OSGi In Practice

Neil Bartlett

December 17, 2009

Contents

List of Figures

List of Tables

List of Code Listings

Preface

Part I.

Nuts and Bolts

1. Introduction

Consider the following question. In any large engineering project, for example the design of a new bridge, skyscraper, or jet airliner, what is nearly always the most difficult challenge to overcome?

The answer is *Complexity*.

The Boeing 747-400 "Jumbo Jet" has six million parts[1], of which half are fasteners. It contains 171 miles (274 km) of wiring and 5 miles (8 km) of tubing. Seventy-five thousand engineering drawings were used to produce the original 747 design. It is a very, *very* complex machine, and because of this complexity, no single person can have a complete understand of how a Boeing 747 works. Yet, this is a machine first designed in the 1960s — modern aircraft have multiplied the level of complexity many times over. The only approach that will allow human beings to design such incredible machines is to break them down into smaller, more understandable modules.

Modularity enables several important benefits:

Division of Labour. We can assign separate individuals or groups to work on separate modules. The people working on a module will have a thorough understanding of their own module but not all the others. For example, the designer of the entertainment system does not need to know anything about how the landing gear works (and *vice versa*!) but can concentrate all her efforts on building the best possible entertainment system.

Abstraction. We can now think about the aircraft as an abstract whole, without needing a complete understanding of every part. For example, we grasp that a 747 is able to fly due to the lift provided by the wings and the forward thrust of the engines, even if we do not understand exactly how fuel is supplied to the engines or how many motors are required to move the flaps up and down.

Reuse. Given the amount of effort that goes into designing even some of the smaller components of an aircraft, it would be a shame to have to start from scratch when we need an identical or very similar component in another aircraft, or even in another part of the same aircraft. It would be helpful if we could reuse components with minimal alterations.

Ease of Maintenance and Repair. It would be a shame to have to scrap an entire aeroplane over a single burst tyre or torn seat cover. Modular

designs allow for failed modules to be removed and either repaired or replaced without affecting the rest of the machine.

It's debatable whether the production of software can be characterised as an engineering discipline, but nevertheless the complexity of sofware rivals what is seen in other fields. Modern software is incredibly complex, and furthermore is accelerating in its complexity. For example, the onboard computers of the NASA Space Shuttle contained half a million lines of code, but today the DVD player in your living room contains around one million lines of code. Microsoft Windows XP is estimated to contain 40 million lines, and Windows Vista 50 million. A BMW 7-series car can contain up to 50 networked computers.

Just like aircraft engineers, software professionals are in the business of creating large and complex machines, which we can only hope to understand by breaking them into modules.

The Java™ programming language is one of the most popular languages today for building both large, enterprise applications and also small but widely deployed mobile applications. However Java alone does not support modularity in any useful way... we will see in the next section why Java's existing mechanisms fail to deliver all four of the above listed benefits of modularity. However, Java's great strength is its flexibility, which has allowed a powerful module system to be built on top.

That module system is called OSGi. OSGi is nothing more nor less than the way to build modular applications in Java.

1.1. What is a Module?

So, what should a software module look like? It should have the following properties:

Self-Contained. A module is a logical whole: it can be moved around, installed and uninstalled as a single unit. It is not an atom — it consists of smaller parts — but those parts cannot stand alone, and the module may cease to function if any single part is removed.

Highly Cohesive. Cohesion is a measure of how strongly related or focussed the responsibilities of a module are. A module should not do many unrelated things, but stick to one logical purpose and fulfil that purpose well.

Loosely Coupled. A module should not be concerned with the internal implementation of other modules that it interacts with. Loose coupling allows us to change the implementation of one module without needing to update all other modules that use it (along with any modules that use *those* modules, and so on).

To support all three properties, it is vital for modules to have a *well-defined interface* for interaction with other modules. A stable interface enforces logical boundaries between modules and prevents access to internal implementation details. Ideally, the interface should be defined in terms of what each module offers to other modules, and what each module requires from other modules.

1.2. The Problem(s) with JARs

The standard unit of deployment in Java is the JAR file, as documented by the JAR File Specification[2]. JARs are archive files based on the ZIP file format, allowing many files to be aggregated into a single file. Typically the files contained in the archive are a mixture of compiled Java class files and resource files such as images and documents. Additionally the specification defines a standard location within a JAR archive for metadata — the `META-INF` folder — and several standard file names and formats within that directly, most important of which is the `MANIFEST.MF` file.

JAR files typically provide either a single library or a portion of the functionality of an application. Rarely is an entire application delivered as a single JAR, unless it is very small. Therefore, constructing Java applications requires composing together many JAR files: large applications can have a JAR count in double or even triple figures. And yet the Java Development Kit (JDK) provides only very rudimentary tools and technologies for managing and composing JARs. In fact the tools available are so simplistic that the term "JAR Hell"[1] has been coined for the problem of managing JARs, and the strange error messages that one encounters when things go wrong.

The three biggest problems with JAR files as a unit of deployment are as follows:

- There is no runtime concept that corresponds to a JAR; they are only meaningful at build-time and deploy-time. Once the Java Virtual Machine is running, the contents of all the JARs are simply concatenated and treated as a single, global list: the so-called "Classpath". This model scales very poorly.
- They contain no standard metadata to indicate their dependencies.
- They are not versioned, and multiple versions of JARs cannot be loaded simultaneous.
- There is no mechanism for information hiding between JARs.

[1] A reference to the term DLL Hell: a comparable problem with managing DLL files on the Microsoft Windows operating system.

1.2.1. Class Loading and the Global Classpath

The term "classpath" originates from the command-line parameter that is passed to the `java` command when running simple Java applications from the command shell (or the DOS prompt in Windows terms), or more usually from some kind of launcher script. It specifies a list of JAR files and directories containing compiled Java class files. For example the following command launches a Java application with both `log4j.jar` and the `classes` directory on the classpath. First as it would be under UNIX (including Mac OS X):

```
java -classpath log4j.jar:classes org.example.HelloWorld
```

And then the DOS/Windows version:

```
java -classpath log4j.jar;classes org.example.HelloWorld
```

The final parameter is the name of the "main" class to execute, which we will assume has been compiled into the `org/example/HelloWorld.class` file in the `classes` directory. Somehow the Java Virtual Machine (JVM) must load the bytes in that class file and transform them into a `Class` object, on which it can then execute the static `main` method. Let's look at how this works in a standard Java runtime environment (JRE).

The class in Java that loads classes is `java.lang.ClassLoader`, and it has two responsibilities:

1. Finding classes, i.e. the physical bytes on disk, given their logical class names.
2. Transforming those physical bytes into a `Class` object in memory.

We can extend the `java.lang.ClassLoader` class and provide our own implementation of the first part, which allows us to extend the JRE to load code from a network or other non-file-based storage system. However the second part, i.e. transforming physical class bytes into `Class` objects, is implemented in `java.lang.ClassLoader` by the `defineClass` method, which is both `native` and `final`. In other words, this functionality is built into the JRE and cannot be overridden.

When we run the command above, the JRE determines that it needs to load the class `org.example.HelloWorld`. Because it is the "main" class, it consults a special `ClassLoader` named the application class loader. The first thing the application class loader does is ask its "parent" to load the class.

This illustrates a key feature of Java class loading called parent-first delegation. Class loaders are organised into a hierarchy, and by default all class loaders

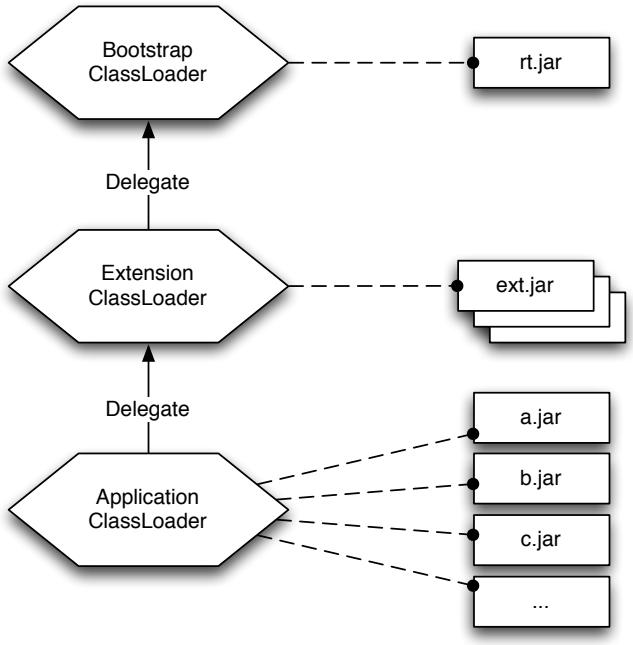


Figure 1.1.: The standard Java class loader hierarchy.

first ask their parent to load a class; only when the parent responds that it knows nothing about the specified class does the class loader attempt to find the class itself. Furthermore the parent will also delegate to its own parent, and so on until the top of the hierarchy is reached. Therefore classes will always be loaded at the highest possible level of the hierarchy.

Figure 1.1 shows the three standard class loaders found in a conventional Java application. The bootstrap class loader sits at the top of the tree, and it is responsible for loading all the classes in the base JRE library, for example everything with a package name beginning with `java`, `javax`, etc. Next down is the extension class loader, which loads from "extension" libraries, i.e. JARs which are not part of the base JRE library but have been installed into the `libext` directory of the JRE by an administrator. Finally there is the aforementioned application class loader, which loads from the "classpath".

Returning to our "HelloWorld" example, we can assume that the `HelloWorld` class is not present in the JRE base or extension libraries, and therefore it will not be found by either the bootstrap or extension class loaders. Therefore the application class loader will try to find it, and it does that by looking inside each entry in the classpath in turn and stopping when it finds the first match. If a match is not found in any entry, then we will see the familiar

`ClassNotFoundException`.

The `HelloWorld` class is probably not inside `log4j.jar`, but it will be found in the `classes` directory, and loaded by the class loader. This will inevitably trigger the loading of several classes that `HelloWorld` depends on, such as its super-class (even if that is only `java.lang.Object`), any classes used in its method bodies and so on. For each of those classes, the whole procedure described above is repeated.

To recap:

1. The JRE asks the application class loader to load a class.
2. The application class loader asks the extension class loader to load the class.
3. The extension class loader asks the bootstrap class loader to load the class.
4. The bootstrap class loader fails to find the class, so the extension class loader tries to find it.
5. The extension class loader fails to find the class, so the application class loader tries to find it, looking first in `log4j.jar`.
6. The class is not in `log4j.jar` so the class loader looks in the `classes` directory.
7. The class is found and loaded, which may trigger the loading of further classes — for each of these we go back to step 1.

1.2.2. Conflicting Classes

The process for loading classes in Java does at least work for much of the time. However, consider what would happen if we accidentally added a JAR file to our classpath containing an older version of `HelloWorld`. Let's call that file `obsolete.jar`.

```
java -classpath obsolete.jar:log4j.jar:classes \
     org.example.HelloWorld
```

Since `obsolete.jar` appears before `classes` in the classpath, and since the application class loader stops as soon as it finds a match, this command will always result in the old version of `HelloWorld` being used. The version in our `classes` directory will *never* be used. This can be one of the most frustrating problems to diagnose in Java: it can appear that a class is not doing what it should be doing, and the changes we are making to it are not having any

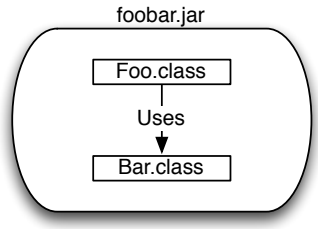


Figure 1.2.: Example of an internal JAR dependency.

effect, simply because that class is being shadowed by another with the same name.

Perhaps this scenario does not seem very likely, and it's true that in a trivial application like this, such a mistake is easily avoided. However, as we discussed, a large Java application can consist of tens or even hundreds of individual JAR files, all of which must be listed on the classpath. Also, JAR files frequently have obscure names which give little clue as to their contents. Faced with such problems it becomes much more likely — perhaps even inevitable — that the classpath will contain conflicting names. For example, one of the commonest cause of conflicts is including two different versions of the same library.

By simply searching classpath entries in order and stopping on the first match, the JRE reduces JARs to mere lists of class files, which are dumped into a single flat list. The boundaries of the "inner" lists — that is, the JARs themselves — are essentially forgotten. For example, suppose a JAR contains two internal classes, `Foo` and `Bar`, such that `Foo` uses `Bar`. This is shown in Figure 1.2.

Presumably, `Foo` expects to resolve to the version of `Bar` that is shipped alongside it. It has almost certainly been built and tested under that assumption. However at runtime the connection between the two classes is arbitrarily broken because another JAR happens to contain a class called `Bar`, as shown in Figure 1.3

1.2.3. Lack of Explicit Dependencies

Some JAR files are "standalone", i.e. they provide functionality without depending on any other libraries except the base JRE libraries. However, many build on the functionality offered by other JARs, and therefore they can only be used if they are deployed alongside those other JARs. For example, the Apache Jakarta Commons HttpClient[3] library depends on both Commons Codec and Commons Logging, so it will not work without `commons-logging.jar` and `commons-codec.jar` being present on our classpath.

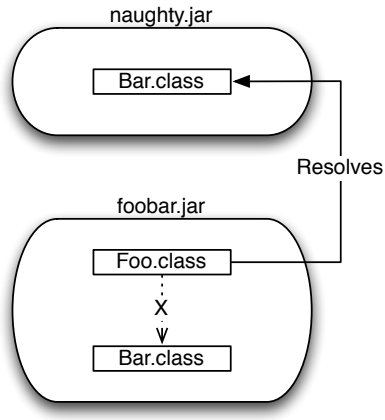


Figure 1.3.: A broken internal JAR dependency.

But how do we know that such a dependency exists? Fortunately HttpClient is well documented, and the dependency is explicitly noted on the project web site. But many libraries do not have such good documentation. Instead the dependency is implicit: lurking inside the class file in the JAR, ready to cause a `ClassNotFoundException` when we try to use it.

In fact, even the good documentation exemplified by HttpClient is not really good enough. What we want is a standard way to declare dependencies, preferably right in the JAR file, such that the declarations can be analysed easily by tools.

There was an early attempt in Java to supply such information through the `Class-Path` attribute, which can be specified in `MANIFEST.MF`. Unfortunately this mechanism is almost entirely useless, because it only allows one to list further JAR files to be added to the classpath using absolute file-system paths, or paths relative to the file-system location of the JAR in question.

1.2.4. Lack of Version Information

The world does not stand still, and neither do the libraries available to us as Java programmers. New libraries and new versions of existing libraries appear all the time.

Therefore it is not enough merely to indicate a dependency on a particular library: we must also know which *version* is required. For example, suppose we determine somehow that a JAR has a dependency on Log4J. Which version of Log4J do we need to supply to make the JAR work? The download site lists 25 different versions (at time of writing). We cannot simply assume the

latest version, since the JAR in question may not have been tested against it, and in the case of Log4J there is an experimental version 1.3.x that almost nobody uses.

Again, documentation sometimes comes to the rescue (e.g., "this library requires version 1.2.10 of Log4J or greater"), but unfortunately this kind of information is very rare, and even when it does exist it is not in a format that can be used by tools. So we need a way to tag our explicitly declared dependencies with a version..... in fact, we need to specify a version *range* because depending on a single specific version of a library would make our system brittle.

Versions cause other problems. Suppose our application requires two libraries, A and B, and both of these libraries depend in turn upon a third library, C. But they require different versions: library A requires version 1.2 or greater of C, but library B requires version 1.1 of C, and will not work with version 1.2! Figure 1.4 illustrates the problem.

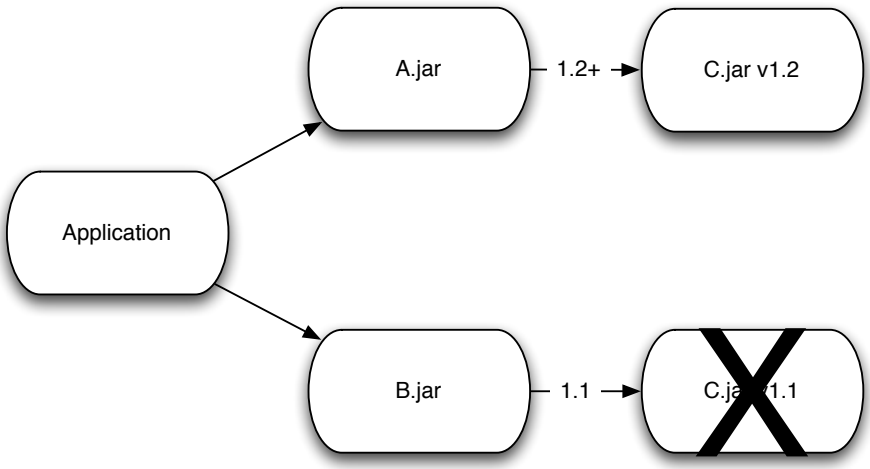


Figure 1.4.: Clashing Version Requirements

This kind of problem simply cannot be solved in traditional Java without rewriting either A or B so that they both work with the same version of C. The reason for this is simply the flat, global classpath: if we try to put both versions of C on the classpath, then only the first version listed will be used, so both A and B will have to use it. Note that version 1.2 in this example does *not* win because it is the higher version; it wins merely because it was placed first in the classpath.

However, some classes from version 1.1 may still be visible if they have different

names from the classes in 1.2! If *B* manages to use one of those classes, then that class will probably attempt to use other classes that are shadowed by version 1.2, and it will get the 1.2 version of those classes rather than the 1.1 version that it has been compiled against. This is an example of the class loading conflict from Figure 1.2. The result is likely to be an error such as `LinkageError`, which few Java developers know how to deal with.

1.2.5. Lack of Information Hiding Across JARs

All Object Oriented Programming languages offer various ways of hiding information across class and module boundaries. Hiding — or *encapsulating* — internal details of a class is essential in order to allow those internal details to change without affecting clients of the class. As such, Java provides four access levels for members of a class, which include both fields and methods:

- `public` members are visible to everybody.
- `protected` members are visible to subclasses and other classes in the same package.
- `private` members are visible only within the same class.
- Members not declared with any of the three previous access levels take the so-called *default* access level. They are visible to other classes within the same package, but not classes outside the package.

For classes themselves, only the `public` and default access levels are available. A public class is visible to every class in every other package; a default access class is only available to other classes within the same package.

There is something missing here. The above access modifiers relate to visibility across packages, but the unit of deployment in Java is not a package, it is a JAR file. Most JAR files offering non-trivial APIs contain more than one package (HttpClient has eight), and generally the classes within the JAR need to have access to the classes in other packages of the same JAR. Unfortunately that means we must make most of our classes `public`, because that is the only access modifier which makes classes visible across package boundaries.

As a consequence, all those classes declared `public` are accessible to clients outside the JAR as well. Therefore the whole JAR is effectively public API, even the parts that we would prefer to keep hidden. This is another symptom of the lack of any runtime representation for JAR files.

1.2.6. Recap: JARs Are Not Modules

We've now looked in detail at some specific problems with JAR files. Now let's recap why they do not meet the requirements for a module system.

Simply, JAR files have almost none of the characteristics of a module as described in Section 1.1. Yes, they are a unit that can be physically moved around... but having moved a JAR we have no idea whether it will still work, because we do not know what dependencies might be missing or incorrect.

JARs are often tightly coupled: thanks to the lack of information hiding, clients can easily use internal implementation details of a JAR, and then break when those details change. And those are just the clients who break encapsulation intentionally; other clients may accidentally use the internals of a JAR thanks to a classpath conflict. Finally, JARs often have low cohesion: since a JAR does not really exist at runtime, we might as well throw any functionality into whichever JAR we like.

Of course, none of this implies that building modular systems in Java is impossible[2]. It simply means that Java provides no assistance towards the goal of modularity. Therefore building modular systems is a matter of *process* and *discipline*. Sadly, it is rare to see large Java applications that are modular, because a disciplined modular approach is usually the first thing discarded when an important deadline looms. Therefore most such applications are a mountain of "spaghetti code", and a maintenance nightmare.

1.3. J2EE Class Loading

The Java 2 Enterprise Edition (J2EE) specification defines a platform for distributed, multi-tier computing. The central feature of the J2EE architecture is the so-called "application server" which hosts multiple application components and offers enterprise-class services to them such as transaction management, security and high availability. Application servers are also required to have a deployment system so that applications can be deployed and un-deployed without restarting the server and without interfering with each other.

These requirements meant that the simplistic class loading hierarchy of Figure 1.1, as used by standalone Java applications, was not sufficient. With a single flat classpath, the classes from one application could easily interfere with other applications. Therefore J2EE application servers use a more complex tree, with a branch for each deployed application. The precise layout of the tree depends on the individual application server, since it is not mandated by the specification, but Figure 1.5 shows the approximate layout used by most servers.

J2EE applications are deployed as "Enterprise ARchive" (EAR) files, which are ZIP files containing a metadata file — `application.xml` — plus one or more of each of the following kind of file:

[2] Although certain scenarios are practically impossible, such as the side-by-side usage of different versions of the same library.

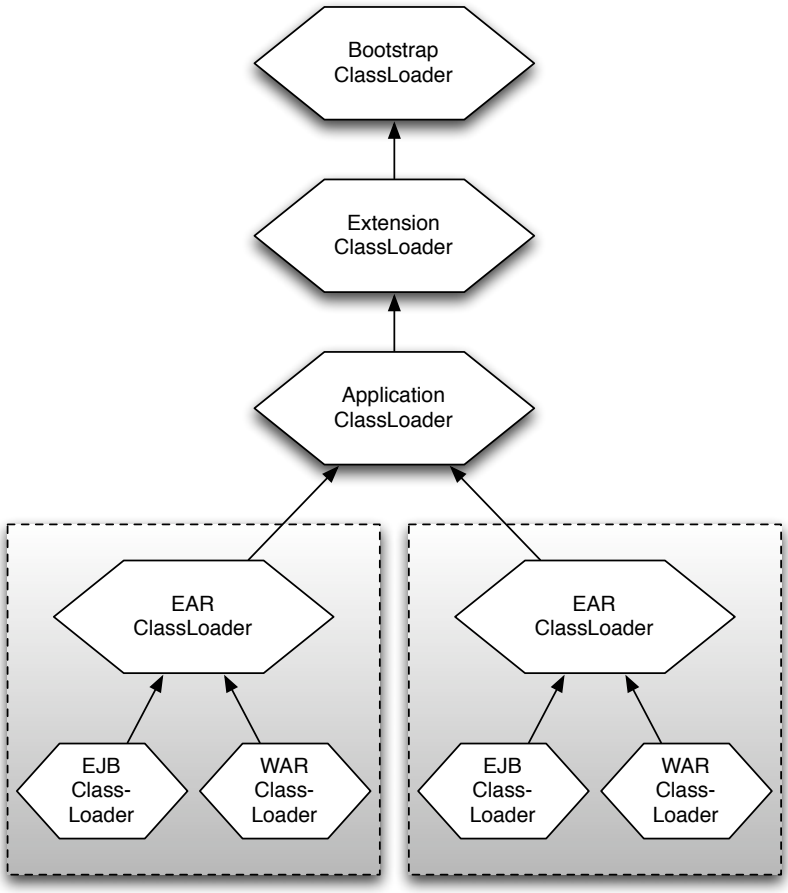


Figure 1.5.: A typical J2EE class loader hierarchy.

- Plain Java library JAR files
- JAR files containing an EJB application (EJB-JARs)
- "Web ARchive" (WAR) files, containing classes implementing Web functionality, such as Servlets and JSPs.

The plain Java library JAR files contain classes that are supposed to be available to all of the EJBs and Web artifact within the EAR. They are therefore loaded by the EAR Class Loader, at the top of the sub-tree for the deployed application.

Referring back to the class loading procedure described in section 1.2.1, it should be clear that, in a branching tree, an individual class loader can only load classes defined by itself or its ancestors; it cannot load classes sideways in the tree, i.e. from its siblings or cousins. Therefore classes that need to be shared across both the EJBs and Web artifacts must be pushed up the tree, into the EAR Class Loader. And if there are classes we wish to share across multiple deployed applications, we must push them up into the system application class loader. Usually this is done by configuring the way the application server itself is started, adding JARs to the global classpath.

Unfortunately, libraries pushed up to that level can no longer be deployed and un-deployed at will. Also, they become available to all deployed applications, even the ones that do not want or need to use them. They cause class conflicts: classes found higher in the tree always take precedence over the classes that are shipped with the application. In fact, every application in the server must now use the *same version* of the library.

Because of these problems, J2EE developers tend to avoid sharing classes across multiple applications. When a library is needed by several applications, it is simply shipped multiple times as part of the EAR file for each one. The end result is a "silo" or "stovepipe" architecture: applications within the J2EE server are little more than standalone systems, completely vertically integrated from the client tier to the database, but unable to horizontally integrate with each other. This hinders collaboration amongst different business areas.

1.4. OSGi: A Simple Idea

OSGi is the module system for Java. It defines a way to create true modules and a way for those modules to interact at runtime.

The central idea of OSGi is in fact rather simple. The source of so many problems in traditional Java is the global, flat classpath, so OSGi takes a completely different approach: each module has its own classpath, separate from the classpath of all other modules. This immediately eliminates almost

all of the problems that were discussed, but of course we cannot simply stop there: modules do still need to work together, which means sharing classes. The key is *control* over that sharing. OSGi has very specific and well-defined rules about how classes can be shared across modules, using a mechanism of explicit imports and exports.

So, what does an OSGi module look like? First, we don't call it a module: in OSGi, we refer to *bundles*.

In fact a bundle is just a JAR file! We do not need to define a new standard for packaging together classes and resources: the JAR file standard works just fine for that. We just need to add a little metadata to promote a JAR file into a bundle. The metadata consists of:

- The *name* of the bundle. We provide a "symbolic" name which is used by OSGi to determine the bundle's unique identity, and optionally we can also provide a human-readable, descriptive name.
- The *version* of the bundle.
- The list of *imports* and *exports*. We will see shortly exactly what is being imported and exported.
- Optionally, information about the minimum Java version that the bundle needs to run on.
- Miscellaneous human-readable information such as the vendor of the bundle, copyright statement, contact address, etc.

These metadata are placed inside the JAR file in a special file called `MANIFEST.MF`, which is part of all standard JAR files and is meant for exactly this purpose: it is a standard place to add arbitrary metadata.

The great advantage of using standard JAR files as OSGi bundles is that a bundle can also be used everywhere that a JAR file is expected. Not everybody uses OSGi (yet...) but since bundles are just JAR files, they can be used outside of the OSGi runtime. The extra metadata inside `MANIFEST.MF` is simply ignored by any application that does not understand it.

1.4.1. From Trees to Graphs

What does it mean to provide a separate classpath for each bundle? Simply, we provide a class loader for each bundle, and that class loader can see the classes and resources inside the bundle's JAR file. However in order for bundles to work together, there must be some way for class loading requests to be delegated from one bundle's class loader to another.

Recall that in both standard Java and Enterprise Edition, class loaders are arranged in a hierarchical tree, and class loading requests are always delegated

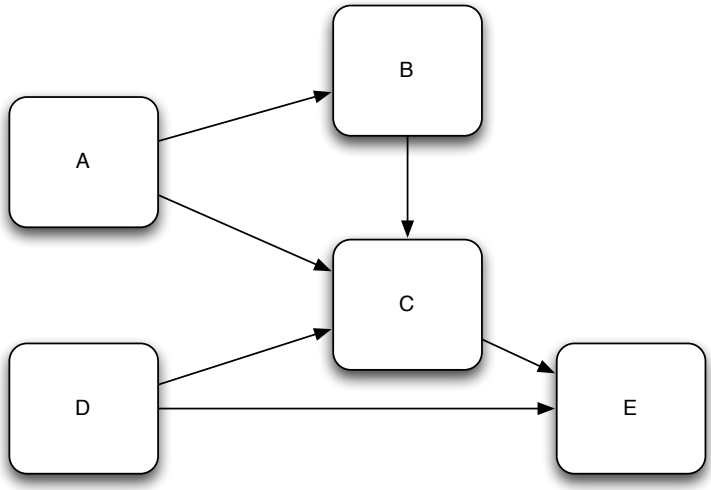


Figure 1.6.: The OSGi class loader graph.

upwards, to the parent of each class loader. Recall also that this arrangement does not allow for sharing of classes horizontally across the tree i.e., between siblings or cousins. To make a library available to multiple branches of the tree it must be pushed up into the common ancestor of those branches, but as soon as we do this we force everybody to use that version of that library, whether they like it or not.

Trees are simply the wrong shape: what we really need is a *graph*. The dependency relationship between two modules is not a hierarchical one: there is no "parent" or "child", only a network of providers and users. Class loading requests are delegated from one bundle's class loader to another's based on the dependency relationship between the bundles.

An example is shown in Figure 1.6. Here we have five libraries with a complex set of interdependencies, and it should be clear that we cannot easily force this into a hierarchical tree shape.

The links between bundles are based on imported and exported packages. That is, Java packages such as `javax.swing` or `org.apache.log4j`.

Suppose bundle B in Figure 1.6 contains a Java package named `org.foo`. It may choose to export that package by declaring it in the exports section of its `MANIFEST.MF`. Bundle A may then choose to import `org.foo` by declaring it in the imports section of its `MANIFEST.MF`. Now, the OSGi framework will take responsibility for matching up the import with a matching export: this is known as the resolution process. Resolution is quite complex, but it is implemented by the OSGi framework, not by bundles themselves. All that

we need to do, as developers of bundles, is write a few simple declarative statements.

