplayManager Listener" );
      thread.start();
  }

  public void stop( BundleContext context ) {
      thread = null;
  }
  public void  run() {
      Thread    current = Thread.currentThread();
      int          n = 0;
      while ( current == thread ) {
          synchronized ( providers ) {
              if ( !providers.isEmpty() ) {
                  if ( n >= providers.size() )
                      n = 0;
                  ContentProvider cp = providers.get(n++);
                  System.out.println( "LISTENER: " + cp.getContent() );
              }
          }
          try { Thread.sleep( 5000 ); } catch( InterruptedException e ) {}
      }
  }
  void addContentProvider( ContentProvider p ) {
      synchronized ( providers ) {
        providers.addElement(p);
      }
  }
  void removeContentProvider( ContentProvider p ) {
      synchronized ( providers ) {
          providers.removeElement(p);
      }
  }
}
```

Whiteboard Pattern

```
@Component(service = {})
public class DisplayManager implements Runnable {
  private volatile Thread                    thread;
  @Reference
  private volatile List<ContentProvider> providers
  @Activate
  void activate() {


      thread = new Thread( this, "DisplayManager Component" );
      thread.start();
  }
  @Deactivate
  void deactivate() {
      thread = null;
  }
  public void  run() {
      Thread    current = Thread.currentThread();
      int          n = 0;
      while ( current == thread ) {
          List<ContentProvider> providers = this.providers;
          if ( !providers.isEmpty() ) {
              if ( n >= providers.size() )
                  n = 0;
              ContentProvider cp = providers.get(n++);
              System.out.println( "COMPONENT: "  + cp.getContent() );
          }

          try { Thread.sleep( 5000 ); } catch( InterruptedException e ) {}
      }
  }
}
```

Obviously, the whiteboard approach is a significantly smaller and simpler implementation especially since we use Declarative Services to manage the lifecycle of the components and their use of services. It is even simpler than the code size suggests. What is not obvious is the fact that the listener pattern has significantly more deadlock possibilities than the whiteboard pattern.

See GitHub[3] for the example code above.

Conclusion

The whiteboard pattern leverages the solid mechanisms in the OSGi framework and Declarative Services to address the dynamic life cycle of objects in the OSGi environment. It significantly simplifies handling life cycle issues. Both the Display implementation as well as the ContentProvider implementation become significantly smaller and easier to understand, and are far less prone to programming errors. And the use of Declarative Services allows the classes to be plain old Java objects (POJOs). This means the classes do not directly reference the OSGi API. The interaction with the OSGi API is fully handled by SCR.

The listener pattern attempts to put the programming rules and patterns of traditional Java application development on top of the OSGi environment. In practice, the listener pattern is not suited for the dynamic changes one can, and *must*, expect in the OSGi environment. The OSGi environment changes some of the basic programming rules with which Java programmers are familiar since it is more reactive in nature.

So why is the whiteboard pattern not always accepted at face value?
Obviously, the learning curve of the OSGi environment is a factor. The many programming patterns that apply to traditional Java applications must be reconsidered in OSGi. Newcomers to OSGi attempt to apply the standard patterns without realizing the intricacies and issues.

Another and more subtle reason might be programmer philosophy. The listener approach places the programmer of the listener fully in control. He decides the event sources with which to register his listener, and he is in charge of all the dependencies between event source and listener.

This philosophy is questionable in the OSGi environment. A bundle is not an application that runs independently from other applications. It is a component running in the framework that can collaborate with other bundles. Bundles need to be managed and configured by operators. To allow these operators to manage the bundles, bundles need to be written with the philosophy that they are being used and are not in direct control. Bundles should be written to provide a service that can be configured together with other bundles and to provide maximum composability. Often these other bundles are unknown to the designer of the original bundle. The whiteboard pattern supports the philosophy of bundles not being in direct control significantly better than the listener pattern.

All these reasons are compelling reasons to apply the whiteboard pattern whenever there is a design that requires managing inter-bundle dependencies.

Document Information

References

[1] *https://docs.oracle.com/javase/tutorial/uiswing/events/mousemotionlistener.html*

[2] *https://www.javaworld.com/article/2077581/java-tip-79--interact-with-garbage-collector-to-avoid-memory-leaks.html*

[3] *https://github.com/osgi/whiteboard-pattern*

Authors

Peter Kriens, aQute, 9C, Avenue St. Drézéry, 34160 France
Peter.Kriens@aQute.biz

BJ Hargrave, IBM, 325 Bill France Blvd, Daytona Beach, FL 32114 USA
hargrave@us.ibm.com

The OSGi Alliance