Once an import is matched up with an export, the bundles involved are "wired" together for that specific package name. What this means is that when a class load request occurs in bundle A for any class in the `org.foo` package, that request will immediately be delegated to the class loader of bundle B. Other imports in A may be wired to other bundles, for example A may also import the package `org.bar` that is exported by bundle C, so any loading requests for classes in the `org.bar` package will be delegated to C's class loader. This is extremely efficient: whereas in standard Java class load events invariably involve searching through a long list of classes, OSGi class loaders generally know immediately where to find a class, with little or no searching.

What happens when resolution fails? For example, what if we forget to include bundle B with our application, and no other bundle offers package `org.foo` to satisfy the import statement in bundle A? In that case, A will not resolve, and cannot be used. We will also get a helpful error message telling us exactly why A could not be resolved. Assuming our bundles are correctly constructed (i.e., their metadata is accurate) then we should never see errors like `ClassNotFoundException` or `NoClassDefFoundError`[3]. In standard Java these errors can pop up at any time during the execution of an application, for example when a particular code path is followed for the first time. By contrast an OSGi-based application can tell us at start-up that something is wrong. In fact, using simple tools to inspect the bundle metadata, we can know about resolution errors in a set of bundles *before* we ever execute the application.

1.4.2. Information Hiding in OSGi Bundles

Note that we always talk about matching up an import with an export. But why are the export declarations even necessary? It should be possible simply to look at the contents of a bundle JAR to find out what packages are contained within it, so why duplicate this information in the exports section of the `MANIFEST.MF`?

The answer is, we don't necessarily want to export all packages from a bundle for reasons of information hiding as discussed in Section 1.2.5. In OSGi, only packages that are explicitly exported from a bundle can be imported and used in another bundle. Therefore all packages are "bundle private" by default, making it easy for us to hide the internal implementation details of our bundles from clients.

[3] An exception to this is where dynamic reflection is used to load arbitrary classes at runtime, a topic which is discussed in Chapter **??**

1.4.3. Versioning and Side-by-Side Versions

OSGi does not merely offer dependencies based on package names: it also adds versioning of packages. This allows us to cope with changes in the released versions of libraries that we use.

Exports of packages are declared with a version attribute, but imports declare a version *range*. This allows us to have a bundle depend on, e.g., version 1.2.0 through to version 2.1.0 of a library. If no bundle is exporting a version of that package that falls within the specified range, then our bundle will fail to resolve, and again we get a helpful error message telling us what is wrong.

We can even have different versions of the same library side-by-side in the same application, so the scenario described in Section 1.2.4 (wherein two libraries each need to use different versions of a third library) will work under OSGi.

Note that, because exports are declared on each exported package, there is no need for all exports from a bundle to be the same version. That is, a single bundle can export both version 1.2 of `org.foo` and version 3.5 of `org.bar`. Of course, it cannot export two versions of the *same* package.

1.5. Dynamic Modules

OSGi is not just the module system for Java. It is the *dynamic* module system for Java. Bundles in OSGi can be installed, updated and uninstalled without taking down the entire application. This makes it possible to, for example, upgrade parts of a server application — either to include new features or to fix bugs found in production — without affecting the running of other parts. Also, desktop applications can download new versions without requiring a restart, so the user doesn't even need to be interrupted.

To support dynamic modules, OSGi has a fully developed lifecycle layer, along with a programming model that allows "services" to be dynamically published and consumed across bundle boundaries. We will look at these facilities in depth in later osgi-in-practice.

Incidentally, many developers and system administrators are nervous or sceptical about OSGi's dynamic capabilities. This is perfectly understandable after the experience of J2EE, which offered limited and unreliable hot deployment of EAR modules. These people should still consider using OSGi anyway for its great modularity benefits, and feel free to ignore the lifecycle layer. Nevertheless, OSGi's dynamic capabilities are not mere hype: they really do work, and some experimentation will confirm that.

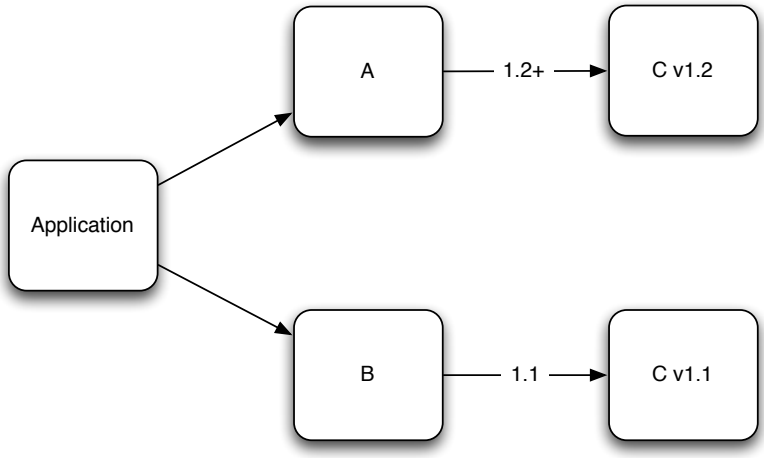


Figure 1.7.: Different Versions of the Same Bundle.

1.6. The OSGi Alliance and Standards

OSGi is a standard defined by an Alliance of around forty companies. Its specifications are freely available and are comprehensive enough that a highly compliant implementation can be written using only the documents for reference.

Two of the commonest questions about the name OSGi are: what does it stand for, and why is the "i" lower-case? Here is the definitive answer to both questions: officially OSGi *doesn't stand for anything*! However, it *used* to stand for "Open Service Gateway initiative", and this is the source of the lower-case letter "i", since "initiative" was not considered to be stylistically part of the brand name. But the long name is now deprecated, since OSGi has expanded far beyond its original role in home gateways. As a result, the OSGi name is rather odd, but it has the great advantage of being easily Google-able since it seems not to be a word (or even part of a word) in any language. In speech, the name is always spelt out ("Oh Ess Gee Eye") rather than pronounced as a word (e.g., "Ozjee").

The OSGi Alliance's role is to define the specification for new releases of the platform, and to certify implementations of the current release of the specification. The technical work is done by a number of Expert Groups (EGs) which include the Core Platform Expert Group (CPEG), Mobile (MEG), Vehicle (VEG) and Enterprise (EEG) Expert Groups. In this book we will mostly look at the work of the CPEG.

1.7. OSGi Implementations

Several independently implemented OSGi frameworks exist today, including four that are available as open source software.

Equinox [4] is the most widely deployed OSGi framework today owing to its use in the core runtime of Eclipse. It can also be found in many in-house custom applications as well as packaged products such as Lotus Notes and IBM WebSphere Application Server. Equinox implements Release 4.1 of the OSGi specifications and is licensed under the Eclipse Public License (EPL)[5].

Knopflerfish [6] is a popular and mature implementation of both OSGi Release 3 and Release 4.1 specifications. It is developed and maintained by Makewave AB, a company based in Sweden, and is licensed under a BSD-style license. Makewave also offers Knopflerfish Pro, a commercially supported version of Knopflerfish.

Felix [7] is a community implementation of the OSGi Release 4.x under the Apache Group umbrella. It is designed particularly for compactness and ease of embedding, and is the smallest (in terms of minimal JAR size) of the Release 4 implementations. It is licensed under the Apache License Version 2.0.

Concierge [8] is a very compact and highly optimized implementation of OSGi Release 3. This makes it particularly suited to resource-constrained platforms such as mobile phones. Concierge is licensed under a BSD-style license. However, OSGi Release 3 is not covered by this book[4].

All of the example code and applications in this book should work on any of the three listed Release 4 implementations, except where explicitly noted. However, the way we work with each of these frameworks — for example the command line parameters, the built-in shell, the management of bundles — differs enough that it would be awkward to give full instructions for all three. Therefore it is necessary to pick a single implementation for pedagogical purposes, and for reasons explained later we will work with Equinox.

1.8. Alternatives to OSGi

At its core, OSGi is very simple and, as with all good and simple ideas, many people have independently invented their own versions at different times and places. None of these have achieved the maturity or widespread usage that

[4] Also note that Concierge has not been certified compliant with OSGi R3, since OSGi Alliance rules only allow implementations of the *current* specification release to be certified.

OSGi now enjoys, but it is informative to review some of these alternatives. At the very least we may learn something about why OSGi has made the design choices it has made, but we may also find good ideas that can be brought into OSGi.

1.8.1. Build Tools: Maven and Ivy

Maven and Ivy are both popular tools that have some characteristics of a module system, but they are build-time tools rather than runtime frameworks. Thus they do not compete directly with OSGi, in fact they are complementary and many developers are using Maven or Ivy to build OSGi-based systems.

Maven is a complete, standalone build tool, whereas Ivy is a component that can be integrated into an ANT-based build. Both tools attempt to make JARs more manageable by adding modular features to them. Principally this means dependencies: both allow us to specify the versioned dependencies of a JAR using metadata in XML files. They use this information to download the correct set of JARs and construct a compile-time classpath.

However, as they do not have any runtime features neither Maven nor Ivy can solve the runtime problems with JARs, such as the flat global classpath, the lack of information hiding, and so on. Also the metadata formats used by these tools is unfortunately not compatible with the format used by OSGi, so if we use Maven or Ivy to build an OSGi-based system we typically have to specify the metadata twice: once for OSGi and once for the build tool. However, some efforts are currently being made to better integrate Maven with OSGi.

1.8.2. Eclipse Plug-in System

As already noted, the Eclipse IDE and platform are based on an implementation of OSGi. However this was not always the case: prior to version 3.0, Eclipse used its own custom module system.

In Eclipse terminology, a module is a "plug-in". In fact, Eclipse developers often still use the term plug-in as an alternative name for an OSGi bundle. In the old Eclipse system, a plug-in was a directory containing a file at the top level named `plugin.xml`. This file contained metadata that was broadly similar to the metadata in an OSGi manifest: the name of the plug-in, vendor, version, exported packages and required plug-ins.

Notice a key difference here. In the Eclipse plug-in system, dependencies were not declared at the level of Java packages but of whole plug-ins. We would declare a dependency on a plug-in based on its ID, and this would give us access to *all* of the exported packages in that plug-in. OSGi actually supports

whole-bundle dependencies also, but the use of this capability is frowned upon for reasons we will examine in Chapter 3.

The biggest deficiency of the Eclipse plug-in system was its inability to install, update or uninstall plug-ins dynamically: whenever the plug-in graph changed, a full restart was required. In early 2004 the core Eclipse developers began a project, code-named Equinox, to support dynamic plug-ins. They intended to do this either by enhancing the existing system or selecting an existing module system and adapting it to Eclipse. In the end, OSGi was selected and Equinox became a leading implementation of it.

1.8.3. JSR 277

Java Specification Request (JSR) number 277 is titled *Java Module System*[9] and as such one would expect it to attempt to solve a similar set of problems to OSGi. However JSR 277 tackles them in a different way to OSGi.

The most important point to note is that, at the time of writing, JSR 277 is an incomplete specification with no implementation yet. It is scheduled to be included with Java 7 but it is unclear when (or even if!) Java 7 will be released. An Early Draft Review (EDR) of JSR 277 was released in October 2006, which is the best information currently available.

Like the old Eclipse plug-in system, JSR 277 does not support package-level dependencies, instead using whole-module dependencies. However JSR 277 differs further by using programmatic resolution "scripts" rather than declarative statements to resolve module dependencies. This allows for extreme flexibility, but it's debatable whether such flexibility is ever necessary or desirable. Programmatic code can return different results at different times, so for example we could write a module that resolves successfully only on Tuesday afternoons! Therefore we completely lose the ability to use static dependency analysis tools.

Furthermore, JSR 277 is not dynamic, it requires the Java Virtual Machine to be restarted in order to install, update or uninstall a module.

In a sense, JSR 277 is an affirmation from its sponsors (principally Sun) that modularity is important and currently missing from standard Java. Sadly, JSR 277 comes many years later than OSGi, includes some questionable design features, and is substantially less ambitious despite its opportunity to change the underlying Java runtime. Therefore we hope that JSR 277 will at the very least be compatible with OSGi, since it currently appears to represent a significant step backwards for modularity in Java.

2. First Steps in OSGi

OSGi is a module system, but in OSGi we refer to modules as "bundles". In this chapter we will look at the structure of a bundle, how it depends on other bundles, and how to create an OSGi project using standard Java tools.

2.1. Bundle Construction

As discussed in the Introduction, an OSGi bundle is simply a JAR file, and the only difference between a bundle JAR and a "plain" JAR is a small amount of metadata added by OSGi in the form of additional headers in the `META-INF/MANIFEST.MF` file. Bundles can be used outside of an OSGi framework, for example in a plain Java application, because applications are required to ignore any attributes in the manifest that they do not understand.

Listing 2.1 A Typical OSGi MANIFEST.MF File

```
1 Manifest-Version: 1.0
2 Created-By: 1.4.2_06-b03 (Sun Microsystems Inc.)
3 Bundle-ManifestVersion: 2
4 Bundle-Name: My First OSGi Bundle
5 Bundle-SymbolicName: org.osgi.example1
6 Bundle-Version: 1.0.0
7 Bundle-RequiredExecutionEnvironment: J2SE-1.5
8 Import-Package: javax.swing
```

Listing 2.1 shows an example of a manifest containing some of the most common attributes used in OSGi. The lines in *italic font* are standard in all JAR file manifests, although only the `Manifest-Version` attribute is mandatory and it must appear as the first entry in the file. The JAR File Specification[2] describes several other optional attributes which can appear, but applications and add-on frameworks (such as OSGi) are free to define additional headers.

All of the other attributes shown here are defined by OSGi, but most are optional. To create a valid bundle, only the `Bundle-SymbolicName` attribute is mandatory.

2.2. OSGi Development Tools

In theory, one does not need any tools for building OSGi bundles beyond the standard Java tools: `javac` for Java source code compilation, `jar` for packaging, and a straightforward text editor for creating the `MANIFEST.MF`.

However very few Java programmers work with such basic tools because they require lots of effort to use, both in repeatedly typing long command lines and in remembering to execute all the steps in the right sequence. In practice we use build tools like Ant or Maven, and IDEs like Eclipse, NetBeans or IntelliJ. The same is true when developing for OSGi.

Likewise, directly editing the `MANIFEST.MF` file and constructing bundles with the `jar` command is burdensome. In particular the format of the `MANIFEST.MF` file is designed primarily for efficient processing by the JVM rather than for manual editing, and therefore it has some unusual rules that make it hard to work with directly. For example, each line is limited to 72 *bytes* (not characters!) of UTF-8 encoded data[1]. Also OSGi's syntax requires some strings to be repeated many times within the same file. Again, this is done for speed of automated processing rather than for convenient editing by humans[2].

There are tools that can make the task of developing and building bundles easier and less error-prone, so in this section we will review some of the tools available.

2.2.1. Eclipse Plug-in Development Environment

Eclipse is built on OSGi. For the Eclipse platform and community to grow, it was (and still is) essential to make it easy to produce new plug-ins that extend the functionality of Eclipse. Therefore, Eclipse provides a rich set of tools for building plug-ins. Taken together these tools are called the Plug-in Development Environment, usually abbreviated to PDE, which works as a layer on top of the Java Development Tools (JDT). Both PDE and JDT are included with the "Classic" Eclipse SDK download package, but PDE is *not* included with the "Eclipse IDE for Java Developers" download.

Since Eclipse plug-ins are just OSGi bundles[3], the PDE can be used for OSGi

[1] A single UTF-8 character is represented by between one and six bytes of data.

[2] It is the author's opinion that these goals — efficient processing by machines, and ease of editing by humans — are fundamentally at odds. Formats convenient for humans to write are hard for machines to parse, and *vice versa*. XML attempts to be convenient for both, but manages to be difficult for humans to read and write while *still* being inefficient for machines to parse and generate.

[3] The term "plug-in" has always been used by Eclipse since its first version, which was not based on OSGi. Today there is no difference at all between a "plug-in" and a bundle. More details about the relationship between Eclipse and OSGi can be found in Chapter **??**

development. In fact it is one of the simplest ways to very quickly and easily create new bundles.

However, we will not use PDE for the examples in this book, for a number of reasons:

- Most importantly, using PDE would force you to use Eclipse to work through the examples. Eclipse is a great IDE and the author's first choice for Java development, but OSGi should be accessible to users of other IDEs — such as NetBeans and IntelliJ — and even to those who prefer to use Emacs or Vim[4]!
- Second, PDE is a highly interactive and graphical tool. Unfortunately this is inconvenient for pedagogical purposes, since it is difficult to describe the steps needed to complete a complex operation without filling up the book with numerous large screenshots.
- Finally, in the author's opinion PDE encourages some "anti-patterns", or practices which go against the recommended OSGi approach[5].

2.2.2. Bnd

Bnd[**?**] is a command-line tool, developed by Peter Kriens, for building OSGi bundles. Compared to PDE it may seem primitive, because it offers no GUI, instead using plain text properties files and instructions issued on the command line. However the core of the tool is very elegant and powerful, and it combines well with standard and familiar build tools such as ANT.

Listing 2.2 A Typical Bnd Descriptor File

```
# sample.bnd
Private-Package: org.example
Bundle-Activator: org.example.MyActivator
```

Listing 2.2 shows an example of a typical Bnd descriptor file. This looks very similar to a bundle manifest file, so to avoid confusion we will prefix bnd files with a comment indicating the file name, e.g. `# sample.bnd`. These files are much easier to edit than a `MANIFEST.MF` thanks to the following differences:

- As an alternative to the colon+space separator for the name/value pairs, an equals sign (=) can be used with or without surrounding spaces.
- There is no line length limit.

[4] Though I draw the line at Notepad.

[5] Specifically I feel that PDE has very weak support for the `Import-Package` form of dependency declaration, thus encouraging the use of `Require-Bundle`. The meaning of these terms is explained in Chapter 3.

- Line continuations, i.e. long strings broken over several lines to make them more readable, are indicated with a backslash character (\) at the end of the continued line.
- Shortcuts and wildcards are available in the bnd syntax for entering data that, in the underlying MANIFEST.MF syntax, would need to be repeated.

This descriptor file is used by `bnd` to generate not just the `MANIFEST.MF` but the JAR itself. The descriptor tells `bnd` both how to generate the `MANIFEST.MF` and also what the contents of the JAR should be. In this case the `Private-Package` instruction tells `bnd` that the JAR should contain the classes and resources found in the `org.example` package. It will use its classpath — which can be supplied either on the command line, or through the properties of the Ant task — to source the contents of the specified package.

2.3. Installing a Framework

In the introduction were listed four open source implementations of the OSGi standard. As was mentioned, all the sample code in this book is intended to run on all of the OSGi Release 4 implementations — i.e. Equinox, Knopflerfish and Felix, but not Concierge which is a Release 3 implementation[6] — however each framework has its own idiosyncratic way to be launched and controlled. Sadly it would be impractical to show instructions for working with all three frameworks, so we must choose one, but fortunately the differences are slight enough that almost all the instructions can still be followed on another framework simply using the cross-reference of commands in Appendix **??**.

We will mainly work with Equinox. It has a few advantages: in particular it is the "Reference Implementation" and as such tends to support new OSGi features a little earlier. Also it is, at the time of writing, slightly faster and more scalable, although this may change as all of the frameworks are continually improving in this area. Neither of these reasons is particularly strong and they are certainly not criticismsof the other frameworks. Nevertheless we must choose one and Equinox is it.

The main download page for Equinox is at:

http://download.eclipse.org/equinox

On opening this page, the first choice you are presented with is the desired version. For the purposes of following the advanced osgi-in-practice in the

[6] In fact many code samples *will* work on Concierge/OSGi R3, however in general R4 is assumed.

latter half of the book, it is best to choose the most recent released version, which at the time of writing is **3.5.1**. Next we are asked to choose how much or how little of the Equinox runtime we wish to download. The smallest useful download is 1.1Mb and it includes *only* the core Equinox runtime. However we will very soon need to use additional bundles from the Equinox distribution, and it is also useful to have source code because this enables our IDE to give us more information when we are calling or implementing APIs. Therefore the best download to choose is the "Eclipse Equinox SDK" file, which comes in at 24Mb.

Once you have downloaded this ZIP file, it can be extracted to a location of your choosing. It is a good idea to extract it to a directory named something like `Equinox-SDK-3.5.1` as this helps to distinguish it from other versions and editions of Equinox which we may download in the future. We will refer to this top-level directory henceforth as `EQUINOX_HOME` . After decompressing we will have a directory named `plugins`. In that directory we will find all the JARs (and source JARs) that implement Equinox and its supporting bundles.

2.4. Setting up Eclipse

Section 2.2.1 discussed several problems with the Eclipse Plug-in Development Environment. However, in the examples we *will* still use the Eclipse IDE, because Java development is far easier and more productive in an IDE than in a plain text editor, and because Eclipse is by far the most popular IDE for Java. But rather than using PDE, a tool which is available only in Eclipse, we will use the basic Java Development Tooling (JDT), which has direct parallels in other Java IDEs. Therefore, although the instructions given are specific to Eclipse, they can be directly translated to those other IDEs. Please check the documentation for your preferred IDE: most include a section on how to translate from Eclipse concepts.

Before creating any projects, we will first define a "User Library" for the Equinox framework, which will help us to reference Equinox from many individual projects. To do this open the system preferences dialogue by selecting `Window` → `Preferences` (Mac users: `Eclipse` → `Preferences`) and navigating the tree on the left edge to `Java` → `Build Path` → `User Libraries`. Then click the `New` button and type the library name "Equinox". Next, with the Equinox library selected, click `Add JARs` and add `plugins/org.eclipse.osgi_`*`version`*`.jar` from the Equinox directory (`EQUINOX_HOME`) (where *`version`* is the actual version tag attached to the JAR file, which depends on the version of Equinox that was downloaded).

Next you need to inform Eclipse of the location of the source code. Expand the Equinox JAR file entry in the User Library list, select "Source

attachment" and click `Edit`. Click `External File` and browse to the file `plugins/org.eclipse.osgi.source_`*`version`*`.jar` under `EQUINOX_HOME`.

You should now have something that looks like Figure 2.1.

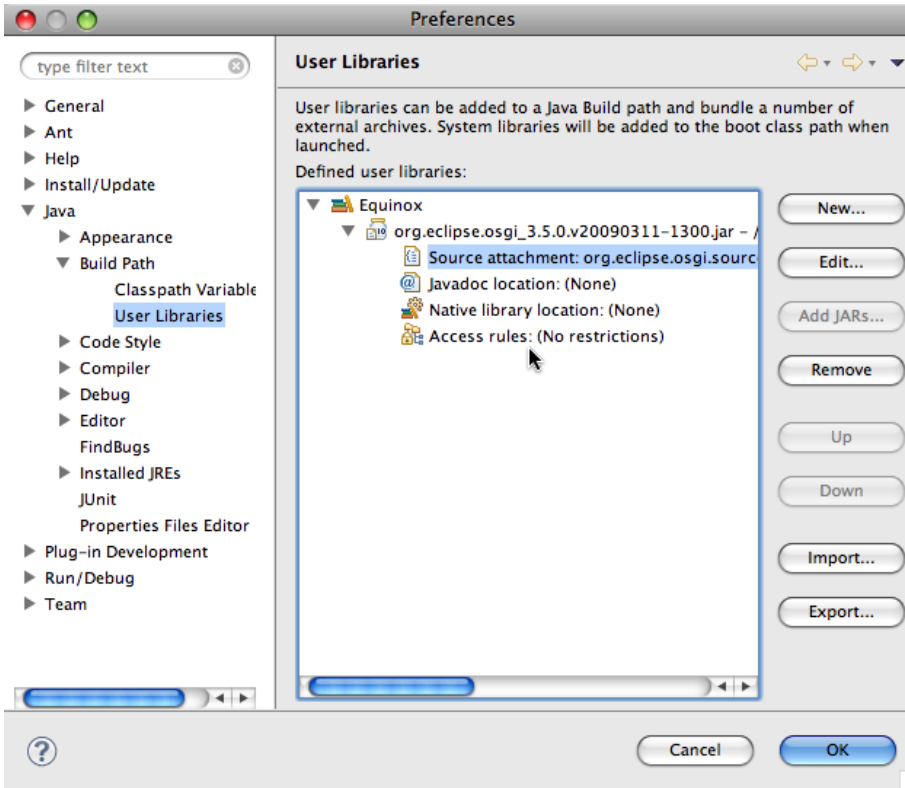


Figure 2.1.: Adding Equinox as a User Library in Eclipse

Now we can create a new Java Project by selecting `File` → `New` → `Java Project`. Eclipse will show a wizard dialogue box. Enter the project name "OSGi Tutorial" and accept all the other defaults as given. Click `Next` to go to the second page of the wizard. Here we can add the Equinox library by selecting the `Libraries` tab and clicking `Add Library`. A sub-wizard dialogue pops up: select `User Library` and click `Next`, then tick the box next to Equinox and click `Finish`. Back in the main wizard, which should now look like Figure 2.2, we can click `Finish`, and Eclipse will create the project.

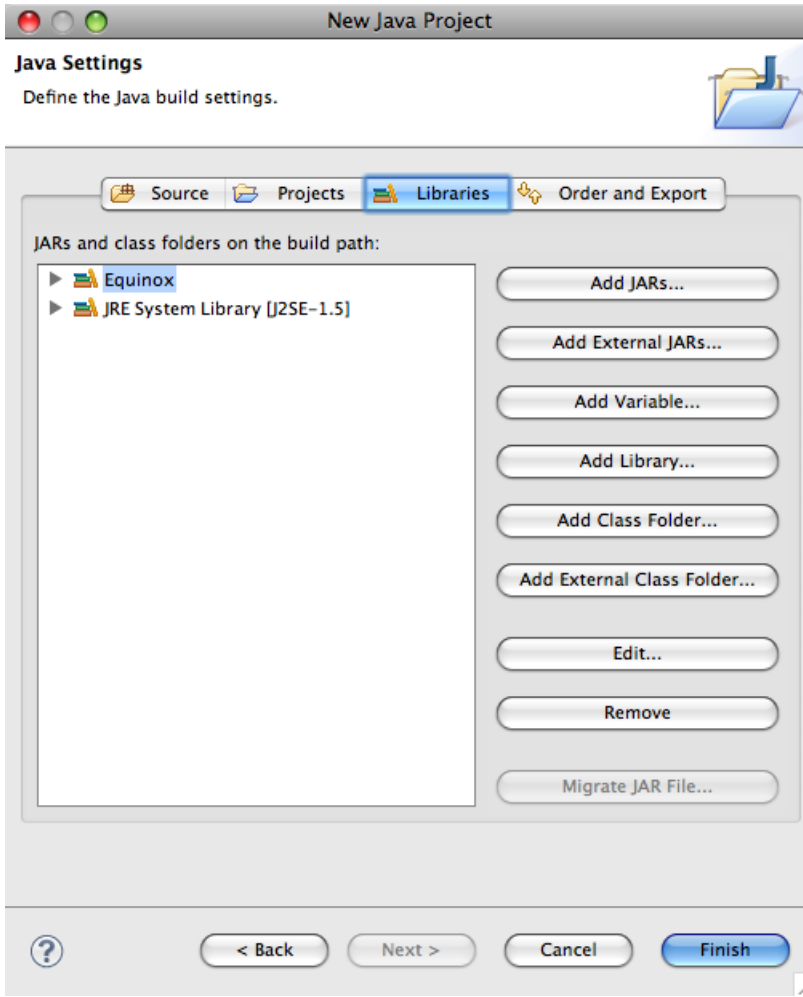


Figure 2.2.: Creating a new Java project in Eclipse: adding the Equinox library

2.5. Running Equinox

We're now going to run Equinox using the Eclipse launcher. We could run Equinox from a shell or command prompt window, but if we run under Eclipse we can easily switch to debugging mode when things go wrong.

From the main menu, select `Run` → `Run Configurations` (or `Open Run Dialog` in older Eclipse versions). On the left hand side select "Java Application" and click the "New" button. Change the Name field at the top to "Equinox". Ensure the `Project` field is set to the "OSGi Tutorial" project, and then click the `Search` button next to the `Main class` field. The only available "main" class should be the one called `EclipseStarter`, so choose that one.

Now flip to the `Arguments` tap of the Run Configurations dialog. In the `Program Arguments` field enter the following arguments exactly:

```
-console -configuration runtime
```

The first argument `-console` indicates that Equinox should create an interactive console for us to work with – if we forget this argument then it will start up and then immediately exit! The second argument (consisting of two words, `-configuration runtime`) asks Equinox to create its "configuration area" in a local directory named `runtime`. The configuration area is where Equinox stores its temporary state information.

Now click `Run`. Equinox will print the following:

```
osgi>
```

This is Equinox's standard shell prompt and indicates it is waiting for input from us. We are now ready to run some commands. The most frequently used command is `ss`, which prints the list of currently installed bundles along with their state. Let's try running `ss` now:

```
osgi> ss

Framework is launched.

id  State       Bundle
0   ACTIVE      org.eclipse.osgi_3.5.0.v20090311-1300
```

The only bundle here is Equinox itself, listed under the name of the Equinox JAR file. It is also called the "System Bundle", and it always has the bundle ID of zero.

For the moment, there is not much more we can do with Equinox until we install some more interesting bundles. However you may at this point wish to explore the commands available by typing `help`. When bored with this, don't shutdown Equinox just yet; leave it running for now.

For reference, if we wanted to run Equinox from the command line then we could do so as follows, assuming we have a shell variable named `EQUINOX_HOME` set to the installation directory we created earlier:

```
java -jar $EQUINOX_HOME/plugins/org.eclipse.osgi_*.jar
  -console -configuration runtime
```

2.6. Installing bnd

Peter Kriens' bnd is an ingenious little tool. It is packaged as a single JAR file, yet it is simultaneously:

- a standalone Java program, which can be invoked with the `java -jar` command;
- an Eclipse plug-in;
- an Ant task;
- a Maven plug-in.

We will install it as an Eclipse plug-in, but we will also be using it standalone and as an Ant task soon. First download it from:

http://www.aqute.biz/Code/Download#bnd

then take a copy and save it into the `plugins` directory underneath your Eclipse installation directory[7]. Put another copy into the project directory in Eclipse. Finally, restart Eclipse with the `-clean` command line parameter.

Next we will configure Eclipse to use its standard Java "properties" file editor to open files ending in the `.bnd` extension. To do this, open the system preferences and navigate to `General` → `Editors` → `File Associations`. Click the `Add` button next to the upper list and type `*.bnd`. Now click the `Add` button next to the lower list and select `Internal Editors` → `Properties File Editor`. Click `OK` to close the preferences window.

2.7. Hello, World!

In keeping with long-standing tradition, our first program in OSGi will be one that simply prints "Hello, World" to the console. However, most such programs immediately exit as soon as they have printed the message. We will

[7] Or the `dropins` directory if you are using Eclipse 3.4 or later.

embrace and extend the tradition with our first piece of OSGi code: since OSGi bundles have a concept of lifecycle, we will not only print "Hello" upon start-up but also "Goodbye" upon shutdown.

To do this we need to write a bundle activator. This is a class that implements the `BundleActivator` interface, one of the most important interfaces in OSGi.

Bundle activators are very simple. They have two methods, `start` and `stop`, which are called by the framework when the bundle is started and stopped respectively. Our "Hello/Goodbye, World!" bundle activator is as simple as the class `HelloWorldActivator` as shown in Listing 2.3.

Listing 2.3 Hello World Activator

```
1  package org.osgi.tutorial;

3  import org.osgi.framework.BundleActivator;
4  import org.osgi.framework.BundleContext;

6  public class HelloWorldActivator implements BundleActivator {
7      public void start(BundleContext context) throws Exception {
8          System.out.println("Hello, World!");
9      }

11     public void stop(BundleContext context) throws Exception {
12         System.out.println("Goodbye, World!");
13     }
14 }
```

There's not a lot to explain about this class, so let's just get on and build it. We need a bnd descriptor file, so create a file at the top level of the project called `helloworld.bnd`, and copy in the following contents:

Listing 2.4 Bnd Descriptor for the Hello World Activator

```
# helloworld.bnd
Private-Package: org.osgi.tutorial
Bundle-Activator: org.osgi.tutorial.HelloWorldActivator
```

This says that the bundle we want to build consists of the specified package, which is private (i.e. non-exported), and that the bundle has an activator, namely the class we just wrote.

If you installed `bnd` as an Eclipse plug-in, you can now right-click on the `helloworld.bnd` file and select `Make Bundle`. As a result, bnd will generate a bundle JAR called `helloworld.jar`. However, if you are *not* using Eclipse or the `bnd` plug-in for Eclipse, you will have to build it manually: first use the `javac` compiler to generate class files, then use the command

```
java -jar bnd.jar build -classpath classes helloworld.bnd
```

where *classes* is the directory containing the class files. Alternatively you could try the ANT build script given in Appendix A.

Now we can try installing this bundle into Equinox, which you should still have running from the previous section (if not, simply start it again). At the Equinox shell prompt, type the command:

```
install file:helloworld.jar
```

The `install` command always takes a URL as its parameter rather than a file name, hence the `file:` prefix which indicates we are using the local file URI scheme[**?**]. Strictly, this URL is invalid because it is relative rather than absolute, however Equinox allows you, in the interests of convenience, to provide a path relative to the current working directory. Therefore since the working directory is the project directory, we need only the file name in this case.

Equinox will respond with the bundle ID that it has assigned to the bundle:

```
Bundle id is 1
```

That ID will be used in subsequent commands to manipulate the bundle. It's worth noting at this point that the bundle ID you get *may* be different to what you see in this text. That's more likely to happen later on, but it's important to be aware of your actual bundle IDs and try not to copy slavishly what appears in these examples.

Let's take a quick look at the bundle list by typing `ss`. It should look like this:

```
Framework is launched.

id   State       Bundle
0    ACTIVE      org.eclipse.osgi_3.5.0.v20090311-1300
1    INSTALLED   helloworld_0.0.0
```

Now the moment of truth: start the new bundle by typing `start 1`. We should see:

```
osgi> start 1
Hello, World!
```

What does the bundle list look like now if we type `ss`? We'll ignore the rest of the listing, as it hasn't changed, and focus on our bundle:

```
1    ACTIVE      helloworld_0.0.0
```

The bundle is now in the "ACTIVE" state, which makes sense since we have explicitly started it. Now stop the bundle with `stop 1`:

```
osgi> stop 1
Goodbye, World!
```

This works as expected. What does the bundle list look like now?

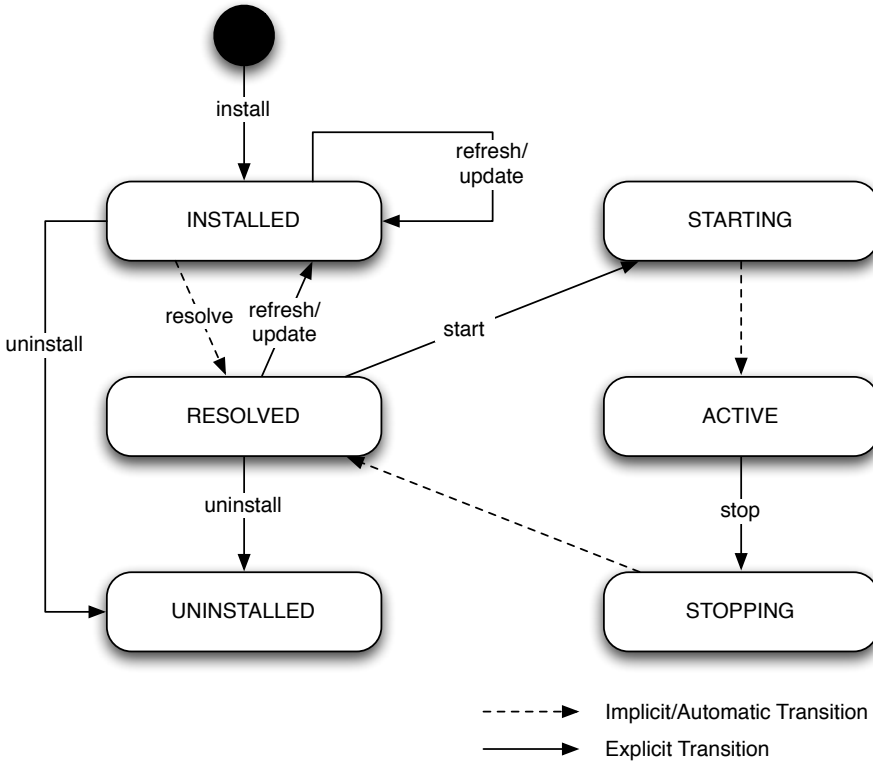


Figure 2.3.: Bundle Lifecycle

```
1   RESOLVED     helloworld_0.0.0
```

This is interesting. When we started the bundle it changed its state from Installed to Active, but when we stopped the bundle it didn't go back to Installed: instead it went to a new state "RESOLVED". What's going on here? Time to look at the lifeycle of a bundle.

2.8. Bundle Lifecycle

It was mentioned that OSGi bundles have a lifecycle, but what exactly is that lifecycle? How and why do bundles move from one state to another? Figure 2.3 shows the full lifecycle.

It's worth spending some time to trace the path through this diagram that was taken by our "Hello/Goodbye, World" bundle. Like all bundles, it started at the black circle, the entry point of the diagram, before we installed it by

executing the `install` command. At that point it entered the INSTALLED state.

Next we executed the `start` command and it appeared to transition directly to the ACTIVE state, although the diagram shows no direct link between those two states. Strictly speaking, bundles can only be started when they are in RESOLVED state, however when we attempt to start an INSTALLED bundle, the framework simply attempts to resolve it first before proceeding to start it.

RESOLVED means that the bundle's constraints have all been satisfied. In other words:

- The Java execution environment — e.g. CDC Foundation, Java 5, etc — matches or exceeds what was specified by the bundle;
- The imported packages of the bundle are available and exported with the right version range by other RESOLVED bundles, or bundles that can be RESOLVED at the same time as this bundle;
- The required bundles of the bundle are available and RESOLVED, or can be RESOLVED.

Once those constraints are satisfied, a bundle can be resolved, i.e. moved from the INSTALLED state to the RESOLVED state.

When we executed the `start` command on an INSTALLED bundle, the framework noticed that it first needed to attempt to resolve the bundle. It did so immediately, because in this case our bundle was so simple that it didn't have any constraints that needed to be satisfied. So the transition to RESOLVED state was automatic, and quickly followed by a transition to STARTING state.

STARTING simply means that the bundle is in the process of being activated. For example, the framework is currently calling the `start` method of its bundle activator. Usually the STARTING state is over in the blink of an eye, and you would not be able to see any bundles in this state on the bundle list.

Once the STARTING process is complete, the bundle reaches ACTIVE state. In this state the bundle is not necessarily actively running any code. The activity that happens during the ACTIVE state of a bundle generally depends on whatever was kicked off by the `start` method of the bundle activator.

When the `stop` command is executed, the bundle transitions to STOPPING state while the framework calls the `stop` method of it bundle activator. This is a chance for the bundle to terminate and clean up anything that it created during the STARTING phase. STOPPING is also a transient state which exists for a very short period of time before the bundle returns to RESOLVED state.

Finally we can choose to un-install the bundle, although we did not do it in the example above. Although the UNINSTALLED state is shown here, we can never see a bundle in that state, and an UNINSTALLED bundle cannot transition to any other state. Even if we reinstall the same bundle JAR file, it will be considered a different bundle by the framework, and assigned a new bundle ID.

2.9. Incremental Development

Development tends to work best as an incremental process: we prefer to make small changes and test them quickly, because we can find and correct our mistakes and misunderstandings sooner. If we write large blocks of code before testing them, there's a good chance we will have to rewrite them in full, so incremental development wastes less effort.

In some programming languages such as Scheme, development is centred around a "Read-Evaluate-Print Loop" (REPL) or interactive shell which gives us quick feedback about small changes to our code. This style of development has always been difficult in Java, as in other compiled languages, since the code must be built and deployed before it can be run, and redeployment usually implied restarting the JVM. In extreme cases such as J2EE development we might have to run a five-minute build script and then spend another five minutes restarting our application server.

OSGi can help. Although we are unlikely to ever be able to make the development cycle for Java as tight as that for Scheme, the modularity of our code and the ability to install and un-install individual bundles on the fly means that we don't need to "rebuild the world" when we make a small change, and we don't need to restart anything except the individual bundle.

Suppose we make a change to the "Hello, World!" bundle, for example we would like to print the messages in French instead of English. We can change the code as necessary and rebuild just this bundle by right-clicking on `helloworld.bnd` and selecting `Make Bundle`.

Back in the Equinox shell, we could choose to un-install the old `helloworld` bundle and install it again, but that would result in a new bundle with a new bundle ID. In fact we can simply update the existing bundle:

```
osgi> update 1

osgi> start 1
Bonjour le Monde !

osgi> stop 1
Au revoir !
```

Note that we didn't have to tell Equinox where to update the bundle from: it remembers the URL from which it was originally installed, and simply re-reads from that URL. This still works even if Equinox has been shutdown and restarted since the bundle was installed. Of course sometimes we wish to update from a new location, in which case we can pass that location as a second parameter to the `update` command:

```
osgi> update 1 file:helloworld_new.jar
```

2.10. Interacting with the Framework

Taking a look back at the code for `HelloWorldActivator`, we see that something else happens in the bundle activator. When the framework calls our activator's `start` and `stop` methods, it passes in an object of type `BundleContext`.

The bundle context is the "magic ticket" we need to interact with the OSGi framework from our bundle code. *All* interaction with the framework goes through the context, and the *only* way to access a bundle context is to implement an activator and have the framework give it to us[8].

So, what sort of things can we do with the framework via `BundleContext`? Here is an (incomplete) list:

- Look up system-wide configuration properties;
- Find another installed bundle by its ID;
- Obtain a list of all installed bundles;
- Introspect and manipulate other bundles programmatically: start them, stop them, un-install them, update them, etc;
- Install new bundles programmatically;
- Store or retrieve a file in a persistent storage area managed by the framework;
- Register and unregister bundle listeners, which tell us when the state of any bundle in the framework changes;
- Register and unregister service listeners, which tell us when the state of any service in the framework changes (services and service listeners are the subject of Chapter 4);

[8] In fact this is a small lie. There is another way to get a bundle context, which is discussed in Section 8.7 of Chapter 8, but this is an advanced technique and would only confuse matters at this stage.

- Register and unregister framework listeners, which tell us about general framework events.

Time for another example. Suppose we are *very* interested in the total number of bundles currently installed. We would like a bundle that lets us know if a bundle is installed or un-installed, along with the new total number of bundles. It should also print the total number when it starts up.

One way to approach this would be to attempt to track the running total of bundles ourselves, by first retrieving the current total when our bundle is started, and incrementing and decrementing the total as we are notified of bundles being installed and un-installed. However, that approach quickly runs into some tricky multi-threading issues. It is also unnecessary since the framework tracks all bundles anyway, so we can simply ask the framework for the current total each time we know that the total has changed. This approach is shown in Listing 2.5.

This bundle activator class is also a bundle listener: it implements both interfaces. When it starts up, it registers itself as a bundle listener and then prints the total number of bundles. Incidentally, we *must* do it that way around: if we first printed the total and then registered as a listener, there would be a potential problem, as follows. Suppose the bundle count were 10, and then a new bundle happened to be installed just before our listener started working. That is, a bundle is installed between our calls to `getBundles` and `addBundleListener`. We would get no message regarding the new bundle because we missed the event, so we would still think the bundle count is 10 when in fact it is 11. If another bundle were to be installed later, we would see the following confusing series of messages:

```
There are currently 10 bundles
Bundle installed
There are currently 12 bundles
```

By registering the listener first, we can be sure not to miss any events, but there is a price: we may get duplicate messages if a bundle is installed or un-installed between registering the listener and printing the total. In other words we may see this series of messages:

```
Bundle installed
There are currently 11 bundles
There are currently 11 bundles
```

The repetition may be annoying, but it is substantially less confusing than the apparent "jump" from ten bundles to twelve. Listing 2.6 shows the bnd descriptor that we need to build the bundle, `bundlecounter.bnd`.

To test the bundle counter, install and start it as before and then test it by un-installing and re-installing the `helloworld` bundle.

Listing 2.5 Bundle Counter Activator

```
1  package org.osgi.tutorial;
2
3  import org.osgi.framework.BundleActivator;
4  import org.osgi.framework.BundleContext;
5  import org.osgi.framework.BundleEvent;
6  import org.osgi.framework.BundleListener;
7
8  public class BundleCounterActivator implements BundleActivator,
9        BundleListener {
10
11    private BundleContext context;
12
13    public void start(BundleContext context) throws Exception {
14       this.context = context;
15
16       context.addBundleListener(this); //1
17       printBundleCount();              //2
18    }
19
20    public void stop(BundleContext context) throws Exception {
21       context.removeBundleListener(this);
22    }
23
24    public void bundleChanged(BundleEvent event) {
25       switch (event.getType()) {
26       case BundleEvent.INSTALLED:
27          System.out.println("Bundle installed");
28          printBundleCount();
29          break;
30       case BundleEvent.UNINSTALLED:
31          System.out.println("Bundle uninstalled");
32          printBundleCount();
33          break;
34       }
35    }
36
37    private void printBundleCount() {
38       int count = context.getBundles().length;
39       System.out.println("There are currently " + count + " bundles");
40    }
41 }
```

Listing 2.6 Bnd Descriptor for the Bundle Counter

```
# bundlecounter.bnd
Private-Package: org.osgi.tutorial
Bundle-Activator: org.osgi.tutorial.BundleCounterActivator
```

2.11. Starting and Stopping Threads

One of the things that we can do from the `start` method of a bundle activator is start a thread. Java makes this very easy, but the part that we must worry about is cleanly stopping our threads when the bundle is stopped.

Listing 2.7 Heartbeat Activator

```
1  package org.osgi.tutorial;
2
3  import org.osgi.framework.BundleActivator;
4  import org.osgi.framework.BundleContext;
5
6  public class HeartbeatActivator implements BundleActivator {
7
8      private Thread thread;
9
10     public void start(BundleContext context) throws Exception {
11         thread = new Thread(new Heartbeat());
12         thread.start();
13     }
14
15     public void stop(BundleContext context) throws Exception {
16         thread.interrupt();
17     }
18 }
19
20 class Heartbeat implements Runnable {
21
22     public void run() {
23         try {
24             while (!Thread.currentThread().isInterrupted()) {
25                 Thread.sleep(5000);
26                 System.out.println("I'm still here.");
27             }
28         } catch (InterruptedException e) {
29             System.out.println("I'm going now.");
30         }
31     }
32 }
```

The code in Listing 2.7 shows a simple "heartbeat" thread that prints a message every five seconds. In this case, we can use Java's interruption mechanism, which is a simple boolean status that can be set on a thread by calling the `interrupt` method. Unfortunately it is not always so easy to wake up a thread and ask it to stop, as we will see in Chapter 6.

2.12. Manipulating Bundles

The OSGi framework gives us quite a lot of programmatic control over other bundles. Let's look at an example in which we tie the lifecycle of a "target" bundle to its source file in the filesystem.

The proposed scenario is as follows: we have a bundle — our original "Hello, World!" bundle, say — that is changing frequently (we just can't decide which language those messages should be in!). Every time the bundle is rebuilt we could simply type the `update` command, but even this can be a hassle after a while. It can be easy to forget to update a bundle, and then wonder why the code you just wrote isn't working. Therefore we would like some kind of "manager" bundle that will continually monitor the `helloworld.jar` file and execute the update for us when the file changes.

The code in Listing 2.8 does exactly that. It uses a polling loop, like the heartbeat example from the last section, but on each beat it checks whether the file `helloworld.jar` is newer, according to its last-modified timestamp, than the corresponding bundle. If it is, then it updates the bundle, causing it to re-load from the file. If the bundle is up to date, or if either the bundle or the file do not exist, then it does nothing.

2.13. Exercises

1. Write a bundle that periodically checks the contents of a directory in the filesystem. Whenever a new file with the extension `.jar` appears in that directory, attempt to install it as a bundle, but only if a corresponding bundle is not already present.
2. Extend the last exercise by checking for deleted files. If a file corresponding to an installed bundle is deleted, then un-install that bundle.
3. Finally, extend your bundle to detect changes in all files in the directory that correspond to bundles, and update those bundles as necessary.

Listing 2.8 Hello Updater Activator

```
1  package org.osgi.tutorial;
2
3  import java.io.File;
4
5  import org.osgi.framework.Bundle;
6  import org.osgi.framework.BundleActivator;
7  import org.osgi.framework.BundleContext;
8  import org.osgi.framework.BundleException;
9
10 public class HelloUpdaterActivator implements BundleActivator {
11
12    private static final long INTERVAL = 5000;
13    private static final String BUNDLE = "helloworld.jar";
14
15    private final Thread thread = new Thread(new BundleUpdater());
16    private volatile BundleContext context;
17
18    public void start(BundleContext context) throws Exception {
19       this.context = context;
20       thread.start();
21    }
22
23    public void stop(BundleContext context) throws Exception {
24       thread.interrupt();
25    }
26
27    protected Bundle findBundleByLocation(String location) {
28       Bundle[] bundles = context.getBundles();
29       for (int i = 0; i < bundles.length; i++) {
30          if (bundles[i].getLocation().equals(location)) {
31             return bundles[i];
32          }
33       }
34
35       return null;
36    }
37
38    private class BundleUpdater implements Runnable {
39       public void run() {
40          try {
41             File file = new File(BUNDLE);
42             String location = "file:" + BUNDLE;
43             while (!Thread.currentThread().isInterrupted()) {
44                Thread.sleep(INTERVAL);
45                Bundle bundle = findBundleByLocation(location);
46                if (bundle != null && file.exists()) {
47                   long bundleModified = bundle.getLastModified();
48                   long fileModified = file.lastModified();
49                   if (fileModified > bundleModified) {
50                      System.out.println("File is newer, updating");
51                      bundle.update();
52                   }
53                }
54             }
55          } catch (InterruptedException e) {
56             System.out.println("I'm going now.");
57          } catch (BundleException e) {
58             System.err.println("Error updating bundle");
59             e.printStackTrace();
60          }
61       }
62    }
63 }
```

3. Bundle Dependencies

In the last chapter we created some simple bundles and showed how those bundles can interact with the framework. In this chapter, we will start to look at how bundles can interact with each other. In particular we will see how to manage dependencies between bundles.

As discussed in the Introduction, managing dependencies is the key to achieving modularity. This can be a big problem in Java — and in many other languages as well, since few provide the kind of module systems that are needed to build large applications. The default module system in Java (and it is a stretch to call it that) is the JAR-centric "classpath" model, which fails mainly because it does not manage dependencies, but instead leaves them up to chance.

What is a dependency? Simply, it is a set of assumptions made by a program or block of code about the environment in which it will run.

For example, a Java class may assume that it is running on a specific version of the Java VM, and that it has access to a specific library. It therefore depends on that Java version and that library. However those assumptions are *implicit*, meaning we don't know that they exist until the code first runs in an environment in which they are false, and an error occurs. Suppose a Java class depends on the Apache Log4J library — i.e. it assumes the Log4J JAR is available on the classpath — and then it is run in an environment where that is false (Log4J is *not* on the classpath). It will produce errors at runtime such as `ClassNotFoundException` or `NoClassDefFoundError`. Therefore we have no real idea whether the class will work in any particular environment except by trying it out, and even if it initially appears to work, it may fail at a later time when a particular execution path is followed for the first time and that path exposes a new, previously unknown, dependency.

OSGi takes away the element of chance, by managing dependencies so that they are *explicit*, *declarative* and *versioned*.

Explicit A bundle's dependencies are in the open for anybody to see, rather than hidden in a code path inside a class file, waiting to be found at runtime.

Declarative Dependencies are specified in a simple, static, textual form for easy inspection. A tool can calculate which set of bundles are required to

satisfy the dependencies of a particular bundle without actually installing or running any of them.

Versioned Libraries change over time, and it is not enough to merely depend on a library without regard to its version. OSGi therefore allows all inter-bundle dependencies to specify a version range, and even allows for multiple versions of the same bundle to be present and in use at the same time.

3.1. Introducing the Example Application

Rather than working on multiple small example pieces of code which never amount to anything interesting, for the remainder of this book we will start to build up a significant example application. The example will be based around the concept of a "message centre", for processing and displaying messages from multiple sources.

Thanks to the internet, we are today bombarded by messages of many kinds. Email is an obvious example, but also there are the blogs that we subscribe to through RSS or ATOM feeds; SMS text messages; "microblogging" sites such as Twitter[**?**] or Jaiku[**?**]; IM systems such as AOL, MSN or IRC; and perhaps for professionals in certain fields, Reuters or Bloomberg newswire feeds and market updates. Sadly we still need to flip between several different applications to view and respond to these messages. Also there is no coherent way to apply rules and automated processing to all of our inbound messages. For example, I would like a way to apply my spam filters, which do a reasonable job for my email, to my SMS text messages, as spam is increasingly a problem in that medium.

So, let's build a message centre application with two principal components:

- A graphical "reader" tool for interactively reading messages.
- An agent platform for running automated processing (e.g. filtering) over inbound message streams. This component will be designed to run either in-process with the reader tool, or as a separate server executable.

3.2. Defining an API

To support multiple kinds of message sources, we need to build an abstraction over messages and mailboxes, so a good place to start is to think about what those abstractions should look like. In Java we would represent them as interfaces. Listing 3.1 contains a reasonable attempt to define a message in the most general way.

Listing 3.1 The Message Interface

```
1  package org.osgi.book.reader.api;

3  import java.io.InputStream;

5  public interface Message {

7      /**
8       * @return The unique (within this message's mailbox) message ID.
9       */
10     long getId();

12     /**
13      * @return A human-readable text summary of the message. In some
14      *         messaging systems this would map to the "subject" field.
15      */
16     String getSummary();

18     /**
19      * @return The Internet MIME type of the message content.
20      */
21     String getMIMEType();

23     /**
24      * Access the content of the message.
25      *
26      * @throws MessageReaderException
27      */
28     InputStream getContent() throws MessageReaderException;

30 }
```

Objects implementing this interface are really just message headers. The body of the message could be of any type: text, image, video, etc. We need the header object to tell us what type the body data is, and how to access it.

Listing 3.2 The Mailbox Interface

```
1  package org.osgi.book.reader.api;
2
3  public interface Mailbox {
4
5      public static final String NAME_PROPERTY = "mailboxName";
6
7      /**
8       * Retrieve all messages available in the mailbox.
9       *
10      * @return An array of message IDs.
11      * @throws MailboxException
12      */
13     long[] getAllMessages() throws MailboxException;
14
15     /**
16      * Retrieve all messages received after the specified message.
17      *
18      * @param id The message ID.
19      * @return An array of message IDs.
20      * @throws MailboxException
21      */
22     long[] getMessagesSince(long id) throws MailboxException;
23
24     /**
25      * Mark the specified messages as read/unread on the back-end
26      * message source, where supported, e.g. IMAP supports this
27      * feature.
28      *
29      * @param read Whether the specified messages have been read.
30      * @param ids An array of message IDs.
31      * @throws MailboxException
32      */
33     void markRead(boolean read, long[] ids) throws MailboxException;
34
35     /**
36      * Retrieve the specified messages.
37      *
38      * @param ids The IDs of the messages to be retrieved.
39      * @return An array of Messages.
40      * @throws MailboxException
41      */
42     Message[] getMessages(long[] ids) throws MailboxException;
43
44 }
```

Next we need a way to retrieve messages. The interface for a mailbox could look like Listing 3.2. We need a unique identifier to refer to each message, so we assume that an ID of type `long` can be generated or assigned by the mailbox implementation. We also assume that the mailbox maintains some temporal ordering of messages, and is capable of telling us about all the new messages available given the ID of the most recent message known about by the reader tool. In this way the tool can notify us only of new messages, rather than ones that we have already read.

Many back-end message sources, such as the IMAP protocol for retrieving email, support storing the read/unread state of a message on the server, allowing that state to be synchronized across multiple clients. So, our reader tool needs to notify the mailbox when a message has been read. In other protocols where the back-end message source does not support the read/unread status, this notification can be simply ignored.

Finally we need the code for the exceptions that might be thrown. These are shown in Listing 3.3.

Listing 3.3 Mailbox API Exceptions

```
1  package org.osgi.book.reader.api;

3  public class MessageReaderException extends Exception {

5      private static final long serialVersionUID = 1L;

7      public MessageReaderException(String message) {
8          super(message);
9      }

11     public MessageReaderException(Throwable cause) {
12         super(cause);
13     }

15     public MessageReaderException(String message, Throwable cause) {
16         super(message, cause);
17     }

19 }
```

```
1  package org.osgi.book.reader.api;

3  public class MailboxException extends Exception {

5      public MailboxException(String message) {
6          super(message);
7      }

9      public MailboxException(Throwable cause) {
10         super(cause);
11     }

13     public MailboxException(String message, Throwable cause) {
14         super(message, cause);
15     }

17 }
```

3.3. Exporting the API

Now let's package up these classes into an API bundle. Create a bnd descriptor named `mailbox_api.bnd`, as shown in Listing 3.4.

Listing 3.4 Bnd Descriptor for the Mailbox API

```
# mailbox_api.bnd
Export-Package: org.osgi.book.reader.api
```

Unlike the previous bnd descriptor files, this descriptor does not have either a `Private-Package` or a `Bundle-Activator` entry. The bundle we are building here does not need to interact with the OSGi framework, it simply provides API to other bundles, so we need the `Export-Package` directive, which instructs bnd to do *two* separate things:

- It ensures that the contents of the named packages are included in the output bundle JAR.
- It adds an `Export-Package` header to the `MANIFEST.MF` of the JAR.

From this descriptor, bnd will generate `mailbox_api.jar`, so let's take a look inside the JAR. On Windows you can use a tool such as WinZip[?] to peek inside; on UNIX platforms including Linux and Mac OS X there is a wider variety of tools, but the command-line tool `unzip` should be available on nearly all of them.

The file content listing of the JAR should be no surprise: we simply have the four API classes (two interfaces, two exceptions) in the normal Java package directory tree:

```
$ unzip -l mailbox_api.jar
Archive:  mailbox_api.jar
  Length      Name
 --------     ----
      302     META-INF/MANIFEST.MF
        0     org/
        0     org/osgi/
        0     org/osgi/book/
        0     org/osgi/book/reader/
        0     org/osgi/book/reader/api/
      379     org/osgi/book/reader/api/Mailbox.class
      677     org/osgi/book/reader/api/MailboxException.class
      331     org/osgi/book/reader/api/Message.class
      759     org/osgi/book/reader/api/MessageReaderException.class
 --------     -------
     2448     10 files
```

Let's take a look at the generated manifest. WinZip users can right click on `MANIFEST.MF` and select "View with Internal Viewer". UNIX users can run `unzip` with the `-p` ("pipe") switch:

```
$ unzip -p mailbox_api.jar META-INF/MANIFEST.MF
Manifest-Version: 1.0
Bundle-Name: mailbox_api
Created-By: 1.5.0_13 (Apple Inc.)
Import-Package: org.osgi.book.reader.api
Bundle-ManifestVersion: 2
Bundle-SymbolicName: mailbox_api
Tool: Bnd-0.0.223
```

```
Bnd-LastModified: 1198796713427
Export-Package: org.osgi.book.reader.api
Bundle-Version: 0
```

Here we see the `Export-Package` header, along with a few other headers which have been added by bnd. This header tells OSGi to make the named list of packages (in this case only one package) available to be imported by other bundles. We will take a look at how to import packages shortly.

As we saw in Section 1.2.5, plain Java makes all public classes in all packages of a JAR available to clients of that JAR, making it impossible to hide implementation details. OSGi uses the `Export-Package` header to fix that problem. In fact the default is reversed: in OSGi, all packages are hidden from clients unless they are explicitly named in the `Export-Package` header.

In bnd the general form of the `Export-Package` directive is a comma-separated list of patterns, where each pattern is either a literal package name, or a "glob" (wildcard) pattern, or a negation introduced with the `!` character. For example the following directive instructs bnd to include every package it can find on the classpath except for those starting with "`com.`":

```
Export-Package: !com.*, *
```

3.4. Importing the API

Now let's write some code that depends on the API: an implementation of a mailbox and message types. For the sake of simplicity at this stage, this will be a mailbox that holds only a fixed number of hard-coded messages. The code will live in a new package, `org.osgi.book.reader.fixedmailbox`. First we write an implementation of the `Message` interface, giving us the class `StringMessage` in Listing 3.5, and next the `Mailbox` interface implementation, giving us `FixedMailbox`[1] in Listing 3.6.

Now, what should the bnd descriptor look like? In the last section we used the `Export-Package` directive to include the specified package in the bundle, and also export it. This time, we want to include our new package in the bundle but we don't want to export it, so we use the `Private-Package` instruction:

```
# fixed_mailbox.bnd
Private-Package: org.osgi.book.reader.fixedmailbox
```

Unlike `Export-Package`, the `Private-Package` directive does not correspond to a recognized `MANIFEST.MF` attribute in OSGi. It is merely an instruction to bnd that causes the specified packages to be included in the bundle JAR —

[1] You may be curious at this stage why methods of `FixedMailbox` are declared as `synchronized`. This is required to keep the implementation thread-safe when we extend it later, in Chapter 7

Listing 3.5 String Message

```
1  package org.osgi.book.reader.fixedmailbox;
2
3  import java.io.ByteArrayInputStream;
4  import java.io.InputStream;
5
6  import org.osgi.book.reader.api.Message;
7  import org.osgi.book.reader.api.MessageReaderException;
8
9  public class StringMessage implements Message {
10
11    private static final String MIME_TYPE_TEXT = "text/plain";
12
13    private final long id;
14    private final String subject;
15    private final String text;
16
17    public StringMessage(long id, String subject, String text) {
18       this.id = id;
19       this.subject = subject;
20       this.text = text;
21    }
22
23    public InputStream getContent() throws MessageReaderException {
24       return new ByteArrayInputStream(text.getBytes());
25    }
26
27    public long getId() {
28       return id;
29    }
30
31    public String getMIMEType() {
32       return MIME_TYPE_TEXT;
33    }
34
35    public String getSummary() {
36       return subject;
37    }
38 }
```

Listing 3.6 Fixed Mailbox

```
1  package org.osgi.book.reader.fixedmailbox;
2
3  import java.util.ArrayList;
4  import java.util.List;
5
6  import org.osgi.book.reader.api.Mailbox;
7  import org.osgi.book.reader.api.MailboxException;
8  import org.osgi.book.reader.api.Message;
9
10 public class FixedMailbox implements Mailbox {
11
12    protected final List<Message> messages;
13
14    public FixedMailbox() {
15       messages = new ArrayList<Message>(2);
16       messages.add(new StringMessage(0, "Hello", "Welcome to OSGi"));
17       messages.add(new StringMessage(1, "Getting Started",
18             "To learn about OSGi, read my book."));
19    }
20
21    public synchronized long[] getAllMessages() {
22       long[] ids = new long[messages.size()];
23       for (int i = 0; i < ids.length; i++) {
24          ids[i] = i;
25       }
26       return ids;
27    }
28
29    public synchronized Message[] getMessages(long[] ids)
30          throws MailboxException {
31       Message[] result = new Message[ids.length];
32       for (int i = 0; i < ids.length; i++) {
33          long id = ids[i];
34          if (id < 0 || id >= messages.size()) {
35             throw new MailboxException("Invalid message ID: " + id);
36          }
37          result[i] = messages.get((int) id);
38       }
39       return result;
40    }
41
42    public synchronized long[] getMessagesSince(long id)
43          throws MailboxException {
44       int first = (int) (id + 1);
45       if (first < 0) {
46          throw new MailboxException("Invalid message ID: " + first);
47       }
48       int length = Math.max(0, messages.size() - first);
49       long[] ids = new long[length];
50       for (int i = 0; i < length; i++) {
51          ids[i] = i + first;
52       }
53       return ids;
54    }
55
56    public void markRead(boolean read, long[] ids) {
57       // Ignore
58    }
59 }
```

you can verify this by looking at the contents of the JAR. Listing 3.7 shows what the generated `MANIFEST.MF` should look like.

Listing 3.7 `MANIFEST.MF` generated from `fixed_mailbox.bnd`

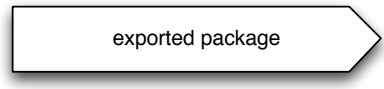
```
Manifest-Version: 1.0
Bundle-Name: fixed_mailbox
Created-By: 1.5.0_13 (Apple Inc.)
Private-Package: org.osgi.book.reader.fixedmailbox
Import-Package: org.osgi.book.reader.api
Bundle-ManifestVersion: 2
Bundle-SymbolicName: fixed_mailbox
Tool: Bnd-0.0.223
Bnd-LastModified: 1198893954164
Bundle-Version: 0
```

The `Private-Package` header does appear here, because bnd copies it, but it will be ignored by the OSGi framework. One of the other headers inserted by bnd is very important though: the `Import-Package` header.

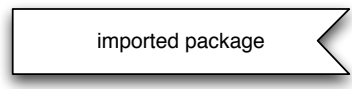
Just as packages are not made available from bundles except when explicitly exported using the `Export-Package` header, bundles cannot use the classes from a package unless they have explicitly imported that package using the `Import-Package` header (or the `Require-Bundle` header, which we will see shortly). Since the bundle we are building currently has code-level dependency on the package `org.osgi.book.reader.api`, it must import that package.

All packages that we use in the code for a bundle *must* be imported, except for packages beginning with `java.*`, which must *not* be imported. The `java.*` packages, and only those packages, are always available without being imported. There is a common misconception that packages in the standard Java runtime APIs do not need to be imported. *They do.* For example, if a bundle uses Swing, it must import `javax.swing`, but it need not import `java.awt`. If a bundle uses SAX parsing for XML documents, it must import `org.xml.sax`.

If we represent an exported package diagrammatically as follows:



and an imported package as follows:



then the runtime resolution of the two bundles will be as in Figure 3.1. The thick, dashed line in this figure represents the "wiring" together by the framework of the import with its corresponding export.

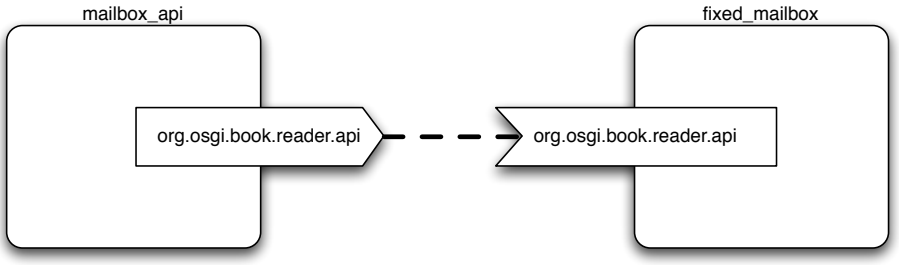


Figure 3.1.: The runtime resolution of matching import and export.

A package import is one of the *constraints* that the framework must satisfy in order to resolve a bundle, i.e. move it from the INSTALLED state to the RESOLVED state. In this case the constraint on the `fixed_mailbox` bundle was satisfied by matching it with the corresponding export on the `mailbox_api` bunde. In general the framework must match all of the imported packages against packages exported from RESOLVED bundles before it can satisfy the import constraints of a bundle. Note that the framework can move two or more bundles into RESOLVED state simultaneously, making it possible to resolve bundles that have circular dependencies.

3.5. Interlude: How Bnd Works

In the previous example, the bnd descriptor for the mailbox implementation bundle contained only a single line — the `Private-Package` instruction — yet the generated manifest contained a correct `Import-Package` statement. How did that happen?

In fact this is one of the main features of bnd: it is able to calculate the imports of a bundle through careful inspection of the class files that we tell it to include in the bundle. Since we are making so much use of bnd, we need to take a closer look at how it generates bundles and manifests.

Bnd has several modes in which it can be used, but we will mostly be using it in its "build" mode for constructing bundles. In build mode, bnd follows a two-stage process: first it works out what the contents of the bundle should be, i.e. which packages and classes need to be included; and second, it calculates the dependencies of the included classes and generates an `Import-Package` statement.

The `Export-Package` and `Private-Package` instructions determine the contents of the bundle. Any package named in either of these instructions, either explicitly or through a wildcard pattern, will be included in the bundle JAR. Listing 3.8 shows an example.

Listing 3.8 Bnd Sample: Controlling Bundle Contents

```
# bnd sample
Export-Package: org.osgi.book.reader*
Private-Package: org.osgi.tutorial
```

This will result in all packages beginning with `org.osgi.book.reader` being included in the bundle and exported; and the package `org.osgi.tutorial` included but not exported.

Once bnd knows what should be in the JAR, it can calculate the imports. As mentioned, it inspects the bytecode of each class file found in the included packages, which yields a list of classes and packages. By default, every package found is added to the `Import-Package` header, but we can assert more control over this process by including an `Import-Package` *instruction* in our bnd file. For example, Listing 3.9 shows how to apply a specific version range to some imports and mark others as optional.

Listing 3.9 Bnd Sample: Controlling Imports

```
# bnd sample
Import-Package: org.apache.log4j*;version="[1.2.0,1.3.0)",\
                javax.swing*;resolution:=optional,\
                *
```

The entries in this list are *patterns*. Each package dependency found through bytecode analysis is tested against the patterns here in order, and if a match is found then the additional attributes are applied to the import. In this sample, if a dependency on a Log4J package is found then it will be marked with the version attribute specified; if a dependency on any Swing package is found then it will be marked as optional. The final "*" is a catch-all; any packages matching neither of the first two patterns will pass straight into the manifest without any additional attributes.

The use of wildcards like this can be initially alarming if one does not understand the way that bnd checks the actual discovered dependencies against the patterns in the `Import-Package` statement. For example the pattern `javax.swing*` may appear to be importing the whole of Swing; i.e. all 17 packages of it. If that were the case then the simple `*` on its own would be even more alarming! The confusion arises because bnd's instructions, such as

`Import-Package`, have exactly the same name as OSGi manifest headers, but they are treated quite differently.

We can also explicitly add and remove packages from the imports:

```
# bnd sample
Import-Package: !org.foo,\
                com.bar,\
                *
```

This results in `org.foo` being excluded from the imports list, even if bnd detects a dependency on it. This is dangerous, but useful if the dependency exists only in a part of the code that we know can never be reached. The second entry inserts the package `com.bar` into the imports list, whether or not bnd is able to detect a dependency on it. This is mainly useful in cases where the bundle code uses reflection to dynamically load classes. Note that only fully named packages can be used: if we wrote `com.bar*`, that would be a pattern, and only included if a package starting with `com.bar` were to be found in bytecode analysis.

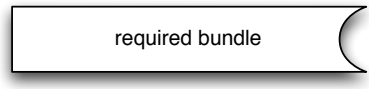
Note that the final asterisk is very important. If you omit it then *only* the dependencies listed explicitly will be included in the generated manifest. Any others that bnd finds will be excluded, which is generally not what one wants to achieve.

3.6. Requiring a Bundle

Another kind of constraint that can be placed on a bundle is the `Require-Bundle` header. This is similar to `Import-Package` header in that it makes exported packages from another bundle available to our bundle, but it works on the whole bundle rather than individual packages:

```
Require-Bundle: mailbox-api
```

If we represent a required bundle as follows:



Then the runtime resolution of a bundle using `Required-Bundle` looks like Figure 3.2. In this figure, Bundle B has a `Require-Bundle` dependency on Bundle A, meaning that Bundle B cannot resolve unless Bundle A is in RESOLVED state.

The effect at runtime is as if Bundle B had declared an `Import-Package` header naming every package exported by Bundle A. However, Bundle B is giving up

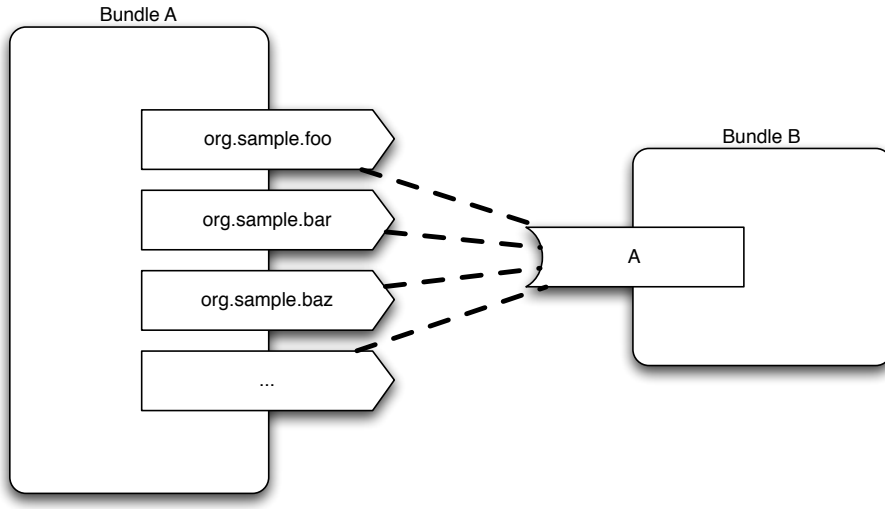


Figure 3.2.: The runtime resolution of a Required Bundle

a lot of control, because the list of imports is determined by the set of packages exported by Bundle A. If additional exports are added to Bundle A, then they are automatically added as imports to Bundle B.

The use of `Require-Bundle` is strongly discouraged by most OSGi practitioners except where absolutely necessary, because there are several flaws with this kind of dependency scheme.

First, a bundle using `Require-Bundle` to import code from another bundle is at the mercy of what is provided by that bundle — something that can change over time. We really have no idea what packages will be provided by another bundle at any point in the future, yet nevertheless our bundle will successfully resolve even if the required bundle stops exporting some functionality that we rely on. The result will almost certainly be class loading errors such as `ClassNotFoundException` or `NoClassDefFoundError` arising in bundles that were previously working.

The second and closely related problem is that requiring bundles limits our ability to refactor the composition of those bundles. Suppose at some point we notice that Bundle A has grown too big, and some of the functionality it provides is really unrelated to the core and should be separated into a new bundle, which we will call Bundle A'. As a result, some of the exports of A move into A'. For any consumers of that functionality who are using purely `Import-Package`, the refactoring of A into A and A' will be entirely transparent: the imports will simply be wired differently at runtime by the framework. Figures 3.3 and 3.4 show the "before" and "after" states of per-

forming this refactoring where Bundle B depends on the packages of Bundle A using `Import-Package`. After refactoring, Bundle B will continue to work correctly because it will still import all the packages it needs – and it will be oblivious of the fact that they now come from two bundles rather than one.

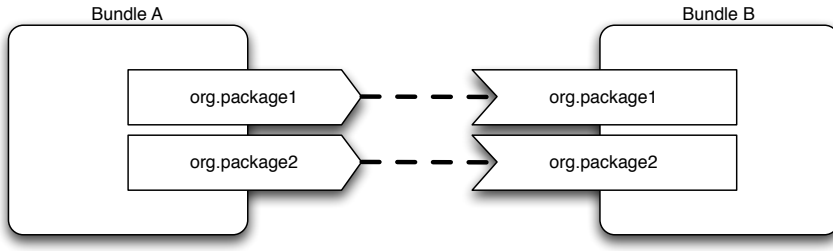


Figure 3.3.: Refactoring with `Import-Package`: (*Before*)

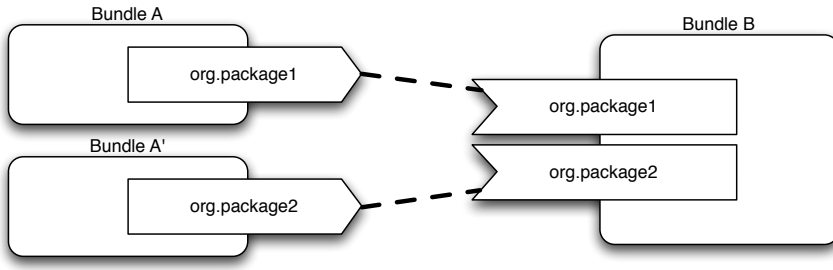


Figure 3.4.: Refactoring with `Import-Package`: (*After*)

However, Figures 3.5 and 3.6 show the "before" and "after" states when Bundle B *requires* Bundle A using `Require-Bundle`. After refactoring, Bundle B will no longer import one of the packages it needs, but it will still be able to enter the RESOLVED state because the `Require-Bundle` constraint is still satisfied. Therefore we are likely to get `NoClassDefFoundErrors` when B attempts to use classes from `org.package2`.

Third, whole-module dependency systems tend to cause a high degree of "fan-out". When a bundle requires another bundle, we have no idea (except by low-level examination of the code) which part of the required bundle it is actually using. This can result in us bringing in a large amount of functionality when only a small amount is really required. The required bundle may also require several other bundles, and those bundles require yet more bundles, and so on. In this way we can easily be required to pull in fifty or more bundles just to resolve a single, small bundle. Using purely `Import-Package` gives us the opportunity to break this fan-out by finding instances where only a small portion of a large bundle is used, and either splitting that bundle or finding

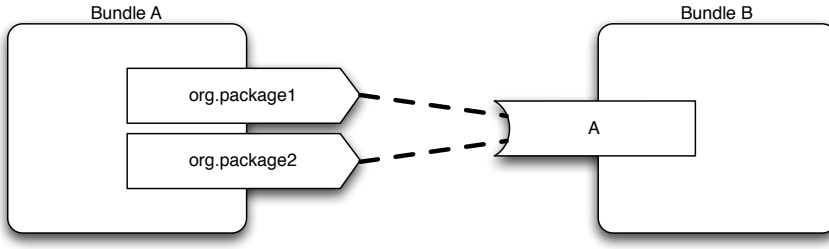


Figure 3.5.: Refactoring with `Require-Bundle`: (*Before*)

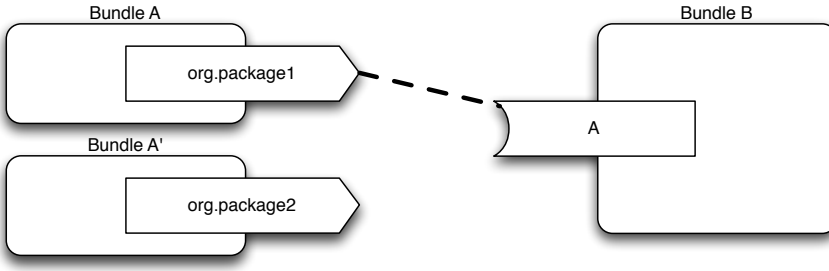


Figure 3.6.: Refactoring with `Require-Bundle`: (*After*)

an alternative supplier for the functionality we need.

Essentially, using `Require-Bundle` is like grabbing hold of the wrapper around what we need, rather than the contents itself. It might actually work if JARs in Java were more cohesive, and the constituent packages did not change over time, but that is not the case.

`Require-Bundle` was only recently introduced into OSGi in Release 4, and given all the problems listed, it may seem mysterious that it was introduced at all. The main reason was to support various legacy issues in Eclipse, which abandoned an earlier module system in favour of OSGi. The pre-OSGi module system used by Eclipse was based on whole-module dependencies, and if OSGi had offered only `Import-Package` then the challenges of making thousands of existing Eclipse plug-ins work as OSGi bundles would have been insurmountable. Nevertheless, the existence of `Require-Bundle` remains highly controversial, and there are very, very few good reasons ever to use it in new bundles.

3.7. Version Numbers and Ranges

Versioning is a critically important part of OSGi. Modules, or libraries, are not immortal and unchanging entities — they evolve as their requirements change and as new features are added.

To take the example of Apache Log4J again, the most widely used version is 1.2. Earlier versions than this are obsolete, and there were also significant changes in the API between 1.1 and 1.2, so any code that has been written to use version 1.2 will almost certainly not work on 1.1 or earlier. There is also a 1.3 version, but it introduced a number of incompatibilities with 1.2 and is now discontinued — it is considered a failed experiment, and never produced a "stable" release. Finally there is a new version 2.0, but it is still experimental and not widely used.

This illustrates that we cannot simply use a bundle without caring about which version we wish to use. Therefore OSGi provides features which allow us first to describe in a consistent way the versions of all our bundles and their exports, and second to allow bundles to describe the range of versions that are acceptable for each of their dependencies.

3.7.1. Version Numbers

OSGi follows a consistent scheme for all version numbers. It uses three numeric segments plus one alphanumeric segment, where any segment may be omitted. For example the following are all valid versions in OSGi:

- `1`
- `1.2`
- `1.2.3`
- `1.2.3.beta_3`

The three numeric segments are known as the *major*, *minor* and *micro* numbers and the final alphanumeric segment is known as the *qualifier*. When any one of the numeric segments is missing, it takes the implicit value of zero, so `1` is equivalent to `1.0` and `1.0.0`. When a version string is not supplied at all, the version `0.0.0` is implied.

Versions have a total ordering, using a simple algorithm which descends from the major version to the qualifier. The first segment with a difference in value between two versions "short-circuits" the comparison, so later segments need not be compared. For example `2.0.0` is considered higher than `1.999.999`, and this is decided without even looking at the minor or micro levels.

Things get interesting when we consider the qualifier, which may contain letters A-Z in upper or lower case, numbers, hyphens (-) and underscores (_). The qualifier is compared lexicographically using the algorithm found in the `compareTo()` method of the standard Java String class. There are two points to beware of: qualifier strings are compared character by character until a difference is found, and shorter strings are considered lower in value than longer strings.

Suppose you are getting ready to release version 2.0 of a library. This is a significant new version and you want to get it right, so you go through a series of "alpha" and "beta" releases. The alphas are numbered as follows:

- `2.0.0.alpha1`
- `2.0.0.alpha2`
- ...

This works fine up until the ninth alpha, but then when you release version `2.0.0.alpha10` you find that it doesn't appear to be the highest version! This is because the number ten starts with the digit 1, which comes before the digit 2. So version `alpha10` will actually come between versions `alpha1` and `alpha2`. Therefore, if you need to use a number component inside the qualifier, always be sure to include some leading zeros. Assuming that 99 alpha releases are enough, we should have started with `alpha01`, `alpha02` and so on.

Finally after lots of testing, you are ready to unleash the final release version, which you call simply `2.0.0`. Sadly this doesn't work either, as it comes before *all* of the alpha and beta releases. The qualifier is now the empty string, which comes before all non-empty strings. So we need to add a qualifier such as `final` to the final release version.

Another approach that can work is to always add a date-based qualifier, which can be generated as part of the build process. The date would need to be written "backwards" — i.e. year number first, then month, then day and perhaps time — to ensure the lexicographic ordering matches the temporal order. For example:

- `2.0.0.2008-04-28_1230`

This approach scales well, so it is especially useful for projects that release frequently. For example, Eclipse follows a very similar scheme to this. However this approach can be inconvenient because the version strings are so verbose, and it's difficult to tell which versions are important releases versus mere development snapshots.

3.7.2. Versioning Bundles

A version number can be given to a bundle by supplying the `Bundle-Version` manifest header:

```
1 Bundle-Version: 1.2.3.alpha
```

Bundle-level versioning is important because of `Require-Bundle`, and also because OSGi allows multiple versions of the same bundle to be present in the framework simultaneously. A bundle can be uniquely identified by the combination of its `Bundle-SymbolicName` and its `Bundle-Version`.

3.7.3. Versioning Packages

However, bundle-level versioning is not enough. The recommended way to describe dependencies is with `Import-Package`, and a single bundle could contain implementations of multiple different APIs. Therefore we need version information at the package level as well. OSGi allows us to do this by tagging each exported package with a version attribute, as follows:

```
Export-Package: org.osgi.book.reader.api;version="1.2.3.alpha",
  org.osgi.book.reader.util;version="1.2.3.alpha"
```

Unfortunately in the manifest file, the version attributes must be added to each exported package individually. However bnd gives us a shortcut:

```
# bnd sample
Export-Package: org.osgi.book.reader*;version="1.2.3.alpha"
```

We can also, if we wish to, keep the bundle version and the package export versions synchronized:

```
# bnd sample
ver: 1.2.3.alpha
Bundle-Version: ${ver}
Export-Package: org.osgi.book.reader*;version=${ver}
```

3.7.4. Version Ranges

When we import a package or require a bundle, it would be too restrictive to only target a single specific version. Instead we need to specify a range.

Recall the discussion of Apache Log4J and its various versions. Suppose we wish to depend on the stable release, version 1.2. However, "1.2" is not just a single version, it is in fact a range from 1.2.0 through to 1.2.15, the latest at the time of writing, meaning there have been fifteen "point" releases since the main 1.2 release. However, as in most projects, Log4J tries to avoid changes in the API between point releases, instead confining itself to bug fixes and

perhaps minor API tweaks that will not break backwards compatibility for clients. Therefore it is likely that our code will work with any of the 1.2 point releases, so we describe our dependency on 1.2 using a version *range*.

A range is expressed as a floor and a ceiling, enclosed on each side either by a bracket "[" or a parenthesis "(". A bracket indicates that the range is inclusive of the floor or ceiling value, whereas a parenthesis indicates it is exclusive. For example:

- `[1.0.0,2.0.0)`

This range includes the value `1.0.0`, because of the opening bracket. but excludes the value `2.0.0` because of the closing parenthesis. Informally we could write it as "1.x".

If we write a single version number where a range is expected, the framework still interprets this as a range but with a ceiling of infinity. In other words `1.0.0` would be interpreted as the range `[1.0.0,`∞`)`. To specify a single exact version, we have to write it twice, as follows: `[1.2.3,1.2.3]`.

For reference, here are some further examples of ranges, and what they mean when compared to an arbitrary version x:

`[1.2.3,4.5.6)`	$1.2.3 \leq x < 4.5.6$
`[1.2.3,4.5.6]`	$1.2.3 \leq x \leq 4.5.6$
`(1.2.3,4.5.6)`	$1.2.3 < x < 4.5.6$
`(1.2.3,4.5.6]`	$1.2.3 < x \leq 4.5.6$
`1.2.3`	$1.2.3 \leq x$
	$0.0.0 \leq x$

In our Log4J example and in real world usage, the first of these styles is most useful. By specifying the range `[1.2,1.3)` we can match any of the `1.2.`x point releases. However, this may be a leap of faith: such a range would match all future versions in the 1.2 series, e.g. version `1.2.999` (if it were ever written) and beyond. We cannot test against all future versions of a library, so we must simply trust the developers of the library not to introduce breaking changes into a future point release. If we don't or can't trust those developers, then we must specify a range which includes only the known versions, such as `[1.2,1.2.15]`.

In general it is probably best to go with the open-ended version range in most cases. The cost in terms of lost flexibility with the more conservative closed range outweighs the risk of breaking changes in future versions.

3.7.5. Versioning `Import-Package` and `Require-Bundle`

We can add version ranges to the `Import-Package` statement in exactly the same way as we added a version to `Export-Package`, using an attribute:

```
Import-Package: org.apache.log4j;version="[1.2,1.3)",
  org.apache.log4j.config;version="[1.2,1.3)"
```

Again, bnd can help to reduce the verbosity:

```
# bnd
Import-Package: org.apache.log4j*;version="[1.2,1.3)"
```

We can also add a version range when requiring a bundle, but the attribute name is slightly different:

```
Require-Bundle: mailbox-api;bundle-version="[1.0.0,1.1.0)"
```

It's always possible that a package import could match against two or more exports from different bundles, or a required bundle could match two or more bundles. In that case, the framework chooses the provider with the highest version, so long as that version is in the range specified by the consumer. Unfortunately, even this rule sometimes fails to come up with a single winner, because two bundles can export the same version of a package. When that happens the framework arbitrarily chooses the one with the lowest bundle ID, which tends to map to whichever was installed first.

3.8. Class Loading in OSGi

In Section 1.2.1 in the Introduction, we saw how normal hierarchical class loading works in Java. Figure 3.7 shows, in slightly simplified form, the process by which classes are searched in OSGi.

All resolved bundles have a class loader, and when that class loader needs to load a class it first checks whether the package name begins with `java.*` or is listed in a special configuration property, `org.osgi.framework.bootdelegation`. If that's the case, then the bundle class loader immediately delegates to its parent class loader, which is usually the "application" class loader, familiar from traditional Java class loading.

Why does OSGi include this element of hierarchical class loading, if it is supposed to be based on a graph of dependencies? There are two reasons:

- The only class loader permitted to define classes for the `java.*` packages is the system bootstrap class loader. This rule is enforced by the Java Virtual Machine, and if any other class loader (e.g. an OSGi bundle class loader) attempts to define one of these classes it will receive a `SecurityException`.

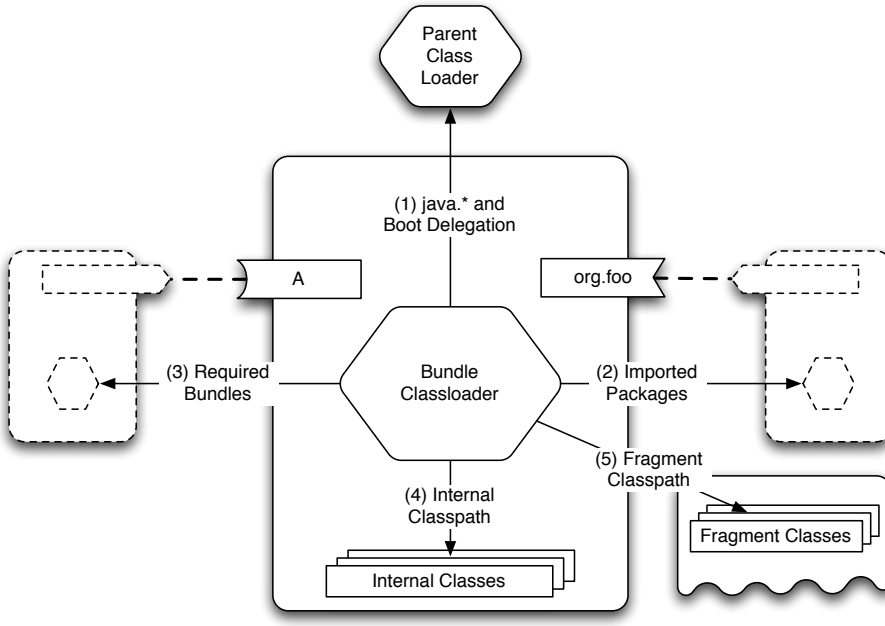


Figure 3.7.: Simplified OSGi Class Search Order

- Unfortunately some Java Virtual Machines, including most versions of Sun's VM and OpenJDK, rely on the incorrect assumption that parent delegation always occurs. Because of this assumption, some internal VM classes expect to be able to find certain other internal classes through any arbitrary class loader. Therefore OSGi provides the `org.osgi.framework.bootdelegation` property to allow for parent delegation to occur for a limited set of packages, i.e. those providing the internal VM classes.

The second step that the bundle class loader takes when searching for a class is to check whether it is in a package imported with `Import-Package`. If so then it "follows the wire" to the bundle exporting the package and delegates the class loading request to that bundle's class loader. The exporting bundle's class loader will run the same procedure, and as a result may itself delegate to another bundle's class loader. This means it is quite valid for a bundle to re-export a package that it itself imports.

The third step is to check whether the class is in a package imported using `Require-Bundle`. If it is then it the class loading request is delegated to the required bundle's class loader.

Fourth, the class loader checks the bundle's own internal classes, i.e. the classes inside its JAR.

Fifth, the class loader searches the internal classes of any *fragments* that might be currently attached to the bundle. Fragments will be discussed in Section 3.11.

There is a sixth step that is omitted by Figure 3.7, which is related to dynamic class loading. This is an advanced topic that will be discussed in Chapter **??**.

Figure 3.8 shows the full class search order in the form of a flowchart. This diagram is derived from Figure 3.18 in the OSGi R4.1 Core Specification.

The class search algorithm as described always attempts to load classes from another bundle in preference to classes that may be on the classpath of the present bundle. This may seem counterintuitive at first, but in fact it makes a lot of sense, and it fits perfectly with the conventional Java approach. A traditional Java class loader always first delegates to its parent before attempting to define a class itself. This helps to ensure that classes are loaded as few times as possible, by favouring already-defined copies of a class over redefining it in a lower-level class loader. The opposite approach would result in many copies of the same class, which would be considered incompatible because they were defined by different class loaders. Back in OSGi, where the class loaders are arranged in a graph rather than a tree, the same principle of minimising class loading translates to making every effort to find a class from a bundle's imports rather than loading it from the internal bundle classpath.

Remember, the identity of a class is defined by the combination of its fully qualified name *and* the class loader which loaded it, so class `org.foo.Bar` loaded by class loader A is considered different from `org.foo.Bar` loaded by class loader B, even if they were loaded from the same physical bytes on disk. When this happens the result can be very confusing, for example a `ClassCastException` being thrown on assignment of a value of type `org.foo.Bar` to a variable of type `org.foo.Bar`!

3.9. JRE Packages

In Section 3.4 we saw that it is necessary to always import all packages used in the bundle except for `java.*`. Therefore all of the other packages in the base JRE libraries, e.g. those beginning `javax.*`, `org.omg.*`, `org.w3c.*` etc must all be imported if used.

As ordinary imports, these packages are not subject to parent delegation, so they must be supplied in the normal way by wiring the import to a bundle that provides them as an export. However, you do not need to create that bundle yourself: the system bundle performs this task.

Recall from Section **??** that the system bundle is always present, and it represents the framework itself as a bundle. One of the jobs performed by the

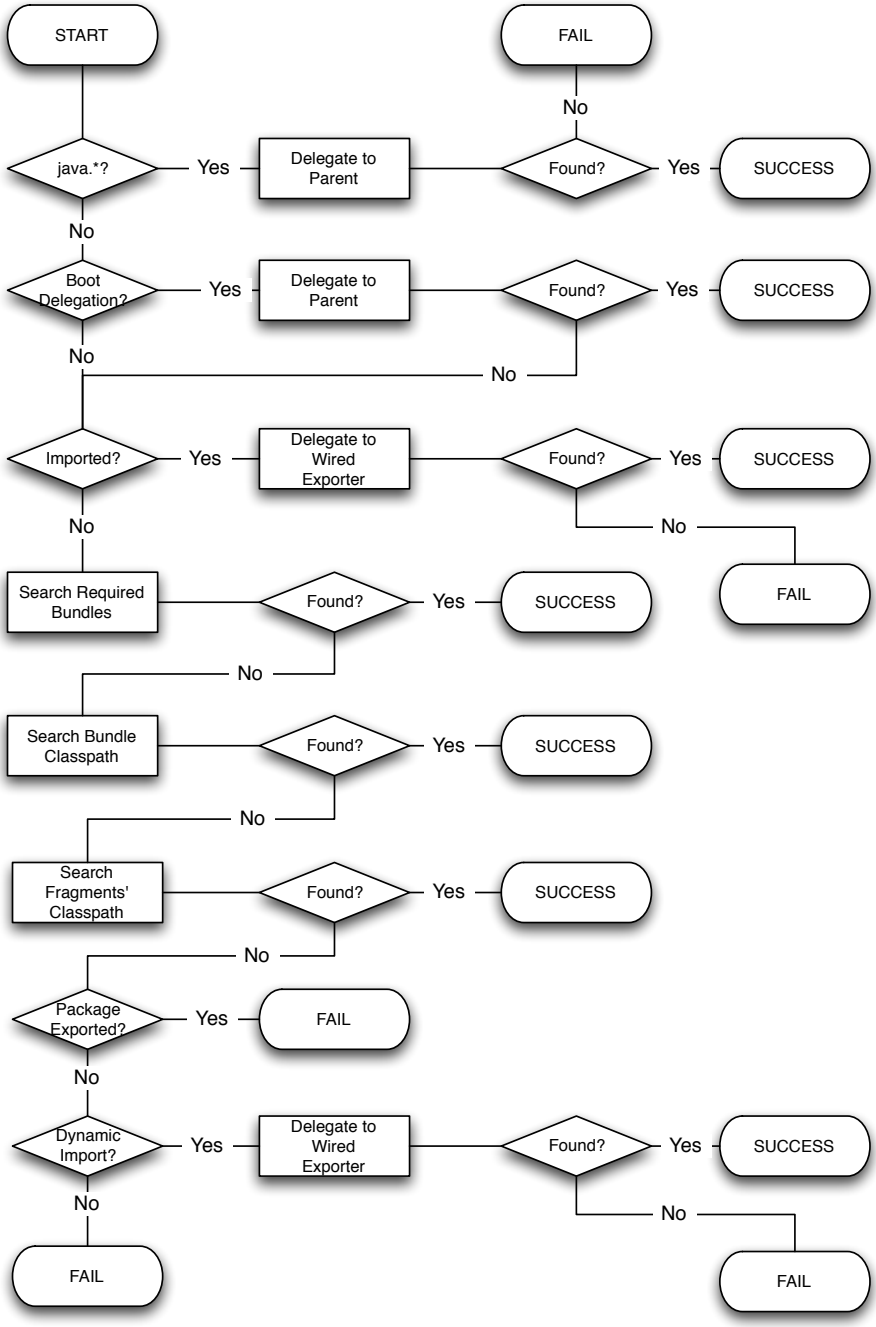


Figure 3.8.: Full OSGi Search Order

system bundle is to export packages from the base JRE libraries. As it is a special bundle, the class loader inside the system bundle follows different rules and is able to load from the main application or bootstrap class loaders.

However, the list of packages exported by the system bundle is not fixed, but subject to configuration: if you take a look at Felix's `config.properties` file you will see a property named `org.osgi.framework.system.packages` which lists the packages to be exported from the system bundle. The list is different depending on which version and edition of Java we are running, because later versions of Java have more packages in their API.

This approach of importing JRE libraries using the normal wiring system provides a very clean approach to environmental dependencies. An advantage is that the importing bundle doesn't necessarily know or care that the exporter of a package is the system bundle. Some APIs which are distributed as part of the base JRE libraries in recent versions of Java can still be used on older Java versions if they are obtained as separate JAR files. For example, Java 6 includes scripting capabilities[2] in the `javax.script` package, however there is also a JAR available[3] which implements the same functionality and works on Java 1.4 and Java 5. We could take that JAR and create a bundle from it, so that if another bundle wishes to use scripting functionality it can simply import the `javax.script` package.

3.10. Execution Environments

Unfortunately not all environmental dependencies can be handled through importing packages. As well as the `javax.*` packages, different versions of Java have different sets of `java.*` packages, different classes in those packages and even different fields and methods in those classes.

Some of these changes are easy to catch. For example, Java 5 introduced several new methods on core classes such as `java.lang.String`. The following code will not compile on Java 1.4 because the `contains()` method was added in Java 5:

```
public class Java5Test {
   public boolean stringTest(String s) {
      return s.contains("foo");
   }
}
```

Java provides a simple defence mechanism against us compiling this code with a Java 5 compiler and then running it on a Java 1.4 environment, in the form of the class file version. The Java 5 compiler will tag all class files it creates

[2] as defined by JSR 223[?]
[3] From https://scripting.dev.java.net/

with version 49.0, which will be rejected by a Java 1.4 VM as it expects the version to be no higher than 48.0.

In principle we could work around this mechanism using the `-source` and `-target` flags of the `javac` compiler, in which case the class would successfully load in the 1.4 VM but throw a `NoSuchMethodError` when it reached the `contains()` method. However it's difficult to imagine why anybody would inflict this kind of pain on themselves.

Unfortunately this defence mechanism still kicks in rather later than we would like it to. In OSGi terms, our bundle will resolve without any issues, but as soon as the offending class is loaded — which could happen at any arbitrary time after bundle resolution — we will get an exception. Even the error message is unhelpful:

```
Exception in thread "main" java.lang.UnsupportedClassVersionError:
Java5Test (Unsupported major.minor version 49.0)
        at java.lang.ClassLoader.defineClass0(Native Method)
        at java.lang.ClassLoader.defineClass(ClassLoader.java:539)
        at ....
```

It would be far better if we could see a clear error message "this code requires Java 5" and if that error happened at bundle resolution time.

Another problem is that a simple class version number cannot accurately represent the complex evolution tree of Java. There's not only the line of standard edition releases from 1.1 through to 6, but also the mobile edition releases CDC 1.0, CDC 1.1 and various profiles such as Foundation 1.0 and 1.1, PersonalJava 1.1 and 1.2, and so on. Even the linear evolution of Java SE may begin to look more complex in the future if Google's Android[**?**] or open source forks of OpenJDK pick up significant market share. Ideally, we would like to be able to specify exactly which of these environments our bundle runs on.

OSGi offers this functionality in the form of another kind of bundle resolution constraint. We can specify a list of the "execution environments" supported by our bundles, and at runtime the current Java VM must either be one of those environments or strictly upwards compatible with one of them, otherwise the bundle will fail to resolve. Furthermore OSGi tools will tell us clearly why the bundle cannot be resolved. In return we must ensure that our bundles only use packages, classes and methods that are available in *all* of the listed environments.

There is no fixed list of execution environments, since it is subject to change as the JCP creates new versions and editions of Java. However, the following set is currently supported by all three open source OSGi implementations:

- CDC-1.0/Foundation-1.0
- CDC-1.1/Foundation-1.1
- JRE-1.1

- J2SE-1.2
- J2SE-1.3
- J2SE-1.4
- J2SE-1.5
- J2SE-1.6
- OSGi/Minimum-1.0
- OSGi/Minimum-1.1

For example to specify that our bundle requires Java 5, we can add the following line to the bnd descriptor:

```
Bundle-RequiredExecutionEnvironment: J2SE-1.5
```

This entry will simply be copied verbatim to the `MANIFEST.MF` and recognised by the framework.

The first eight environments should need little explanation — they simply correspond directly with major releases of the Java specification. However the last two, OSGi/Minimum-1.0 and 1.1, are artificial environments that represent a subset of both Java SE and CDC 1.0 or 1.1. By choosing to target the Minimum-1.0 environment we can make our bundle run essentially everywhere, or at least everywhere that OSGi itself can run.

It's a good idea to add an execution environment declaration to every bundle you build. However this is only half the problem. Having declared that our bundle requires a particular environment we need to have processes in place to ensure our code really does only use features from that environment.

It is a common misconception that the `-source` and `-target` flags of the compiler can be used to produce, say, Java 1.4-compatible code using a Java 5 compiler. However these are not sufficient on their own. The `-source` flag controls only language features, so setting it to 1.4 turns off generics, annotations, for-each loops, and so on. The `-target` flag controls the class file version, so setting this to 1.4 makes the class file readable by a 1.4 VM. Neither of these flags do anything to restrict the APIs used from code, so we can still quite easily produce code that calls `String.contains()` or any other Java 5-only APIs.

The most practical and reliable way to produce 1.4-compatible code is to build it with version 1.4 of the JDK. Likewise for 1.3-compatible code, 5-compatible code, etc. The key is the `rt.jar` which contains the JRE library for that version of the JDK: only by compiling with the correct `rt.jar` in the classpath can we ensure API compatibility with the desired version. Since `rt.jar` is not available as a separate download, we must obtain the whole JDK.

3.11. Fragment Bundles

In Release 4, OSGi introduced the concept of Fragment bundles. A fragment, as its name suggests, is a kind of incomplete bundle. It cannot do anything on its own: it must attach to a host bundle, which must itself be a full, non-fragment bundle. When attached, the fragment can add classes or resources to the host bundle. At runtime, its classes are merged into the internal classpath of the host bundle.

What are fragments useful for? Here is one possibility: imagine you have a library which requires some platform-specific code. That is, the implementation of the library differs slightly when on Windows versus Mac OS X or Linux. A good example of this is the SWT library for building graphical user interfaces, which uses the "native" widget toolkit supplied by the operating system.

We don't want to create entirely separate bundles for each platform, because much of the code is platform independent, and thus would have to be duplicated. So what about isolating the platform independent code into its own bundle (let's call this bundle A), and making separate bundles for the platform-specific portion (which we will call P_{Win}, P_{MacOS}, etc.)? Unfortunately this separation is not always easy to achieve, because the platform-specific bundles P_* may have lots of dependencies on the platform independent bundle, A, and in order to import those dependencies into P_*, they would have to be exported by A. However, those exports are really internal features of the library; we don't want any other bundles except P_* to access them. But once exported, they are available to anybody — in OSGi, there is no way to export packages only to specific importers.

The solution to this problem is to make P_* into fragments hosted by A rather than fully fledged bundles. Suppose we are running on Windows. In this case, the fragment P_{Win} will be merged at runtime into the internal classpath of A. Now P_{Win} has full visibility of all packages in A, *including* non-exported packages, and likewise A has full visibility of all packages in P_{Win}.

Another use case is providing resources that differ depending on the locale of the user. Typically GUI applications need to provide resource bundles — usually properties files — containing all of the natural-language strings in the GUI. By separating these resources into fragments of the host bundle, we can save disk space and download time by delivering only the language fragment that the user wants.

We can even deliver different functionality for different locales. For example, written sentences in Japanese contain no spaces, yet still have discrete words which must not be split by a line break. The challenge for a word processor is to know where it is valid to insert a line break: special heuristic algorithms must be used. English may seem simpler, but even here we need to work out where long words can sensibly be split by a hyphen. Using fragments,

we can separate this functionality from the base language-independent word processing functionality, and deliver only the most appropriate set of fragments for the language the user wishes to use.

3.12. Class Space Consistency and "Uses" Constraints

TODO

4. Services

OSGi provides one of the most powerful module systems in any language or platform, and we saw just a small amount of its power in the previous section. However what we have seen so far only addresses the management of static, compiled-in dependencies between modules.

This alone is not enough. A module system that only supports static dependencies is fragile and fails to offer extensibility. Fragility means that we cannot remove any single module without breaking almost the entire graph. Lack of extensibility means we cannot add to the functionality of a system without recompiling parts of the existing system to make it aware of new components.

To build a module system that is both robust and extensible, we need the ability to perform *late binding*.

4.1. Late Binding in Java

One of the most important goals of the Object Oriented Programming movement is to increase the flexibility and reuse of code by reducing the coupling between the providers of functionality (objects) and the consumers of that functionality (clients).

In Java, we approach this goal through the use of *interfaces*. An interface is a purely abstract class which provides no functionality itself, but allows any class to implement it by providing a defined list of methods. By coding against interfaces rather than concrete types, we can write code that isolates itself almost entirely from any specific implementation.

For example, suppose we wish to write a mailbox scanner component, which periodically scans a mailbox for new messages. If we design our `Mailbox` interface carefully then we can write a scanner that neither knows nor cares which specific type of mailbox it is scanning — whether it be an IMAP mailbox, an RSS/ATOM "mailbox", an SMS mailbox, etc. Therefore we can reuse the same scanner component for all of these mailbox types without changing its code at all. Because of these benefits, so-called "interface based" programming is now widely recognised as good Java programming practice.

However, there is a catch: we cannot create new objects unless we know their specific type. Interfaces and abstract classes can be used to refer to an object

after it is created, but they cannot be used to instantiate new ones, because we need to know exactly what type of thing to create. This would cause a problem if our scanner component expects to create the mailbox object itself. A naïve solution might be to put some kind of switch in the constructor of the scanner to make it decide at runtime what kind of mailbox to create, as shown in Listing 4.1.

Listing 4.1 Naïve Solution to Instantiating an Interface

```
1 public class MailboxScanner {
2    private final Mailbox mailbox;
3    public MailboxScanner(String mailboxType) {
4       if("imap".equals(mailboxType)) {
5          mailbox = new IMAPMailbox();
6       } else if("rss".equals(mailboxType)) {
7          mailbox = new RSSMailbox();
8       } else {
9       // ...
```

Most programmers would recognise this as a terrible idea, and in OSGi it's even worse, because it will only work if the bundle containing the scanner imports *every* package that might contain a mailbox implementation class... including ones that might be written in the future!

The normal solution to this quandary is for the scanner not to try to instantiate the mailbox itself, but to allow a mailbox to be supplied to it by something else. This is the essence of late binding: the consumer of functionality is not bound to a specific provider until runtime. But who or what should this "something else" be?

There are many possible answers. A large application may have hundreds of classes like `MailboxScanner` which all need to be supplied with implementations of the interfaces they depend on, so a common theme in many solutions is to centralise object creation into an "Assembler" class. That central class may even support some form of scripting, to allow the network of objects to be rewired without recompilation.

4.1.1. Dependency Injection Frameworks

After writing a few examples of the "Assembler" class in different projects, it's easy to see that it is a common pattern that can be extracted out into a framework. And indeed several open source frameworks have emerged that do exactly this: popular examples in Java are the Spring Framework[**?**] and Google Guice[**?**].

Such a framework is often called a "Dependency Injection" (DI) framework[1]. Unfortunately both these and the manually-created Assembler pattern have

[1]They have also previously been called "Inversion of Control" (IoC) frameworks, but this

traditionally suffered from being mostly static: the wiring together of "beans" (i.e., plain Java objects, such as the mailbox implementation) with their consumers (other beans, such as the mailbox scanner) tends to happen once, at start-up of the application, and remains in force until the application is shutdown.

Static wiring results in many problems. The first is fragility due to a sensitive dependence on start-up ordering. For example if object B depends on object A, then A must be created before B. When we scale up to thousands of objects and their interdependencies, the dependency graph becomes brittle and far too complex for any human to understand. We must avoid circular dependencies between objects, at all costs.

Another problem is the impossibility of on-the-fly updates. Most production systems need to be patched occasionally as bugs are fixed or requirements change. In a statically wired dependency graph, we usually need to shut down the entire system even when updating only the tiniest corner of it.

4.1.2. Dynamic Services

OSGi solves these problems with dynamic *services*.

A service, like a bean in a DI framework, is a plain Java object. It is published in the OSGi Service Registry under the name of one or more Java interfaces, and consumers who wish to use it may look it up using any of those interfaces names. Services may consume other services, but rather than being wired into a fixed graph, services can be registered or unregistered dynamically at any time, so they form only temporary associations with each other.

Start ordering problems can now be solved easily: services start in any order. Suppose we start the bundle containing service B before starting the bundle containing service A. In this case, B simply waits for A to become available. Also we can update individual components without restarting the system. When taking away service A and replacing it with A', OSGi sends events to service B to keep it informed of the situation.

Services offer an interface-based programming model. The only requirement on a service is that it implements an interface; *any* interface will do, even ones from the base JRE or third-party libraries. The chosen interface — or interfaces, since a Java object can implement many interfaces — forms the primary addressing mechanism for a service. For example, a service publisher declares "I have this object available which is `Mailbox`." The consumers declare "I am looking for a `Mailbox`." The Service Registry provides a venue for

term has largely fallen out of use thanks to Martin Fowler, who popularised the term "Dependency Injection" in his 2004 article "Inversion of Control Containers and the Dependency Injection pattern"[?].

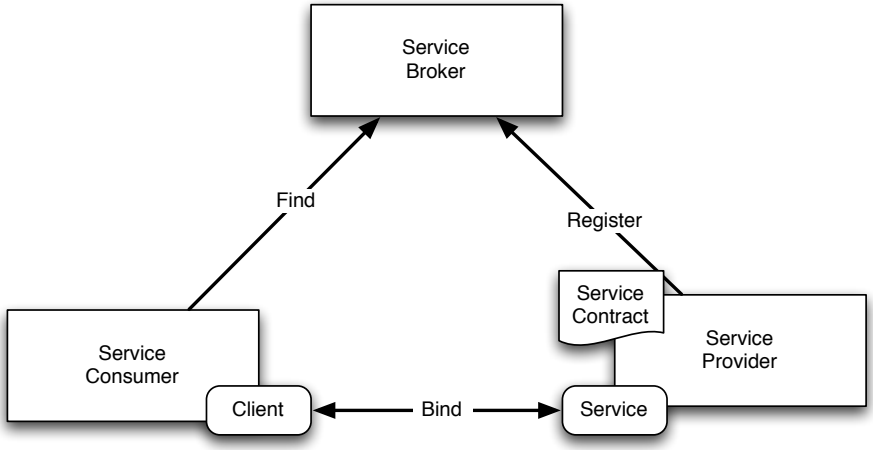


Figure 4.1.: Service Oriented Architecture

publishers and consumers to find each other. The consumer does not need to know the implementation class of the published service, just the interface through which it interacts[2].

OSGi's Service Registry has been called a form of Service Oriented Architecture, or SOA. Many people think of SOA as being associated with distributed computing, Web Services, SOAP and so on, but that is just one example of SOA, which is really just a pattern or style of architecture. OSGi services are limited in scope to a single JVM — they are not distributed[3] — yet they map very cleanly to the concepts of SOA. Figure 4.1 is adapted from a diagram used by the World-Wide Web Consortium to explain SOA, and we will see shortly how each of the abstract concepts in that diagram maps to a concrete entity in OSGi.

Sadly, there is some complexity cost involved in handling dynamic services versus static objects. It is obviously easier to make use of something if we know that it is always available! However, the real world is not static, and therefore we need to write systems that are robust enough to handle entities that come and go. The good news is that several high-level abstractions have been built on top of the OSGi Service Registry that greatly simplify the code you need to write while still taking advantage of dynamic behaviour. In fact, the Dependency Injection frameworks have started to support dynamic behaviour, and in the case of the Spring Framework that support is achieved

[2] In fact we can publish services under the names of abstract or concrete classes also, but this defeats object of using the Registry to decouple consumers from implementations.

[3] Although of course some people have sought to build distributed systems on top of OSGi services

through direct usage of the OSGi Service Registry.

In this chapter though, we look at the nuts and bolts of services, in order to gain a firm foundation when advancing to higher-level abstractions.

4.2. Registering a Service

Recall that in Section 3.4 we created an implementation of the `Mailbox` interface. However, as its package was not exported, there was no way for any other bundle to actually use that implementation! By registering an instance of the `FixedMailbox` class as a service, other bundles can access the object without needing to have a dependency on its class.

Registering a service is an example of interaction with the OSGi framework, and as always we need to have a `BundleContext` in order to do that. Therefore we need to write an implementation of `BundleActivator` as shown in Listing 4.2.

Listing 4.2 Welcome Mailbox Activator

```
1  package org.osgi.book.reader.fixedmailbox;

3  import org.osgi.book.reader.api.Mailbox;
4  import org.osgi.framework.BundleActivator;
5  import org.osgi.framework.BundleContext;

7  public class WelcomeMailboxActivator implements BundleActivator {

9      public void start(BundleContext context) throws Exception {
10         Mailbox mbox = new FixedMailbox();
11         context.registerService(Mailbox.class.getName(), mbox, null); //1
12     }

14     public void stop(BundleContext context) throws Exception {
15     }
16 }
```

The first two lines of code simply create a mailbox with some hard-coded messages in it. The line marked *//1* is the one that calls the framework to register the mailbox as a service, by calling the `registerService` method with three parameters:

1. The *interface name* under which the service is to be registered. This should be the name of a Java interface class, and will be the primary means by which clients will find the service.

2. The *service object* itself, i.e. the mailbox object that was instantiated in the previous lines. This object *must* implement the interface named in the previous parameter.

3. A set of *service properties*, which here has been left blank by passing `null`. We will look at service properties shortly.

We say that a service object is registered "under" an interface name, because for consumers the most important fact about our service is the interface it implements. In fact, just as Java objects can be implementations of multiple interfaces, we can register a service under multiple interface names, by calling a variant of the `registerService` method that takes an array of `String`s as its first parameter. When we do this, consumers can find the service using any one of those interface names. There is still just one service object registered: the additional entries can be considered aliases.

Note that we could have passed a literal String for the interface name i.e., `context.registerService("org.osgi.book.reader.api.Mailbox", mbox, null)`. However this is not good practice, because the class or package name might change in the future, for example when the code is refactored. It is better to use `Mailbox.class.getName()` because most IDEs can automatically update the import statement — and even if we are not using an IDE, we will get a helpful compilation error if we change the package name that `Mailbox` lives in without updating this source file.

Listing 4.3 Bnd Descriptor for the Welcome Mailbox Bundle

```
# welcome_mailbox.bnd
Private-Package: org.osgi.book.reader.fixedmailbox
Bundle-Activator: \
   org.osgi.book.reader.fixedmailbox.WelcomeMailboxActivator
```

Let's build this bundle and see what effect it has. The `bnd` descriptor should look like the one in Listing 4.3. After building, installing and starting the bundle, try typing `services`. You will see a long list of results, but somewhere within it you should see this:

```
osgi> services
...
{org.osgi.book.reader.api.Mailbox}={service.id=24}
  Registered by bundle: welcome_mailbox_0.0.0 [2]
  No bundles using service.
...
```

This shows we have successfully registered a service under the `Mailbox` interface, and it has been assigned a service ID of 24 — again, you will probably get a different ID when you run this for yourself. Incidentally you can try the `services` command on its own, without a bundle ID parameter, which will give you a list of all registered services by all bundles, although showing less detail.

4.3. Unregistering a Service

In the code listing in Section 4.2, we did not do anything in the `stop()` method of the activator, which may seem oddly asymmetrical. Usually we have to undo in the `stop()` method whatever we did in the `start()` method, so shouldn't we have to unregister the service?

Actually this is not necessary because the OSGi framework automatically unregisters any services registered by our bundle when it is deactivated. We don't need to explicitly clean it up ourselves, so long as we are happy for the service's lifecycle — the period of time during which it was registered and available to clients — to coincide with the lifecycle of the bundle itself. Sometimes though we need a different lifecycle for the service. We may only want to offer the service when some other conditions are met, and in that case we will have to control both registering and unregistering ourselves.

Suppose, for example, we only want our service to be available while a particular file exists on the filesystem. Perhaps that file contains some messages which we want to offer as a mailbox: clearly we can only offer the service if the file actually exists. To achieve this we would create a polling thread using the same pattern we saw in Section 2.11. The code is shown in Listing 4.4.

Here we see, at marker *1*, that the `registerService()` method returns an object of type `ServiceRegistration`, and we can use that object to unregister the service later. Each pass through the loop we check whether the file `messages.txt` exists in your home directory[4]. If it does, and the service is not currently registered, then we register it. If the file does *not* exist, and the service *is* currently registered, then we unregister it.

For the purposes of this example, the actual implementation of `FileMailbox` is not particularly interesting. If you simply wish to get this code working and watch the service register and unregister, then you will need to create a simple stub implementation. The code in Listing 4.5 will suffice, assuming we never actually call the methods of the `FileMailbox` object.

The `bnd` descriptor should be as in Listing 4.6.

By creating and deleting the `messages.txt` file, you should be able to see the service appear and disappear, albeit with up to five seconds' delay. Incidentally if we were to provide a real implementation of `FileMailbox`, this delay would be one of the things we would have to take into account in order to make the service robust: we should be able to handle the situation where a client request arrives during the period between the file being deleted and the service being

[4] On Windows XP, this is usually `C:\Documents and Settings\YourName` and on Windows Vista it is `C:\Users\YourName`. Users of non-Windows operating systems tend to know how to find their home directory already.

Listing 4.4 File Mailbox Activator

```
1  package org.osgi.book.reader.filemailbox;

3  import java.io.File;
4  import java.util.Properties;

6  import org.osgi.book.reader.api.Mailbox;
7  import org.osgi.framework.BundleActivator;
8  import org.osgi.framework.BundleContext;
9  import org.osgi.framework.ServiceRegistration;

11 public class FileMailboxActivator implements BundleActivator {

13    private Thread thread;

15    public void start(BundleContext context) throws Exception {
16       File file = new File(System.getProperty("user.home")
17             + System.getProperty("file.separator") + "messages.txt");
18       RegistrationRunnable runnable = new RegistrationRunnable(
19             context, file, null);
20       thread = new Thread(runnable);
21       thread.start();
22    }

24    public void stop(BundleContext context) throws Exception {
25       thread.interrupt();
26    }
27 }

29 class RegistrationRunnable implements Runnable {

31    private final BundleContext context;
32    private final File file;
33    private final Properties props;

35    public RegistrationRunnable(BundleContext context, File file,
36          Properties props) {
37       this.context = context;
38       this.file = file;
39       this.props = props;
40    }

42    public void run() {
43       ServiceRegistration registration = null;
44       try {
45          while (!Thread.currentThread().isInterrupted()) {
46             if (file.exists()) {
47                if (registration == null) {
48                   registration = context.registerService(        // 1
49                         Mailbox.class.getName(),
50                         new FileMailbox(file), props);
51                }
52             } else {
53                if (registration != null) {
54                   registration.unregister();
55                   registration = null;
56                }
57             }
58             Thread.sleep(5000);
59          }
60       } catch (InterruptedException e) {
61          // Allow thread to exit
62       }
63    }
64 }
```

Listing 4.5 File Mailbox (Stub Implementation)

```
1  package org.osgi.book.reader.filemailbox;

3  import java.io.File;

5  import org.osgi.book.reader.api.Mailbox;
6  import org.osgi.book.reader.api.Message;

8  /**
9   * Warning: Empty stub implementation
10  */
11 public class FileMailbox implements Mailbox {

13    private static final long[] EMPTY = new long[0];

15    public FileMailbox(File file) {}
16    public long[] getAllMessages() { return EMPTY; }
17    public Message[] getMessages(long[] ids) {
18       return new Message[0];
19    }
20    public long[] getMessagesSince(long id) { return EMPTY; }
21    public void markRead(boolean read, long[] ids) { }
22 }
```

Listing 4.6 Bnd Descriptor for File Mailbox Bundle

```
# file_mailbox.bnd
Private-Package: org.osgi.book.reader.filemailbox
Bundle-Activator: org.osgi.book.reader.filemailbox.FileMailboxActivator
```

unregistered. In that case we would have to return an error message to the client.

4.4. Looking up a Service

Having seen how to register and unregister services, the next logical step is to look at how to look up and call methods on those services.

Perhaps surprising, this can be a little tricky. The problem is that services, as we have seen, can come and go at any time. If we look for a service at a particular instant, we might not find it, but we would find it if we looked two seconds later. Alternatively we could access a service twice in a row, but get a different service the second time because the one we got first time is no longer around.

Fortunately there is lots of support, both in the OSGi specifications themselves and in external third-party libraries, to help us to abstract away this complexity. In fact programming with dynamic services in OSGi need be hardly any more complex than with static dependency injection, yet it is far more powerful. However, in this section we will look at the most low-level way of accessing services. This is partially to provide a firm base of understanding for when we get onto the more convenient approaches, and partially to drive home the truly dynamic nature of OSGi services.

Suppose we wish to write a bundle that accesses one of the `Mailbox` services and prints the current total number of messages in that mailbox. To keep things as simple as possible we will do this in the `start()` method of an activator, for example `MessageCountActivator` in Listing 4.7.

At marker *1*, we ask the framework to find a `ServiceReference` for a named Java interface. As before, we avoid encoding the interface name as a literal string.

After checking that the service reference is not `null`, meaning the mailbox service is currently available, we proceed to request the actual service object at marker *2*. Again we have to check if the framework returned `null` — this is because although the service was available at marker *1*, it may have become unavailable in the time it took us to reach marker *2*.

At marker *3* we actually call the service. Because our `mbox` variable holds the actual service object rather than any kind of proxy, we can call the methods on it just like any normal Java object.

Finally at marker *4* we "un-get" the service — in other words we let the framework know that we are no longer using it. This is necessary because the framework maintains a count of how many bundles are using a particular

Listing 4.7 Message Count Activator

```
1  package org.osgi.tutorial;
2
3  import org.osgi.book.reader.api.Mailbox;
4  import org.osgi.book.reader.api.MailboxException;
5  import org.osgi.framework.BundleActivator;
6  import org.osgi.framework.BundleContext;
7  import org.osgi.framework.ServiceReference;
8
9  public class MessageCountActivator implements BundleActivator {
10
11    private BundleContext context;
12
13    public void start(BundleContext context) throws Exception {
14       this.context = context;
15       printMessageCount();
16    }
17
18    public void stop(BundleContext context) throws Exception {
19    }
20
21    private void printMessageCount() throws MailboxException {
22       ServiceReference ref = context                              // 1
23             .getServiceReference(Mailbox.class.getName());
24
25       if (ref != null) {
26          Mailbox mbox = (Mailbox) context.getService(ref);        // 2
27          if (mbox != null) {
28             try {
29                int count = mbox.getAllMessages().length;          // 3
30                System.out.println("There are " + count + "messages");
31             } finally {
32                context.ungetService(ref);                         // 4
33             }
34          }
35       }
36    }
37 }
```

service: when that count is zero, it knows that the service can safely be removed when the bundle providing it is deactivated. We have placed the call to `ungetService()` inside a `finally` block to ensure that it is always called on exit from our method, even if an uncaught exception occurs.

Listing 4.8 shows the Bnd descriptor.

Listing 4.8 Bnd Descriptor for the Message Counter Bundle

```
# message_count.bnd
Private-Package: org.osgi.tutorial
Bundle-Activator: org.osgi.tutorial.MessageCountActivator
```

Stepping back from the code, you may wonder why accessing a service requires a two-stage process of first obtaining a reference and then obtaining the actual service object. We will look at the reasons for this in the next section.

The main problem with this code is that it's just too long! Simply to make a single call to a service, we have to write two separate `null` checks and a `try` / `finally` block, along with several noisy method calls to the OSGi framework. We certainly don't want to repeat all of this each time we access a service. Also, the code does not behave particularly well when the `Mailbox` service is unavailable: it simply gives up and prints nothing. We could at least print an error message, but even that is unsatifactory: what if we really *need* to know how many messages are in the mailbox? The information will be available as soon as the mailbox service is registered, but it's just bad luck that the above bundle activator has been called first.

4.5. Service Properties

In addition to the interface name, we may wish to associate additional metadata with our services. For example, in this chapter we have registered several instances of the `Mailbox` service, each with different characteristics. It would be nice to tell clients something about each mailbox so that they can, if they wish, obtain only one specific mailbox, or at least report something to the user about what kind of mailbox they have obtained. This is done with service properties.

Recall that when registering the service in Section 4.2, we left the final parameter `null`. Instead we can pass in a `java.util.Properties` object containing the properties that we want to set on the service[5].

[5] Actually the type of this parameter is `java.util.Dictionary`, of which `java.util.Properties` is a sub-class. Why not use the `java.util.Map` interface, which the `Properties` class also implements? Simply because `Map` has "only" existed since Java 1.2, so it cannot be used on all platforms supported by OSGi!

Let's modify `WelcomeMailboxActivator` to add a "mailbox name" property to the service. The new `start` method is shown in Listing 4.9.

Listing 4.9 Adding Service Properties to the Welcome Mailbox

```
1    public void start(BundleContext context) throws Exception {
2        Mailbox mbox = new FixedMailbox();

4        Properties props = new Properties();
5        props.put(Mailbox.NAME_PROPERTY, "welcome");
6        context.registerService(Mailbox.class.getName(), mbox, props);
7    }
```

Note that we avoid hard-coding the property name everywhere we use it, as this can be error prone and difficult to change later. Instead we use a constant that was defined in the Mailbox interface itself. This is a suitable place to put the constant because it is guaranteed to be visible to both service implementers and service consumers.

Try rebuilding this bundle and updating it in Equinox. If we now re-run the `services` command we should see the property has been added to the service:

```
osgi> services
...
{org.osgi.book.reader.api.Mailbox}={mailboxName=welcome,service.id=27}
  Registered by bundle: welcome_mailbox_0.0.0 [2]
  No bundles using service.
...
```

The other entry we see here, `service.id`, is a built-in property that has been added by the framework. Another built-in property, `objectClass`, is also added: this indicates the interface name that the service is published under. It is a property like `mailboxName` and `service.id`, but because it is such an important property Equinox chooses to bring its value to the front of the entry. There are other standard property names defined in the OSGi specification, but only `service.id` and `objectClass` are mandatory, so they appear on every service. These two, and the rest, can be found in the class `org.osgi.framework.Constants`.

To query the properties on a service, we simply call the `getProperty` method of the `ServiceReference` to get a single property value. If we need to know about all of the properties on the service we can call `getPropertyKeys` method to list them.

Service properties provide a clue as to why we need a two-stage process for looking up a service, i.e. first obtaining a `ServiceReference` before obtaining the actual service. One reason for this is sometimes we don't need the service object (or at least not *yet*) but only its properties. When we obtain the service object, the OSGi framework must keep track of the fact we are using it, creating a very small amount of overhead. It is sensible to avoid that overhead when it is not needed.

Also, service references are small, lightweight objects that can be passed around, copied or discarded at will. The framework does not track the service reference objects it hands out. Also service reference objects can be passed easily between bundles, whereas it can cause problems to pass an actual service instance to another bundle: that would prevent the framework from being able to track which bundles are using each service. In situations where we want to ask another bundle to do something with a service, we should pass the reference and let the other bundle call `getService` and `ungetService` itself.

4.6. Introduction to Service Trackers

To address the excess verbosity of the code in Section 4.4, we can refactor it to use a utility class provided by OSGi called `ServiceTracker`

In fact `ServiceTracker` is one of the most important classes you will use as an OSGi programmer, and it has many more uses than just the one we will be taking advantage of in this code. But for now, the code in Listing 4.10 simply does the same thing as before, and in particular it is no better at dealing with a missing `Mailbox` service:

Listing 4.10 Message Count Activator — `ServiceTracker` version

```
1  package org.osgi.tutorial;
2
3  import org.osgi.book.reader.api.Mailbox;
4  import org.osgi.book.reader.api.MailboxException;
5  import org.osgi.framework.BundleActivator;
6  import org.osgi.framework.BundleContext;
7  import org.osgi.util.tracker.ServiceTracker;
8
9  public class MessageCountActivator2 implements BundleActivator {
10
11    private ServiceTracker mboxTracker;
12
13    public void start(BundleContext context) throws Exception {
14       mboxTracker = new ServiceTracker(context, Mailbox.class     // 1
15             .getName(), null);
16       mboxTracker.open();                                         // 2
17       printMessageCount();
18    }
19
20    public void stop(BundleContext context) throws Exception {
21       mboxTracker.close();                                        // 3
22    }
23
24    private void printMessageCount() throws MailboxException {
25       Mailbox mbox = (Mailbox) mboxTracker.getService();          // 4
26       if (mbox != null) {
27          int count = mbox.getAllMessages().length;                // 5
28          System.out.println("There are " + count + "messages");
29       }
30    }
31 }
```

At first glance this code is barely any shorter than the previous example! Nevertheless we have achieved something important: in exchange for a little extra initial effort, it is now much easier to call the service, as we can see in the method `printMessageCount`. In real code, we would probably make many separate calls to the service, so it is far better to repeat the four lines of code in this `printMessageCount` than the nine lines of code in the previous version.

The first difference, which we can see at marker *1* of the `start` method, is that instead of saving the bundle context directly into a field, we instead construct a new `ServiceTracker` field, passing it the bundle context and the name of the service that we are using it to track. Next at marker *2*, we "open" the tracker, and at marker *3* in the `stop()` method we "close" it. We will look later at what is really going on under the covers when we open and close a service tracker, but for now just remember that the tracker will not work until it is opened.

The next difference is at marker *4*, where we call `getService` on the tracker. Refreshingly, this immediately gives us the actual service object (if available) rather than a `ServiceReference`. So we simply go ahead and call the service at marker *5*, bearing in mind that we still need to check if the service was found. Also, we don't need to clean up after ourselves with a `finally` block: we simply let the variable go out of scope, as the tracker will take care of releasing the service.

Here's another interesting thing we can do with `ServiceTracker`. In the version of `printMessageCount` shown in Listing 4.11 we don't want to fail immediately if the service is unavailable; instead we would like to wait up to five seconds for the service to become available.

Listing 4.11 Waiting for a Service

```
1    private void printMessageCount(String message)
2          throws InterruptedException, MailboxException {
3       Mailbox mbox = (Mailbox) mboxTracker.waitForService(5000);
4       if (mbox != null) {
5          int count = mbox.getAllMessages().length;
6          System.out.println("There are " + count + "messages");
7       }
8    }
```

When the mailbox service is available, `waitForService` works exactly the same as `getService`: it will immediately return the service instance. However if the service is not currently available, the call will block until either the service becomes available or 5000 milliseconds has passed, whichever is sooner. Only after 5000 milliseconds has passed without the service becoming available will it return `null`.

However, `waitForService` needs to be used with caution, and in particular it should not be called from the `start` or `stop` methods of a `BundleActivator`,

because those methods are supposed to return control quickly to the framework. This topic is discussed in more depth in Section **??**.

4.7. Listening to Services

In both versions of the "message counting" bundle above, we failed to handle a missing mailbox service gracefully. If the mailbox wasn't there, we simply gave up.

Sometimes giving up is exactly the right thing to do. For example there is a standard log service in OSGi which clients can use to send logging messages to an application-wide log. Suppose a component wants to write a message to the log, but it discovers that the log service is not currently available — what should it do? Usually it should give up, i.e. not write the message to the log, but carry on with its main task. Logs are nice if we can have them, but should not get in the way of actually running the application.

However at other times, we need to do better. Some components are essentially useless without the services that they consume: a component that counts messages in a mailbox is pointless when there are no mailboxes. We might say that it has a *dependency* on the mailbox service.

Services often depend on other services. For example, suppose we have a relational database that is exposed to our application as a service under the `javax.sql.DataSource` interface — we might then wish to offer a mailbox service that obtains its messages from the database. In that case, the "database mailbox" service would have a dependency on the `DataSource` service, and it would be useless when the `DataSource` is not available. This is very similar to the `FileMailbox` that was discussed in Section 4.3, in which we registered the service only when the backing file was available, and unregistered the service when it was not. In the case of the database mailbox, we can register the service when the `DataSource` service is available, and unregister when it is not.

In the file example, we had to write a thread to poll for the existence of the backing file. Polling is not ideal because it wastes CPU time, and there is inevitably a delay between the state of the file changing and our program detecting that change. But there is currently no way in Java — short of using platform-specific native code — to "watch" or "listen" to the state of a file[6], so polling is our only choice. Not so with services. By listening to the service registry, we can be notified immediately when the `DataSource` service is registered or unregistered, and react immediately by registering or unregistering the corresponding mailbox.

[6] JSR 203[**?**] (also known as "NIO.2") seeks to address this limitation for Java 7.

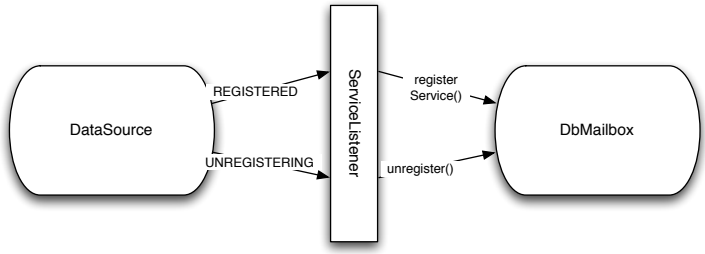


Figure 4.2.: Updating a Service Registration in Response to Another Service

To achieve this, we listen to events published by the service registry. Whenever a service is either registered or unregistered, the framework publishes a `ServiceEvent` to all registered `ServiceListener`s. Any bundle can register a `ServiceListener` through its `BundleContext`. Therefore we could write a simple listener that, when it receives an event of type REGISTERING, registers a new `DbMailbox` service, and when it receives an event of type UNREGISTERING, unregisters that service.

Unfortunately, this will not work.

The problem is that service listeners are only told about state *changes*, not about pre-existing state. If the `DataSource` is already registered as a service before we start our listener, then we will never receive the REGISTERING event, but nevertheless we need to register the `DbMailbox` service. Therefore we really need to do two things:

- Scan the service registry for a pre-existing service under the `DataSource` interface; if such exists, then register a `DbMailbox` for it.
- Hook up a service listener which will register a `DbMailbox` when a `DataSource` service is registered, and unregister it when the `DataSource` service is unregistered

But if we do those two things in the stated order, it will *still* not work! There will be a window of time between scanning the pre-existing services and starting the service listener, and if a service is registered during that window, we will miss it completely. Therefore we need to reverse the two steps: *first* hook up the listener, *then* scan the pre-existing services. Now there is the potential for overlap: if a service registers between the two steps it will be handled both by the scanning code and also by the listener, so we need to be careful to guard against duplication.

All this sounds nightmarishly complex, and indeed the code needed to get all of this right is very unpleasant. We are not even going to look at an example

because you should never have to write any code like this. Instead you should be using `ServiceTracker`.

4.8. Tracking Services

Although we saw `ServiceTracker` in Section 4.6, we only used it in a very limited way. In this section we will see the main purpose for `ServiceTracker`: hiding the complexities of listening to and consuming dynamic services. Rather than simply "listening", which is passive, we wish to actively "track" the services we depend on.

Before proceeding to the example code, we will need to have an implementation of the `DbMailbox` class but, like `FileMailbox`, the actual implementation is not interesting as part of this exposition. Therefore we will use another stub class — the full definition of which is left as an exercise.

Let's write a bundle activator using `ServiceTracker`: this is shown in Listing 4.12. The `start` and `stop` methods of this activator look very similar to our first example of using a tracker in Section 4.6: we simply create and open the tracker on start-up (markers *1* and *2*), and close it on shutdown (marker *3*). However this time the third parameter to the constructor of `ServiceTracker` is not `null`. Instead, we pass in an instance of the `ServiceTrackerCustomizer` interface, which tells the tracker what to do when services are added, removed or modified.

Our customizer simply registers a `DbMailbox` service whenever a `DataSource` service is added, and unregisters it when the `DataSource` service is removed.

Why does this not suffer from the same limitations as the `ServiceListener`-based approach described Section 4.7? Simply because, unlike a listener, the adding and removed methods of `ServiceTracker` are called not only when the state of a service changes but also when the tracker is opened, to notify us of pre-existing services. The `addingService()` method is called multiple times when the tracker is opened, once for each service currently registered, and it is also called whenever a new service is registered at any time later for as long as the tracker is open. Furthermore the `removedService()` is called any time a service that we have been previously been notified of goes away, and it is also called for each service when the tracker closes. Therefore we can deal with services in a uniform fashion without needing to distinguish between pre-existing services and ones that are registered while our listener is active. This greatly simplifies the code we need to write.

Incidentally the `ServiceTracker` class does not use any special hooks into the framework, it builds on the existing facilities that we have already seen. When we call `open()` on a service tracker, it hooks up a `ServiceListener` and then

Listing 4.12 Database Mailbox Activator

```
1  package org.osgi.book.reader.dbmailbox;

3  import javax.sql.DataSource;

5  import org.osgi.book.reader.api.Mailbox;
6  import org.osgi.framework.BundleActivator;
7  import org.osgi.framework.BundleContext;
8  import org.osgi.framework.ServiceReference;
9  import org.osgi.framework.ServiceRegistration;
10 import org.osgi.util.tracker.ServiceTracker;
11 import org.osgi.util.tracker.ServiceTrackerCustomizer;

13 public class DbMailboxActivator implements BundleActivator {

15    private BundleContext context;
16    private ServiceTracker tracker;

18    public void start(BundleContext context) throws Exception {
19       this.context = context;

21       tracker = new ServiceTracker(context, DataSource.class
22             .getName(), new DSCustomizer());                    // 1
23       tracker.open();                                          // 2
24    }

26    public void stop(BundleContext context) throws Exception {
27       tracker.close();                                         // 3
28    }

30    private class DSCustomizer implements ServiceTrackerCustomizer {

32       public Object addingService(ServiceReference ref) {
33          DataSource ds = (DataSource) context.getService(ref);  // 4

35          DbMailbox mbox = new DbMailbox(ds);
36          ServiceRegistration registration = context.registerService(
37                Mailbox.class.getName(), mbox, null);            // 5

39          return registration;                                   // 6
40       }

42       public void modifiedService(ServiceReference ref,
43             Object service) {
44       }

46       public void removedService(ServiceReference ref, Object service) {
47          ServiceRegistration registration =
48             (ServiceRegistration) service;                     // 7

50          registration.unregister();                             // 8
51          context.ungetService(ref);                             // 9
52       }
53    }
54 }
```

scans the pre-existing services, eliminates duplicates etc. The internal code is still complex, but it has been written for us (and exhaustively tested) to save us from having to do it ourselves.

Let's look at the customizer class in a little more detail. At marker *4* we receive a `ServiceReference` from the tracker that points at the newly registered service, and we ask the framework to de-reference it to produce the actual service object. We use the result to create a new instance of `DbMailbox` and at marker *5* we register it as a new mailbox service. At marker *6* we return the registration object back to the tracker.

Why return the registration object? The signature of `addingService()` simply has a return type of `Object`; the tracker does not care what type of object we return, and we can return anything we like. However the tracker promises to remember the object (*unless* we return `null`, see below) and give it back to us in the following situations:

- When we call `getService`, the tracker will return whatever object we returned from `addingService`.
- When the underlying service is modified or unregistered, the tracker will call `modifiedService` or `removedService` respectively, passing both the service reference and the object that we returned from `addingService`.

Because of the first one of these promises, it's conventional to return the actual service object from `addingService` — which in the code above would be the `ds` variable. But that is not a requirement[7]. In general we should return whatever will be most useful for us to find the information or data structure that might need to be updated later. In the above code we want to "update" the registration of the mailbox service, so we return the registration object. Other times we might be storing information into a `Map`, so we return one of the keys from the `Map`.

An exception to this rule is when we return `null` from `addingService`, which the tracker takes to mean that we don't care about this particular service reference. That is, if we return `null` from `addingService`, the tracker will "forget" that particular service reference and will not call either `modifiedService` or `removedService` later if it is modified or removed. Thus returning `null` can be used as a kind of filter, but in the next section we will see a more convenient way to apply filters declaratively.

Looking back at our example, since we know that the second parameter of `removedService` will be of type `ServiceRegistration`, we are able to cast it back to that type at marker *7*. Then we can simply unregister it (marker *8*) and "unget" the `DataSource` service (marker *9*). The final step is necessary as the mirror image of calling `getService` in the adding method.

[7] The API documentation for `ServiceTrackerCustomizer` states that we "should" return the service object, but I believe that to be slightly incorrect, or not completely satisfactory.

4.9. Filtering on Properties

Section 4.5 described how properties can be added to services when they are registered, and how the properties on a `ServiceReference` can be introspected. Now let's look at another important use for properties: filtering service look-ups.

Both of the look-up code samples we saw, in Sections 4.4 and 4.6, obtained a single instance of the mailbox service. Yet it should be clear by now that there can be an arbitrary number of service instances for each particular type, because any bundle can register a new service under that type. Therefore it is sometimes necessary to further restrict the set of services obtained by a look-up. This is done with *filters*, which are applied to the properties of the service. A filter is a simple string, using a format which is easy to construct either manually or programmatically.

For example, suppose we wish to find the "welcome" mailbox service, and not any other kind of mailbox. Recall that that mailbox service has a property named `mailboxName` with the value "welcome". The filter string required to find this service is simply:

```
(mailboxName=welcome)
```

Suppose we added a further property to the welcome mailbox indicating the language. To find the English version, which should have the `lang` property set to "`en`", we construct the following composite filter:

```
(&(mailboxName=welcome) (lang=en))
```

Some languages have variants, such as `en_UK` and `en_US` for British and American English respectively. Suppose we want to match any kind of English:

```
(&(mailboxName=welcome) (lang=en*))
```

Finally, suppose we want either German ("`de`") or any form of English *except* Canadian:

```
(&(mailboxName=welcome) (|(lang=de)(lang=en*)) (!(lang=en_CA)))
```

This syntax is borrowed directly from LDAP search filters, as defined in [**?**]. A filter is either a simple operation, or a composite. Here are some examples of simple operations:

`(foo=*)`	Property `foo` is present
`(foo=bar)`	Value of property `foo` is equal to "bar"
`(count>=1)`	Value of property `count` is 1 or greater
`(count<=10)`	Value of property `count` is 10 or less
`(foo=bar*)`	Value of property `foo` is a string starting "bar"...

Composite filters can be built up from simple filters, or they might compose filters which are themselves composites. Here are the composition operations:

`(!(filter))`	Boolean "NOT": filter is false
`(&(filter1)...(filterN))`	Boolean "AND": all filters are true
`(|(filter1)...(filterN))`	Boolean "OR": any one filter is true

So how do we use these filters? It depends how we are doing the service look-up. If we are using the low-level approach from Section 4.4 then we simply use an alternative signature for the `getServiceReference` method that takes a filter in addition to a service interface name:

```
context.getServiceReference(Mailbox.class.getName(),
   "(&(mailboxName=welcome)(lang=en))");
```

If we are using a service tracker as in Sections 4.6 or 4.8, we need to use an alternative constructor for the `ServiceTracker` class which takes a `Filter` object *instead of* a service interface name. `Filter` objects can be constructed by a call to a static utility method, `FrameworkUtil.createFilter`:

```
Filter filter = FrameworkUtil.createFilter(
   "(&(objectClass=" + Mailbox.class.getName() + ")" +
   "(mailboxName=welcome)(lang=en))");
tracker = new ServiceTracker(context, filter, null);
```

The filter object replaces the service interface name parameter because we can track instances of multiple service types with the same tracker. However when we do wish to restrict the tracker to a single service type, we need to include that constraint in the filter using the built-in property name `objectClass`. In fact, the constructor we were using previously is simply a convenience wrapper for the filter-based constructor.

While we can construct filters easily using string concatenation as above, it can be somewhat error prone — it is all too easy to miss closing a parenthesis. Therefore it's a good idea to use the `String.format` method which was introduced in Java 5. This method uses a string pattern in the style of the standard `printf` function in the C programming language, so we can construct a filter as shown in Listing 4.13. Each `%s` in the format string is a *format specifier* which is replaced by one argument from the argument list. The resulting code is slightly longer, but it is much harder to make a mistake in the syntax, and it is easier to refer to constants defined elsewhere.

Listing 4.13 Building Filter Strings using `String.format`

```
context.createFilter(String.format("(&(%s=%s)(%s=%s)(%s=%s))",
      Constants.OBJECTCLASS, Mailbox.class.getName(),
      Mailbox.NAME_PROPERTY, "welcome",
      "lang", "en*"));
```

Unfortunately the `createFilter` method of `FrameworkUtil` can throw an `InvalidSyntaxException`, which is a checked exception, so we must always

handle it even when our filter expression is hard-coded and we know that the exception cannot possibly occur. But this is a general problem with checked exceptions in Java.

4.10. Cardinality and Selection Rules

Sometimes, no matter how much filtering we apply when looking up a service, we cannot avoid matching against multiple services. Since any bundle is free to register services with any set of properties, there is no way to force each service to have a unique set of properties. An exception is the service ID property, which is supplied by the framework and guaranteed to be unique, but since we cannot know in advance what the ID of a service will be, it is not generally useful to filter on it.

Therefore the cardinality of all service look-ups in OSGi is implicitly "zero to many". So what do we do if we prefer to simply have one?

Here's an example: OSGi provides a standard service for sending logging messages to the system-wide log. To write messages to the log, a component can obtain an instance of the `LogService` service and call the `log` method therein. However, we would prefer to write to just one log. What do we do when there are many?

Looking back at the examples from Sections 4.4 and 4.6, it seems we don't have to worry about this after all. The `getServiceReference` method on `BundleContext` and the `getService` method on `ServiceTracker` both return either a *single* service, or `null` if no matching services are available. They do this by applying two simple rules. The first is to look at a special `service.ranking` property on the services, which can be referenced in code as `Constants.SERVICE_RANKING`. The value of the property is an integer between `Integer.MIN_VALUE` (i.e. -2,147,483,648) and `Integer.MAX_VALUE` (2,147,483,647). The service with the highest ranking is selected — services that do not have an explicit ranking property take the implicit value of zero. If this rule produces a tie, then the service with the lowest service ID is selected. The second rule is somewhat arbitrary, but it tends to result in the "oldest" service being selected, since in most framework implementation service IDs are allocated from an incrementing counter (although this behaviour is not part of the specification and cannot be relied upon).

Concerns about cardinality are generally separated into the *minimum* number of instances required and the *maximum* instances required. If the minimum is zero then the service cardinality is optional; if the minimum is one then the service cardinality is mandatory. If the maximum is one then the cardinality is unary; if the maximum is many (i.e. there is no maximum) then the cardinality is multiple. We can refer to any of the four possible combinations as follows:

`0..1`	Optional and unary
`0..n`	Optional and multiple
`1..1`	Mandatory and unary
`1..n`	Mandatory and multiple

Let's look at some typical usage patterns.

4.10.1. Optional, Unary

This is the simplest case: we wish to use a particular service if it is available, but don't mind if it is not available. Also, if there are many instances, we don't mind which one is used. A typical example of this is the standard log service mentioned above.

The normal pattern for this is the code from Section 4.6: calling `getService` against a simple (un-customized) service tracker when needed.

Another useful pattern is to sub-class `ServiceTracker` and implement the service interface directly. Listing 4.14 shows an example of this idea. Here we have an implementation of `LogService` that calls an underlying service in the registry each time a `log` method is called, if such a service exists; otherwise the log message is silently thrown away. This pattern can be useful when we want let an object use the service interface without making it aware of the the details of service management. Listing 4.15 shows a simple example, where the `DatababaseConnection` class simply requires an instance of `LogService` and does not know anything about service tracking.

4.10.2. Optional, Multiple

In this case we wish to use *all* instances of a service if any are available, but don't mind if none are available. This style is commonly used for notifications or event handlers: all of the handlers should be notified of an event, but the originator of the event doesn't care if there are no handlers. We sometimes refer to this the *Whiteboard Pattern* and we will discuss it at length in Chapter 7.

The normal implementation pattern is almost the same as for optional-unary, simply replacing the call to the `getService` method on `ServiceTracker` with a call to `getServices`. This returns an array containing all of the matching services that are currently registered. Beware that, as in other parts of the OSGi API, `getServices` returns `null` to signify no matches rather than an empty array, so we must always perform a `null`-check before using the result.

Listing 4.16 shows a variant of the `LogTracker` from the previous section. This `MultiLogTracker` will pass on logging messages to *all* available log services... as before, messages will be silently thrown away if there are no logs.

Listing 4.14 Log Tracker

```
1  package org.osgi.book.utils;

3  import org.osgi.framework.BundleContext;
4  import org.osgi.framework.ServiceReference;
5  import org.osgi.service.log.LogService;
6  import org.osgi.util.tracker.ServiceTracker;

8  public class LogTracker extends ServiceTracker implements LogService {

10    public LogTracker(BundleContext context) {
11       super(context, LogService.class.getName(), null);
12    }

14    public void log(int level, String message) {
15       log(null, level, message, null);
16    }

18    public void log(int level, String message, Throwable exception) {
19       log(null, level, message, exception);
20    }

22    public void log(ServiceReference sr, int level, String message) {
23       log(sr, level, message, null);
24    }

26    public void log(ServiceReference sr, int level, String message,
27          Throwable exception) {
28       LogService log = (LogService) getService();
29       if (log != null) {
30          log.log(sr, level, message, exception);
31       }
32    }
33 }
```

Listing 4.15 Sample Usage of Log Tracker

```
1  package org.osgi.book.utils.sample;
2
3  import org.osgi.service.log.LogService;
4
5  public class DbConnection {
6
7     private final LogService log;
8
9     public DbConnection(LogService log) {
10       this.log = log;
11       log.log(LogService.LOG_INFO, "Opening connection");
12       // ...
13    }
14
15    public void disconnect() {
16       log.log(LogService.LOG_INFO, "Disconnecting");
17       // ...
18    }
19 }
```

```
1  package org.osgi.book.utils.sample;
2
3  import org.osgi.book.utils.LogTracker;
4  import org.osgi.framework.BundleActivator;
5  import org.osgi.framework.BundleContext;
6
7  public class DbConnectionActivator implements BundleActivator {
8
9     private LogTracker log;
10    private DbConnection dbconn;
11
12    public void start(BundleContext context) throws Exception {
13       log = new LogTracker(context);
14       log.open();
15
16       dbconn = new DbConnection(log);
17    }
18
19    public void stop(BundleContext context) throws Exception {
20       dbconn.disconnect();
21       log.close();
22    }
23 }
```

Listing 4.16 Multi Log Tracker

```
1  package org.osgi.book.utils;
2
3  import org.osgi.framework.BundleContext;
4  import org.osgi.framework.ServiceReference;
5  import org.osgi.service.log.LogService;
6  import org.osgi.util.tracker.ServiceTracker;
7
8  public class MultiLogTracker extends ServiceTracker
9      implements LogService {
10
11     public MultiLogTracker(BundleContext context) {
12         super(context, LogService.class.getName(), null);
13     }
14
15     public void log(int level, String message) {
16         log(null, level, message, null);
17     }
18
19     public void log(int level, String message, Throwable exception) {
20         log(null, level, message, exception);
21     }
22
23     public void log(ServiceReference sr, int level, String message) {
24         log(sr, level, message, null);
25     }
26
27     public void log(ServiceReference sr, int level, String message,
28             Throwable exception) {
29         Object[] logs = getServices();
30         if (logs != null) {
31             for (Object log : logs) {
32                 ((LogService) log).log(sr, level, message, exception);
33             }
34         }
35     }
36 }
```

4.10.3. Mandatory, Unary

TODO

4.10.4. Mandatory, Multiple

TODO

4.11. Service Factories

Sometimes a service needs to know something about who its consumer is. A good example is a logging service: when a bundle sends a message to the logging service, then one useful piece of information to include in the recorded log is the identity of the bundle that generated the message. We could add a parameter to the methods of the logging service which allowed the caller to state its identity, but it would be very inconvenient for the caller to supply this information every time. Also, callers could supply incorrect information – either accidentally or maliciously – which would make the log unreliable. It would be useful if we would get the service registry to provide us this information. After all, it always knows which services are being accessed by which bundles.

The service registry does indeed provide a way to access this information: service factories.

When we register a new service, we usually provide the service object itself in our call to `BundleContext.registerService`. However we can choose instead to register a *factory* object. The factory does not itself implement the service interface, but it knows how to create an object that does. The service registry will call the factory's creation method when some bundle requests an instance of the service.

A service factory must implement the interface `ServiceFactory` from the core OSGi APIs, which has the interface shown in Listing 4.17.

We implement the `getService` method to provide the instance of the real service object. The service registry gives us two useful pieces of information: the identity of the calling bundle, and a reference to the `ServiceRegistration` object that was created when the service was registered. We can also implement the `ungetService` method to clean up when a particular bundle is no longer using the service (although *other* bundles may still be using it). In `ungetService` we receive the same information, and also a reference to the service object that we returned from `getService`.

Listing 4.17 Summary of the `ServiceFactory` interface

```
1 public interface ServiceFactory {

3    public Object getService(Bundle bundle,
4          ServiceRegistration registration);

6    public void ungetService(Bundle bundle,
7          ServiceRegistration registration, Object service);

9 }
```

Listing 4.18 shows an example of a service factory that makes use of the reference to the calling bundle. The `LogImpl` class implements a `Log` interface and takes a `Bundle` object in its constructor. When a message is sent to the log, it is prefixed with the symbolic name of the calling bundle and then printed to the console. The service factory implementation creates a new instance of this class for each calling bundle; the `ungetService` method is empty because no special clean-up is required, we can simply allow the `LogImpl` instance to be garbage-collected. The activator registers an instance of the `LogServiceFactory` class in the normal way, as if it were an instance of the "real" `Log` class.

In this example we did not use the `ServiceRegistration` reference, and it is not so common to do so. It can be useful though when a single factory object instance is used to create instances for multiple services. In that case the `ServiceRegistration` reference can be checked to work out which specific service object to create.

Note that from the consumer bundle's point of view, the use of a service factory by the provider is completely transparent. It is in fact impossible for a consumer bundle to find out whether a service is implemented as a "normal" service or using a service factory.

Another interesting use of service factories, beyond customising services to their consumers, is they allow us to delay the creation of service objects until they are actually needed. Suppose that we have a service object which is, for whatever reason, expensive or slow to create: if we register it in the normal way then we must pay the price of creating the service even if no consumer ever uses it. If we use a factory then we can defer creation until some consumer actually requests the service.

Listing 4.18 A Service Factory and Activator

```
public interface Log {
   void log(String message);
}
```

```
package org.osgi.book.log;

import org.osgi.framework.Bundle;
import org.osgi.framework.ServiceFactory;
import org.osgi.framework.ServiceRegistration;

class LogImpl implements Log {

   private String sourceBundleName;

   public LogImpl(Bundle bundle) {
      this.sourceBundleName = bundle.getSymbolicName();
   }

   public void log(String message) {
      System.out.println(sourceBundleName + ": " + message);
   }
}

public class LogServiceFactory implements ServiceFactory {

   public Object getService(Bundle bundle,
         ServiceRegistration registration) {
      return new LogImpl(bundle);
   }

   public void ungetService(Bundle bundle,
         ServiceRegistration registration, Object service) {
      // No special clean-up required
   }
}
```

```
package org.osgi.book.log;

import org.osgi.framework.BundleActivator;
import org.osgi.framework.BundleContext;

public class LogServiceFactoryActivator implements BundleActivator {

   public void start(BundleContext context) throws Exception {
      context.registerService(Log.class.getName(),
            new LogServiceFactory(), null);
   }

   public void stop(BundleContext context) throws Exception {
   }
}
```

5. Example: Mailbox Reader GUI

We now have the equipment we need to embark on our task to build a GUI mailbox reader application, as first described in Section 3.1. Let's first think about the design.

The GUI should be simple and built on Java Swing. The central component will be split vertically: in the top part there will be a tabbed pane showing one tab per mailbox, and the bottom part will show the content of the selected message. Each mailbox will be displayed as a table with one row per message.

5.1. The Mailbox Table Model and Panel

We will approach the implementation of this application by building from the bottom up. First, we consider the table of messages that represents an individual mailbox. If we use the Swing `JTable` class then we need a "model" for the content of the table. The easiest way to provide this is to override the `AbstractTableModel` class and provide the few remaining abstract methods as shown in Listing 5.1. Note that this class is pure Swing code and does not illustrate any OSGi concepts, so it can be briefly skimmed if you prefer.

Given the table model class, we can now create a "panel" class that encapsulates the creation of the model, the table and a "scroll pane" to contain the table and provide it with scroll bars. This is shown in Listing 5.2.

5.2. The Mailbox Tracker

Next we will build the tracker that tracks the mailbox services and creates tables to display them. As this tracker class will be a little complex, we will build it up incrementally over the course of this section. In Listing 5.3 we simply define the class and its constructor.

Here we pass the `BundleContext` to the superclass's constructor, along with the fixed name of the service that we will be tracking, i.e. `Mailbox`. In the constructor of this tracker we also expect to receive an instance of `JTabbedPane`, which is the Swing class representing tab panels. We need this because we will

Listing 5.1 The Mailbox Table Model

```
1  package org.osgi.book.reader.gui;
2
3  import java.util.ArrayList;
4  import java.util.List;
5
6  import javax.swing.table.AbstractTableModel;
7
8  import org.osgi.book.reader.api.Mailbox;
9  import org.osgi.book.reader.api.MailboxException;
10 import org.osgi.book.reader.api.Message;
11
12 public class MailboxTableModel extends AbstractTableModel {
13
14    private static final String ERROR = "ERROR";
15
16    private final Mailbox mailbox;
17    private final List<Message> messages;
18
19    public MailboxTableModel(Mailbox mailbox) throws MailboxException {
20       this.mailbox = mailbox;
21       long[] messageIds = mailbox.getAllMessages();
22       messages = new ArrayList<Message>(messageIds.length);
23       Message[] messageArray = mailbox.getMessages(messageIds);
24       for (Message message : messageArray) {
25          messages.add(message);
26       }
27    }
28
29    public synchronized int getRowCount() {
30       return messages.size();
31    }
32
33    public int getColumnCount() {
34       return 2;
35    }
36
37    @Override
38    public String getColumnName(int column) {
39       switch (column) {
40       case 0:
41          return "ID";
42       case 1:
43          return "Subject";
44       }
45       return ERROR;
46    }
47
48    public synchronized Object getValueAt(int row, int column) {
49       Message message = messages.get(row);
50       switch (column) {
51       case 0:
52          return Long.toString(message.getId());
53       case 1:
54          return message.getSummary();
55       }
56       return ERROR;
57    }
58 }
```

Listing 5.2 Mailbox Panel

```
1  package org.osgi.book.reader.gui;

3  import javax.swing.JPanel;
4  import javax.swing.JScrollPane;
5  import javax.swing.JTable;

7  import org.osgi.book.reader.api.Mailbox;
8  import org.osgi.book.reader.api.MailboxException;

10 public class MailboxPanel extends JPanel {

12    private final MailboxTableModel tableModel;

14    public MailboxPanel(Mailbox mbox) throws MailboxException {
15       tableModel = new MailboxTableModel(mbox);
16       JTable table = new JTable(tableModel);
17       JScrollPane scrollPane = new JScrollPane(table);

19       add(scrollPane);
20    }
21 }
```

Listing 5.3 The Mailbox Tracker, Step One: Constructor

```
1  package org.osgi.book.reader.gui;

3  // ... import statements omitted ...


1  public class ScannerMailboxTracker extends ServiceTracker {

3     private final JTabbedPane tabbedPane;

5     public ScannerMailboxTracker(BundleContext ctx,
6           JTabbedPane tabbedPane) {
7        super(ctx, Mailbox.class.getName(), null);
8        this.tabbedPane = tabbedPane;
9     }

11    // ...
```

dynamically add tabs when each mailbox is registered and remove them when the corresponding mailbox is unregistered.

Now consider the `addingService` method. What we want to do is create a `MailboxPanel` from the mailbox, and add it to the tabbed panel. This will create a new tab. Finally, we would like to return the mailbox panel from `addingService` so that the tracker will give it back to us in `removedService`, because we will need it in order to remove the corresponding tab.

Sadly, things are not quite that simple. Like most modern GUI libraries, Swing is single-threaded, meaning we must always call its methods from a specific thread. But in OSGi, we cannot be sure which thread we are in when `addingService` is called. So instead of calling Swing directly, we must create a `Runnable` and pass it to the Swing API via `SwingUtilities.invokeLater`. This will cause the code block to be run on the correct thread, but it will happen at some unknown time in the future. Probably it will run *after* we have returned from our `addingService` method, so we cannot return the mailbox panel object directly since it may not exist yet.

One solution is to wrap the panel object in a *future*. This is a kind of "suspension": a future represents the result of an asynchronous computation that may or may not have completed yet. In Java it is represented by the `java.util.concurrent.Future` interface, which is part of the new concurrency API introduced in Java 5. The `FutureTask` class (in the same package) is an implementation of `Future`, and it also implements `Runnable`, allowing it to be executed by a call to the Swing `invokeLater` utility method. Therefore we can write the `addingService` method as shown in Listing 5.4, returning an object of type `Future<MailboxPanel>` instead of the `MailboxPanel` directly.

Let's examine how this method works step-by-step. The first two lines simply retrieve the mailbox object and its name. The bulk of the method constructs a `Callable` object that implements the computation we wish to perform in the GUI thread. This computation creates the mailbox panel, adds it to the tabbed panel (using the mailbox name as the tab title), and finally returns it. Returning a result from the `Callable` sets the value of the `Future`. Finally, we wrap the computation in a `FutureTask` and pass it to the Swing `invokeLater` method for execution by the GUI thread.

The `removedService` method is now quite easy: see Listing 5.5. Again we use an `invokeLater` call to update the GUI by pulling the panel out of its `Future` wrapper and removing it from the tabbed panel.

5.3. The Main Window

Now let's look at the class that defines the main window frame, creates the tabbed panel, and uses the mailbox tracker. See Listing 5.6.

Listing 5.4 The Mailbox Tracker, Step Two: `addingService` method

```
1     @Override
2     public Object addingService(ServiceReference reference) {
3        final String mboxName =
4           (String) reference.getProperty(Mailbox.NAME_PROPERTY);
5        final Mailbox mbox = (Mailbox) context.getService(reference);

7        Callable<MailboxPanel> callable = new Callable<MailboxPanel>() {
8           public MailboxPanel call() {
9              MailboxPanel panel;
10             try {
11                panel = new MailboxPanel(mbox);
12                String title = (mboxName != null) ?
13                      mboxName : "<unknown>";
14                tabbedPane.addTab(title, panel);
15             } catch (MailboxException e) {
16                JOptionPane.showMessageDialog(tabbedPane, e.getMessage(),
17                            "Error", JOptionPane.ERROR_MESSAGE);
18                panel = null;
19             }
20             return panel;
21          }
22       };
23       FutureTask<MailboxPanel> future =
24          new FutureTask<MailboxPanel>(callable);
25       SwingUtilities.invokeLater(future);

27       return future;
28    }
```

Listing 5.5 The Mailbox Tracker, Step Three: `removedService` method

```
1     @Override
2     public void removedService(ServiceReference reference, Object svc) {

4        @SuppressWarnings("unchecked")
5        final Future<MailboxPanel> panelRef = (Future<MailboxPanel>) svc;

7        SwingUtilities.invokeLater(new Runnable() {
8           public void run() {
9              try {
10                MailboxPanel panel = panelRef.get();
11                if (panel != null) {
12                   tabbedPane.remove(panel);
13                }
14             } catch (ExecutionException e) {
15                // The MailboxPanel was not successfully created
16             } catch (InterruptedException e) {
17                // Restore interruption status
18                Thread.currentThread().interrupt();
19             }
20          }
21       });

23       context.ungetService(reference);
24    }
25 }
```

Listing 5.6 The Mailbox Reader Main Window

```
1  package org.osgi.book.reader.gui;
2
3  import java.awt.BorderLayout;
4  import java.awt.Component;
5  import java.awt.Dimension;
6
7  import javax.swing.JFrame;
8  import javax.swing.JLabel;
9  import javax.swing.JTabbedPane;
10 import javax.swing.SwingConstants;
11
12 import org.osgi.framework.BundleContext;
13
14 public class ScannerFrame extends JFrame {
15
16    private JTabbedPane tabbedPane;
17    private ScannerMailboxTracker tracker;
18
19    public ScannerFrame() {
20       super("Mailbox Scanner");
21
22       tabbedPane = new JTabbedPane();
23       tabbedPane.addTab("Mailboxes", createIntroPanel());
24       tabbedPane.setPreferredSize(new Dimension(400, 400));
25
26       getContentPane().add(tabbedPane, BorderLayout.CENTER);
27    }
28
29
30    private Component createIntroPanel() {
31       JLabel label = new JLabel("Select a Mailbox");
32       label.setHorizontalAlignment(SwingConstants.CENTER);
33       return label;
34    }
35
36
37    protected void openTracking(BundleContext context) {
38       tracker = new ScannerMailboxTracker(context, tabbedPane);
39       tracker.open();
40    }
41
42    protected void closeTracking() {
43       tracker.close();
44    }
45 }
```

This class is also simple. In the usual way when working with Swing we subclass the `JFrame` class, which defines top-level "shell" windows. In the constructor we create the contents of the window, which at this stage is just the tabbed panel. Unfortunately tabbed panels can look a little strange when they have zero tabs, so we add a single fixed tab containing a hard-coded instruction label. In a more sophisticated version, we might wish to hide this placeholder tab when there is at least one mailbox tab showing, and re-show it if the number of mailboxes subsequently drops back to zero.

In addition to the constructor and the `createIntroPanel` utility method, we define two protected methods for managing the mailbox service tracker: `openTracking`, which creates and opens the tracker; and `closeTracking` which closes it. These methods are protected because they will only be called from the same package — specifically, they will be called by the bundle activator, which we will look at next.

5.4. The Bundle Activator

To actually execute the above code in a conventional Java Swing application, we would create a "Main" class with a `public static void main` method, as shown in Listing 5.7, and launch it by naming that class on the command line. Notice the window listener, which we need add to make sure that the Java runtime shuts down when the window is closed by the user. If this were not supplied, then the JVM would continue running even with no windows visible.

Listing 5.7 Conventional Java Approach to Launching a Swing Application

```
1  package org.osgi.book.reader.gui;

3  import java.awt.event.WindowAdapter;
4  import java.awt.event.WindowEvent;

6  public class ScannerMain {

8      public static void main(String[] args) {
9          ScannerFrame frame = new ScannerFrame();
10         frame.pack();

12         frame.addWindowListener(new WindowAdapter() {
13             public void windowClosing(WindowEvent e) {
14                 System.exit(0);
15             }
16         });

18         frame.setVisible(true);
19     }

21 }
```

However in OSGi, we don't need to start and stop the whole JVM. We can use the lifecycle of a bundle to create and destroy the GUI, possibly many times during the life of a single JVM. Naturally we do this by implementing a bundle activator, and the code in Listing 5.8 demonstrates this idea.

Listing 5.8 Using Bundle Lifecycle to Launch a Swing Application

```
1  package org.osgi.book.reader.gui;
2
3  import java.awt.event.WindowAdapter;
4  import java.awt.event.WindowEvent;
5
6  import javax.swing.UIManager;
7
8  import org.osgi.framework.BundleActivator;
9  import org.osgi.framework.BundleContext;
10 import org.osgi.framework.BundleException;
11
12 public class ScannerFrameActivator implements BundleActivator {
13
14    private ScannerFrame frame;
15
16    public void start(final BundleContext context) throws Exception {
17       UIManager.setLookAndFeel(
18             UIManager.getSystemLookAndFeelClassName());
19       frame = new ScannerFrame();
20       frame.pack();
21
22       frame.addWindowListener(new WindowAdapter() {
23          public void windowClosing(WindowEvent e) {
24             try {
25                context.getBundle().stop();
26             } catch (BundleException e1) {
27                // Ignore
28             }
29          }
30       });
31
32       frame.openTracking(context);
33
34       frame.setVisible(true);
35    }
36
37    public void stop(BundleContext context) throws Exception {
38       frame.setVisible(false);
39
40       frame.closeTracking();
41    }
42
43 }
```

When the bundle is started, we create a frame and open it by making it visible. We also tell the frame to start tracking mailbox services, passing in our bundle context. When the bundle is stopped, we hide the frame and turn off the tracker.

Notice that we again registered a window listener. This time, the listener just stops the bundle rather than stopping the entire Java runtime. The `getBundle` method on `BundleContext` returns the bundle's *own* handle object, and

bundles are free to manipulate themselves via the methods of this handle in the same way that they can manipulate other bundles. This time it is not really essential to register the listener, as it was with the standalone program, but to do so creates a pleasingly symmetry. If stopping the bundle closes the window, shouldn't closing the window stop the bundle? The lifecycles of the two are thus linked. If the user closes the window, then we can see that later by looking at the bundle state, and reopen the window by starting the bundle again.

5.5. Putting it Together

We're ready to build and run the application. Let's look at the `bnd` descriptor for the GUI application, as shown in Listing 5.9. This descriptor simple instructs `bnd` to bundle together all of the classes in the package `org.osgi.book.reader.gui` — in other words, all the classes we have seen in this chapter — and it also specifies the activator in the normal way.

Listing 5.9 Bnd Descriptor for the Mailbox Scanner

```
# mailbox_gui.bnd
Private-Package: org.osgi.book.reader.gui
Bundle-Activator: org.osgi.book.reader.gui.ScannerFrameActivator
```

We can build this bundle by running:

```
java -jar path/to/bnd.jar mailbox_gui.bnd
```

And then we can install it into Equinox and start it as follows:

```
osgi> install file:mailbox_gui.jar
Bundle id is 6

osgi> start 6

osgi>
```

The `mailbox_api` bundle must also be installed, otherwise `mailbox_gui` will not resolve, and will therefore return an error when you attempt to start it. However assuming the bundle starts successfully, a Swing window should open, which looks like Figure 5.1.

Note that there are no mailboxes displayed; this is because there are no `Mailbox` services present. To view a mailbox we can install the `welcome_mailbox` bundle from Chapter 4:

```
osgi> install file:welcome_mailbox.jar
Bundle id is 7

osgi> start 7
```

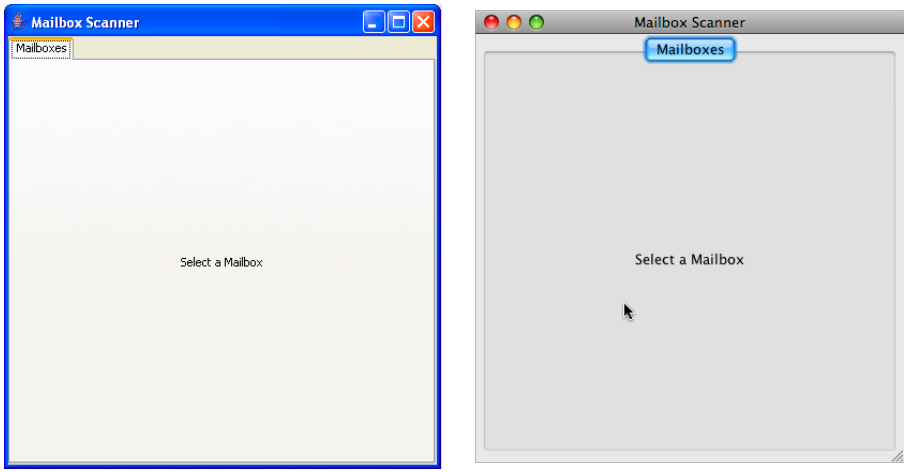


Figure 5.1.: The Mailbox GUI (Windows XP and Mac OS X)

Now a tab should appear with the label "welcome". If we select that tab, the list of messages in the mailbox will be displayed as in Figure 5.2 (from now on, only the Windows XP screenshot will be shown).

It's worth stopping at this point to check that the application behaves as we expect it to. Specifically:

1. If we stop the `fixed_mailbox` bundle, the "welcome" tab will disappear; and if we start it again, the tab will come back.
2. If we stop the `mailbox_gui` bundle, the window will close. Starting again will re-open the window with the same set of tabs as before.
3. If we close the window, the `mailbox_gui` bundle will return to RESOLVED state (use the `ps` command to check). Again, restarting the bundle will open the window in its prior state.

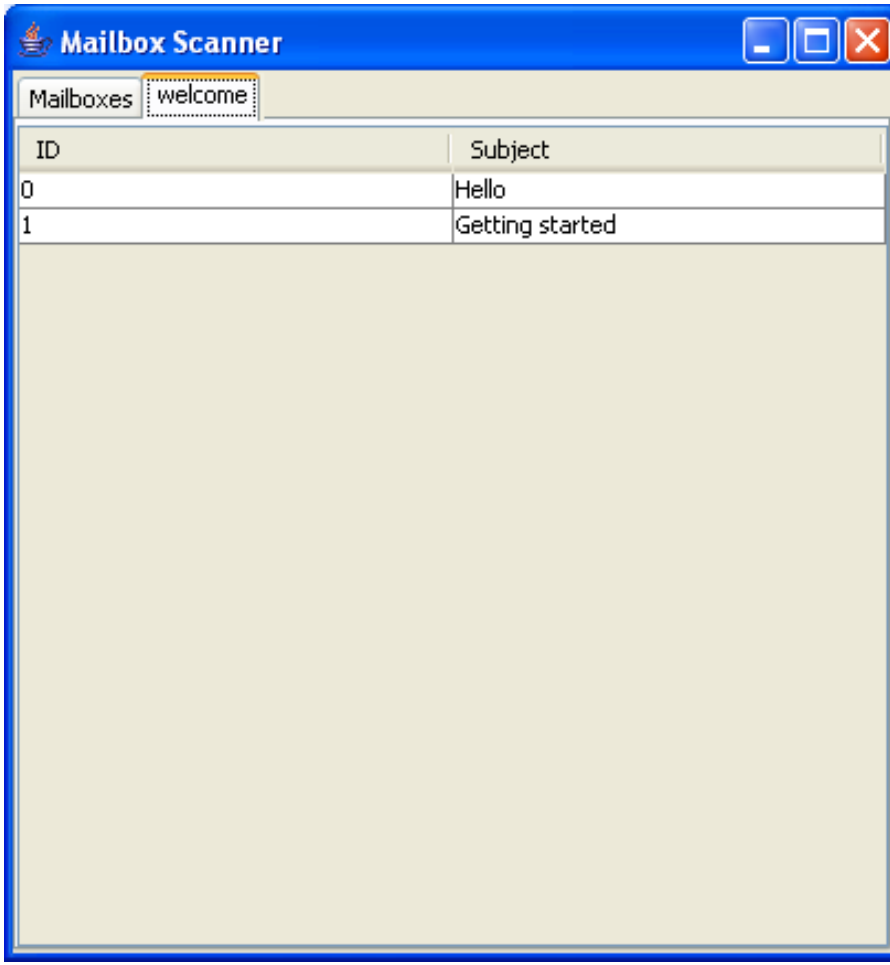


Figure 5.2.: The Mailbox GUI with a Mailbox Selected

6. Concurrency and OSGi

We have touched lightly on some of the issues involved with concurrency and multi-threading in OSGi. Now it's time to look at them seriously.

Unlike heavyweight frameworks such as J2EE, OSGi does not attempt to take control of all the resources of the Java Virtual Machine, and that includes threads: whereas J2EE forbids you from writing code that creates threads or uses explicit synchronization, offering instead a cumbersome and limited "work management" framework, OSGi simply allows you to control the creation and scheduling of threads in your application yourself, as you would in any ordinary application. To support this the OSGi libraries are thread safe and can be called from any thread.

However, this freedom comes at a price. Just as we are free to create threads in our bundles, any other bundle has that freedom also. We can never assume our bundle lives in a single-threaded environment, even if we avoid using threads ourselves: OSGi is implicitly multi-threaded. Therefore we must ensure that our code is appropriately thread-safe, particularly when receiving events or callbacks from the framework or from other bundles.

This chapter is an introduction to the concepts of concurrency in Java as applied to OSGi. For a more thorough treatment of Java concurrency, "Java Concurrency in Practice" by Brian Goetz et al [**?**] is invaluable and, in the author's opinion, should be kept close at hand by all professional Java programmers.

6.1. The Price of Freedom

Let's follow through a simple scenario involving the imaginary bundles A, B and C, which is illustrated in the style of a UML sequence diagram in Figure 6.1. Suppose bundle A starts a thread and at some point it obtains a handle to bundle B and calls the `start` method from that thread. This will cause bundle B to activate, and the `start` method of B's activator will be invoked in the thread created by A. Furthermore, suppose that B registers a service during its `start` method and C happens to be listening with a service tracker for instances of that service type. The `addingService` method of C's tracker will be called, also from the thread created by A. Finally, suppose C creates

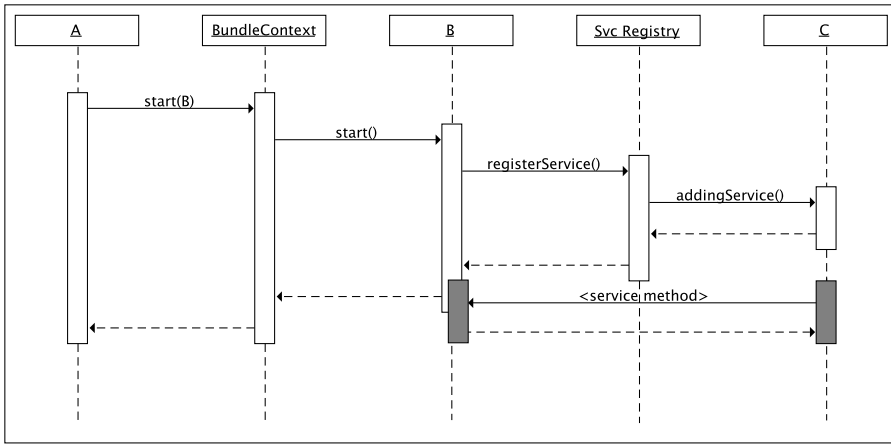


Figure 6.1.: Framework Calls and Callbacks in OSGi

a polling thread which periodically calls the service registered by B. The methods of B's service will be executed in a thread created by C.

When a client gets a service from the service registry, it sees the real service object, not any kind of proxy or wrapper. Therefore when the client invokes methods on the service, those invocations are standard synchronous method calls, which means that the service method executes on a thread which is "owned" by the client bundle.

Furthermore, many notifications in OSGi (but not all) happen *synchronously*. That is, when the framework is called with a method that generates callbacks — such as sending a `ServiceEvent` to registered service listeners, or calling a bundle activator's `start` or `stop` methods — those callbacks are executed in the same thread, and must complete before control is returned to the caller of the framework method.

The above has three major implications:

- Callbacks and service methods can and will be invoked from *any arbitrary thread*, perhaps even from many threads at the same time. Our code will fail in unpredictable ways unless we are conscious of this and allow for it.
- The thread that our callbacks and service methods are invoked from does not "belong" to us. If we execute long-running operations or blocking I/O calls directly from these callbacks then we can delay the entire system.
- When we call an OSGi API method we can't predict the set of callbacks and listeners that will be called by the framework as a result, and therefore what locks those callbacks will attempt to take. If we hold other

locks while calling such methods, we risk causing a deadlock.

The only solution to these problems is to use good concurrent programming practices. However, safe concurrency is really not that difficult, at least in the sense of being intellectually challenging, like quantum physics or a game of Go. One does not need to be a genius to do it correctly. The key is *discipline* — something that many geniuses lack! As long as we consistently apply a few simple rules, we can easily handle most situations we encounter.

1. Immutable objects are automatically thread-safe, and objects that are not shared across threads do not need to be thread-safe. Therefore share as little as possible and favour immutability wherever possible.
2. When objects really must be shared and mutable, guard all accesses (both read *and* write) to shared fields with a lock on the same object, or make appropriate use of volatile variables.
3. Avoid acquiring new locks when holding an existing lock. As a direct consequence of this rule, we must avoid holding any locks when calling unknown or "foreign" code that might attempt to acquire a lock. This includes calls to services or to OSGi APIs, many of which can result in callbacks to other bundles that execute in *our* thread.

6.2. Shared Mutable State

We can substantially reduce the size of the concurrency problem simply by sharing as few objects as possible, and making as many shared objects as possible immutable. Unfortunately it's generally not possible to reduce the problem size to zero by these methods. In most real-world applications we cannot completely avoid sharing mutable state, so we need to find a way to do it safely.

Let's look at some example code based on services. Suppose we wish to maintain a `Map` of registered mailbox services, keyed by name, and we wish to offer a public method `getMailboxByName` which will return the specified mailbox, or `null` if none such currently exists. The sample code in Listing 6.1 uses traditional Java synchronized blocks to achieve thread safety. This class is well behaved because it follows the rules: all accesses to the map, *including* simple read operations, are made from the context of a lock on the same object. Although the map field itself is final, the *content* of the map is mutable, and it is shared across threads, so it needs to be protected by a lock. Also, the synchronized blocks are as short as possible, quickly releasing the lock when it is no longer required.

However, the code in Listing 6.1 does not take any advantage of the new concurrency features introduced in Java 5. Using those features, we can do slightly

Listing 6.1 Thread Safety using Synchronized Blocks

```
1  package org.osgi.book.reader.mailboxmap;

3  import java.util.HashMap;
4  import java.util.Map;

6  import org.osgi.book.reader.api.Mailbox;
7  import org.osgi.framework.BundleActivator;
8  import org.osgi.framework.BundleContext;
9  import org.osgi.framework.ServiceReference;
10 import org.osgi.util.tracker.ServiceTracker;

12 public class MailboxMapActivator1 implements BundleActivator {

14    private final Map map = new HashMap();
15    private ServiceTracker tracker;

17    public void start(BundleContext context) throws Exception {
18       tracker = new MapTracker(context);
19       tracker.open();
20    }
21    public void stop(BundleContext context) throws Exception {
22       tracker.close();
23    }

25    public Mailbox getMailboxByName(String name) {
26       synchronized (map) {
27          return (Mailbox) map.get(name);
28       }
29    }

31    private class MapTracker extends ServiceTracker {

33       public MapTracker(BundleContext context) {
34          super(context, Mailbox.class.getName(), null);
35       }

37       public Object addingService(ServiceReference reference) {
38          String mboxName = (String) reference
39                .getProperty(Mailbox.NAME_PROPERTY);
40          Mailbox mbox = (Mailbox) context.getService(reference);
41          synchronized (map) {
42             map.put(mboxName, mbox);
43          }
44          return mboxName;
45       }

47       public void removedService(ServiceReference reference,
48             Object service) {
49          String mboxName = (String) service;
50          synchronized (map) {
51             map.remove(mboxName);
52          }
53       }
54    }
55 }
```

better thanks to the observation that many threads should be able to call the `getMailboxByName` method at the same time. Traditional synchronized blocks cannot distinguish between read operations and write operations: they ensure *all* operations inside them have exclusive access to the lock object. However Java 5 introduced Read/Write locks that do make this distinction. Listing 6.2 uses this feature to allow many threads to read from the map concurrently, while still preventing concurrent updates to the map:

If Java 5 (or above) is available to you, then it is well worth investigating whether the new concurrency library therein will be of benefit. However even if you are limited to Java 1.4 or earlier, you should not fear the synchronized block. Many developers avoid synchronization because of problems related to its performance, but it is really not that bad, especially when locks are un-contended — this can usually be achieved by sticking to fine-grained synchronized blocks. There is no sense in sacrificing correct concurrent behaviour for a minor performance gain.

6.3. Safe Publication

Safe publication means making an object available to be accessed by other threads so that those threads do not see the object in an invalid or partially constructed state.

"Publishing" an object entails placing it in a location from which it can be read by another thread. The simplest example of such a location is a public field of an already-published object, but placing the object in a private field also effectively publishes it if there is a public method accessing that field. Unfortunately this mechanism is not enough on its own for the publication to be *safe*.

Listing 6.3 is quoted from [**?**] as an example of unsafe publication. Because of the way modern CPU architectures work, and the way those architectures are exposed to Java programs through the Java Memory Model, a thread reading the `holder` field might see a `null` value or it might see a partially constructed version of the `Holder` object, even if that thread reads the variable *after* `initialize` was called in another thread.

There are four ways to safely publish an object, as listed in [**?**]:

- Initialise an object reference from a static initialiser.
- Store a reference into a `volatile` field or `AtomicReference`.
- Store a reference into a `final` field of a properly constructed object.
- Store a reference into a field that is properly guarded by a lock, i.e, making appropriate use of `synchronized` blocks or methods.

Listing 6.2 Thread Safety using Read/Write Locks

```
1  package org.osgi.book.reader.mailboxmap;
2
3  import java.util.HashMap;
4  import java.util.Map;
5  import java.util.concurrent.locks.ReadWriteLock;
6  import java.util.concurrent.locks.ReentrantReadWriteLock;
7
8  import org.osgi.book.reader.api.Mailbox;
9  import org.osgi.framework.BundleActivator;
10 import org.osgi.framework.BundleContext;
11 import org.osgi.framework.ServiceReference;
12 import org.osgi.util.tracker.ServiceTracker;
13
14 public class MailboxMapActivator2 implements BundleActivator {
15
16    private final Map<String, Mailbox> map =
17          new HashMap<String, Mailbox>();
18    private final ReadWriteLock mapLock = new ReentrantReadWriteLock();
19    private volatile ServiceTracker tracker;
20
21    public void start(BundleContext context) throws Exception {
22       tracker = new MapTracker(context);
23       tracker.open();
24    }
25    public void stop(BundleContext context) throws Exception {
26       tracker.close();
27    }
28    public Mailbox getMailboxByName(String name) {
29       try {
30          mapLock.readLock().lock();
31          return map.get(name);
32       } finally {
33          mapLock.readLock().unlock();
34       }
35    }
36
37    private class MapTracker extends ServiceTracker {
38
39       public MapTracker(BundleContext context) {
40          super(context, Mailbox.class.getName(), null);
41       }
42       public Object addingService(ServiceReference reference) {
43          String mboxName = (String) reference
44                .getProperty(Mailbox.NAME_PROPERTY);
45          Mailbox mbox = (Mailbox) context.getService(reference);
46          try {
47             mapLock.writeLock().lock();
48             map.put(mboxName, mbox);
49          } finally {
50             mapLock.writeLock().unlock();
51          }
52          return mboxName;
53       }
54       public void removedService(ServiceReference reference,
55                                  Object service) {
56          String mboxName = (String) service;
57          try {
58             mapLock.writeLock().lock();
59             map.remove(mboxName);
60          } finally {
61             mapLock.writeLock().unlock();
62          }
63       }
64    }
65 }
```

Listing 6.3 Unsafe Publication

```
// Unsafe publication
public Holder holder;

public void initialize() {
    holder = new Holder(42);
}
```

6.3.1. Safe Publication in Services

Suppose we have a "dictionary" service that allows us to both look up definitions of words and add new definitions. The very simple interface for this service is shown in Listing 6.4 and a trivial (but broken) implementation is in Listing 6.5.

Listing 6.4 Dictionary Service interface

```
1 package org.osgi.book.concurrency;

3 public interface DictionaryService {
4    void addDefinition(String word, String definition);
5    String lookup(String word);
6 }
```

Listing 6.5 Unsafe Publication in a Service

```
1  package org.osgi.book.concurrency;

3  import java.util.HashMap;
4  import java.util.Map;

6  public class UnsafeDictionaryService implements DictionaryService {

8     private Map<String,String> map;

10    public void addDefinition(String word, String definition) {
11       if(map == null) map = new HashMap<String, String>();

13       map.put(word, definition);
14    }

16    public String lookup(String word) {
17       return map == null ? null : map.get(word);
18    }
19 }
```

Recall that a service, once registered, can be called from any thread at any time. The `map` field is only initialised on first use of the `addDefinition` method, but a call to the `lookup` method could see a partially constructed `HashMap` object, with unpredictable results[1].

[1]There are other concurrency problems here too. For example, two threads could enter

The root of the problem here is clearly the late initialisation of the `HashMap` object, which appears to be a classic example of "premature optimisation". It would be easier and safer to create the map during construction of our service object. We could then mark it `final` to ensure it is safely published. This is shown in Listing 6.6. Note that we also have to use a thread-safe `Map` implementation rather than plain `HashMap` — we can get one by calling `Collections.synchronizedMap` but we could also have used a `ConcurrentHashMap`.

Listing 6.6 Safe Publication in a Service

```
1  package org.osgi.book.concurrency;
2
3  import java.util.Collections;
4  import java.util.HashMap;
5  import java.util.Map;
6
7  public class SafeDictionaryService implements DictionaryService {
8
9      private final Map<String,String> map =
10         Collections.synchronizedMap(new HashMap<String,String>());
11
12     public void addDefinition(String word, String definition) {
13         map.put(word, definition);
14     }
15
16     public String lookup(String word) {
17         return map.get(word);
18     }
19 }
```

Using a `final` field is the preferred way to safely publish an object, since it results in code that is easy to reason about, but unfortunately this is not always possible. We may not be able to create the object we want during the construction of our service object if it has a dependency on an object passed by a client of the service. For example, take a look at the service interface in Listing 6.7 and the unsafe implementation in Listing 6.8. The problem is now clear but we cannot fix it by moving initialisation of the `connection` field to the constructor and making it `final`, since it depends on the `DataSource` object passed by a client[2].

A solution to this problem is to simply declare the `connection` field to be `volatile`. This means that any modifications will be automatically visible in

`addDefinition` at about the same time and both see a `null` value for the `map` field, and so they would both create a new `HashMap` and put the word definition into it. But only one `HashMap` instance would ultimately remain, so one of the definitions would be lost.

[2] Note that there are more problems with this service. The `initialise` method is unsafe because two clients could call it at the same time. In fact the service interface itself is poorly designed: suppose a client calls `initialise` and then the service instance it called goes away and is replaced by an alternative — the client would need to track the change and call `initialise` again. For this reason we should avoid designing services that require conversational state, i.e., a controlled series of method calls and responses. Services should ideally be stateless.

Listing 6.7 Connection Cache interface

```
1  package org.osgi.book.concurrency;
2
3  import java.sql.Connection;
4  import java.sql.SQLException;
5
6  import javax.sql.DataSource;
7
8  public interface ConnectionCache {
9     void initialise(DataSource dataSource) throws SQLException;
10    Connection getConnection();
11 }
```

Listing 6.8 Unsafe Connection Cache

```
1  package org.osgi.book.concurrency;
2
3  import java.sql.Connection;
4  import java.sql.SQLException;
5
6  import javax.sql.DataSource;
7
8  public class UnsafeConnectionCache implements ConnectionCache {
9
10    private Connection connection;
11
12    public void initialise(DataSource dataSource) throws SQLException {
13       connection = dataSource.getConnection();
14    }
15
16    public Connection getConnection() {
17       return connection;
18    }
19 }
```

full to any other threads, so we can safely publish an object simply by assigning it into a `volatile` field. Note that this behaviour is only guaranteed on Java 5 and above — developers working with older Java versions will need to use a `synchronized` block or method.

6.3.2. Safe Publication in Framework Callbacks

Given the warnings above, we might expect that the activator class in Listing 6.9 would not be thread-safe. It appears to have a serious problem with unsafe publication of the `LogTracker` object into the `log` field, which is declared neither `final` nor `volatile`.

Listing 6.9 Is This Bundle Activator Thread-safe?

```
1  package org.osgi.book.concurrency;

3  import org.osgi.book.utils.LogTracker;
4  import org.osgi.framework.BundleActivator;
5  import org.osgi.framework.BundleContext;
6  import org.osgi.framework.BundleEvent;
7  import org.osgi.framework.SynchronousBundleListener;
8  import org.osgi.service.log.LogService;

10 public class DubiousBundleActivator implements BundleActivator,
11         SynchronousBundleListener {

13     private LogTracker log;

15     public void start(BundleContext context) throws Exception {
16         log = new LogTracker(context);
17         log.open();

19         context.addBundleListener(this);
20     }

22     public void bundleChanged(BundleEvent event) {
23         if(BundleEvent.INSTALLED == event.getType()) {
24             log.log(LogService.LOG_INFO, "Bundle installed");
25         } else if(BundleEvent.UNINSTALLED == event.getType()) {
26             log.log(LogService.LOG_INFO, "Bundle removed");
27         }
28     }

30     public void stop(BundleContext context) throws Exception {
31         context.removeBundleListener(this);
32         log.close();
33     }
34 }
```

If the problem is not yet apparent, consider that all three methods of this class are technically *callbacks*. The `start` callback happens when a bundle decides to explicitly request the framework to start our bundle, and it runs in the calling thread of that bundle. The `stop` callback likewise happens when a bundle (not necessarily the same one) decides to request the framework to stop our bundle, and it also runs in the calling thread of that bundle. The

`bundleChanged` callback is called whenever any bundle changes its state while we have our activator registered as a listener for bundle events. Therefore all three methods can be called on different threads, so we should use safe publication for the `LogTracker` object accessed from each one.

In fact, it turns out that Listing 6.9 is perfectly thread-safe! But the reasons why it is thread-safe are subtle and require some understanding of how the OSGi framework itself is implemented, and for that reason it is this author's opinion that developers should *use safe publication idioms anyway.*

These two statements clearly need some explanation! First, the reason why Listing 6.9 is thread-safe. One of the ways that we can achieve safe publication, besides `final` and `volatile`, is to use a `synchronized` block or method. In the terminology of the Java Memory Model, we need to ensure that the publication of the `LogTracker` object into the `log` field "*happens-before*" the read operations on that field, and the Java Memory model also guarantees that everything that happens on a thread prior to releasing a lock (i.e., exiting from a `synchronized` block) "*happens-before*" anything that happens on another thread after that thread acquires the *same* lock. So by synchronising on an object before setting the `log` field and before accessing it, we can achieve safe publication of the `LogTracker` object it contains. But crucially, it does not matter *which* object is used to synchronise, so long as the same one is consistently used.

Now, the OSGi framework uses a lot of synchronisation internally — it must, because it is able to offer most of its features safely to multiple threads — and we can use this knowledge to "piggyback" on existing synchronisation in the framework, meaning that in many cases we don't need to add our own safe publication idioms such as `final`, `volatile` or `synchronized`. The trick is knowing when we can get away with this and when we cannot.

So for example the framework holds, somewhere in its internal memory, a list of bundle listeners. When we call `BundleContext.addBundleListener` on line 19, the framework uses a `synchronized` block to add us to the list. Later when a bundle state changes, the framework will use another `synchronized` block to get a snapshot of the currently installed listeners before calling `bundleChanged` on those listeners. Therefore everything we did before calling `addBundleListener` *happens-before* everything we do after being called in `bundleChanged`, and our access of the `log` field in lines 24 and 26 is safe.

Similarly, the framework holds the state of all bundles. When our `start` method completes, the framework changes the state of the bundle from STARTING to ACTIVE, and it uses a `synchronized` block to do that. It is illegal to call `stop` on a bundle that is not ACTIVE (or rather, such a call will be ignored), so the framework will check the state of our bundle (using a `synchronized` block) before allowing our `stop` method to run. Therefore another *happens-before* relationship exists which we can exploit, making our access of

the `log` field safe on line 32 also.

We can express these deduced relationships (plus another that happens to exist) more succinctly as "rules" similar to the built-in rules offered by the Java Memory Model itself:

Bundle Activator rule. Each action in the `start` method of a `BundleActivator` *happens-before* every action in the `stop` method.

Listener registration rule. Registering a listener with the framework — including `ServiceListener`, `BundleListener`, `SynchronousBundleListener` or `FrameworkListener` — *happens-before* any callback is invoked on that listener.

Service registration rule. Registering a service *happens-before* any invocation of the methods of that service by a client.

Unfortunately there are problems with relying on these "rules". First, they are not true rules written down in the OSGi specification such that all conforming implementations must abide by them[3]. They are merely deduced from "circumstantial evidence", i.e., the multi-threadedness of the OSGi framework and our expectation of how that is handled internally by the framework. This raises the possibility that a conforming OSGi implementation may perform locking in a slightly different way, so violating these rules.

But the bigger problem is that piggybacking on the framework's synchronisation is a very advanced technique. It just happens that we can write "dumb" code such as the code in Listing 6.9 and find that it works... but we must be very aware of when our license runs out and we must start caring about safe publication again. To do this, we need an intimate understanding of internal framework issues. It seems it would be easier just to use safe publication idioms everywhere. In this example, that simply means adding the `volatile` keyword to the `log` field.

6.4. Don't Hold Locks when Calling Foreign Code

Suppose we have a service that implements the `MailboxRegistrationService` interface shown in Listing 6.10.

When the `registerMailbox` method is called, we should register the supplied mailbox as a service with the specified name. Also we should unregister any previous mailbox service that we may have registered with that same name.

Listing 6.11 shows a naïve implementation of this service interface. This code has a problem: the whole `registerMailbox` method is synchronized, including

[3]Though perhaps they should be!

Listing 6.10 Mailbox Registration Service Interface

```
1 package org.osgi.book.concurrency;

3 import org.osgi.book.reader.api.Mailbox;

5 public interface MailboxRegistrationService {
6    void registerMailbox(String name, Mailbox mailbox);
7 }
```

the calls to OSGi to register and unregister the services. That means we will enter the OSGi API with a lock held and therefore any callbacks — such as the `addingService` method of a tracker — will also be called with that lock held. If any of those callbacks then try to acquire another lock, we may end up in deadlock.

Listing 6.11 Holding a lock while calling OSGi APIs

```
1 public class BadLockingMailboxRegistrationService implements
2       MailboxRegistrationService {

4    private final Map<String, ServiceRegistration> map
5       = new HashMap<String, ServiceRegistration>();
6    private final BundleContext context;

8    public BadLockingMailboxRegistrationService(BundleContext context) {
9       this.context = context;
10   }

12   // DO NOT DO THIS!
13   public synchronized void registerMailbox(String name,
14                                            Mailbox mailbox) {
15      ServiceRegistration priorReg = map.get(name);
16      if(priorReg != null) priorReg.unregister();

18      Properties props = new Properties();
19      props.put(Mailbox.NAME_PROPERTY, name);
20      ServiceRegistration reg = context.registerService(
21                Mailbox.class.getName(), mailbox, props);

23      map.put(name, reg);
24   }
25 }
```

The Wikipedia definition of *Deadlock*[**?**] refers to a charmingly illogical extract from an Act of the Kansas State Legislature:

> "When two trains approach each other at a crossing, both shall come to a full stop and neither shall start up again until the other has gone."[**?**]

This is a classic deadlock: neither train can make any progress because it is waiting for the other train to act first.

Another familiar example of deadlock is the so-called "dining philosophers"

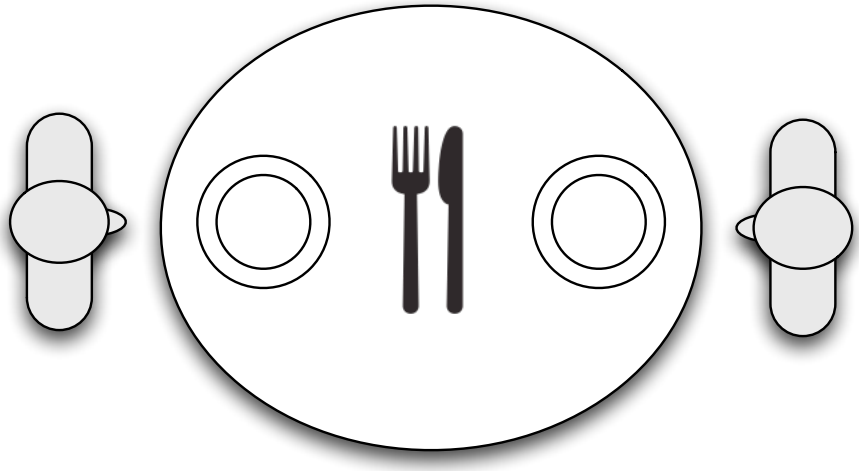


Figure 6.2.: The Dining Philosophers Problem, Simplified

problem, first described by Edsger Dijkstra in [**?**]. To reduce this problem to its bare minimum, imagine two philosophers at a dinner table waiting to eat, but there is only one knife and one fork (Figure 6.2). A philosopher needs both utensils in order to eat, but one immediately picks up the knife the other immediately picks up the fork. They both then wait for the other utensil to become available. It should be clear that this strategy will eventually result in both philosophers starving to death.

To translate this unhappy situation to OSGi programming, imagine that a thread has taken a lock on an object F. Then it tries to call our `registerMailbox`, which locks object K — but it must wait, perhaps because another thread is already executing `registerMailbox`. One of the callbacks resulting from the service registration then attempts to lock F. The result: two starving threads and no work done.

The traditional solution to the dining philosophers problem is simply to always take the knife and fork in the same order. If both philosophers try to get the fork before the knife, then the first one to pick up the fork will eat first, and then the second philosopher will eat when the first is finished. This may not be very fair, but at least nobody starves. Similarly, the safe way to take locks on multiple objects is to always take them in the same order. The ordering can be arbitrary but the important thing is to consistently apply whatever ordering scheme is chosen (the philosophers also get to eat if they both attempt to take the knife first — it is only when they choose a different first utensil that they starve).

Unfortunately when calling an OSGi API method that involves callbacks into

other bundles, we simply cannot enforce any such ordering, because those bundles cannot know about our ordering scheme. The only alternative is to restructure our code to avoid holding any locks when calling the OSGi APIs. Note that avoiding locks does not just mean avoiding the `synchronized` keyword. We could achieve the level of isolation we desire in the `registerMailbox` method by using a binary semaphore or an instance of the `Lock` classes. However when used in such a way, these constructs have exactly the same semantics as a `synchronized` method or block, and can result in the same types of deadlock.

The trick is avoid holding locks during calls to OSGi APIs, but to do this *without* losing the atomicity guarantees that we require. Sometimes this requires some re-ordering of the operations. For example, Listing 6.12 shows a version of the service that uses a lock only to manipulate the `map` field[4]. The result of the operation passes out of the locked region and tells us whether there was a prior registration of the service which needs to be unregistered. Note that the `put` method of `Map` returns the previous mapping for the key if one existed.

Due to the reordering there is a slight change in the externally observable behaviour of this method. The previous version (if it worked!) would result in a short gap between the old service being removed and the new service being created, during which there would be no service available. The new version reverses the steps: the new service is created very slightly *before* the old service is removed, so there will briefly be two services rather than none. It turns out this is not such a bad thing: as well as making it possible to reduce the size of our locked regions, consumers of the service also benefit from having a replacement service immediately available when the first service goes away.

6.5. GUI Development

Another area where multi-threading causes pain is in programming graphical user interfaces (GUIs). Almost all GUI libraries — including the most popular Java ones, Swing and SWT — insist that all calls to those libraries must be made from a single thread, the "event dispatch thread" or EDT. But as we have seen, when our callbacks or service methods are executed, we have no idea whether we are in the EDT or some other, arbitrary thread. Therefore we have to use utilities supplied by the GUI library to pass blocks of code that will be executed in the EDT when it gets around to it. In Swing, we need to pass a `Runnable` instance to the `SwingUtilities.invokeLater` method, but for efficiency, we should first check whether we're already in the EDT by calling `EventQueue.isDispatchThread`

[4] In this example we use a plain `HashMap` and wrap the calls to it in a `synchronized` block in order to be explicit about the locking. We could have used a `ConcurrentHashMap` which performs fine-grained internal locking with no need for a `synchronized` block.

Listing 6.12 Avoiding holding a lock while calling OSGi APIs

```
1  package org.osgi.book.concurrency;
2
3  import java.util.HashMap;
4  import java.util.Map;
5  import java.util.Properties;
6
7  import org.osgi.book.reader.api.Mailbox;
8  import org.osgi.framework.BundleContext;
9  import org.osgi.framework.ServiceRegistration;
10
11 public class GoodLockingMailboxRegistrationService implements
12       MailboxRegistrationService {
13
14    private final Map<String, ServiceRegistration> map
15       = new HashMap<String, ServiceRegistration>();
16
17    private final BundleContext context;
18
19    public GoodLockingMailboxRegistrationService(BundleContext context) {
20       this.context = context;
21    }
22
23    public void registerMailbox(String name, Mailbox mailbox) {
24       Properties props = new Properties();
25       props.put(Mailbox.NAME_PROPERTY, name);
26       ServiceRegistration reg = context.registerService(
27                Mailbox.class.getName(), mailbox, props);
28
29       ServiceRegistration priorReg;
30       synchronized (map) {
31          priorReg = map.put(name, reg);
32       }
33
34       if (priorReg != null) {
35          priorReg.unregister();
36       }
37    }
38 }
```

The example in Listing 6.13 uses this technique to update the text of a Swing label object to indicate the current number of mailbox services.

Unfortunately, programming with Swing (or any GUI library) in a multi-threaded environment can quickly become cumbersome due to the need to create anonymous inner classes implementing the `Runnable` interface. We cannot refactor the complexity out into library methods because the Java language does not support closures or "blocks". This is an area where we can benefit from using a more powerful programming language, such as Scala or JRuby. For example, assuming we have defined a Scala library function for performing actions in the EDT as shown in Listing 6.14, we can then write a same tracker class in Scala using the code in Listing 6.15.

6.6. Using Executors

Perversely, despite the cumbersome code, working with GUIs can *simplify* some aspects of concurrent programming. Since we always have to transfer work to be run on a single thread, we can take advantage of that thread to perform mutations to state without needing to take any locks at all — so long as the variables we mutate are *only* touched within the GUI thread.

We can exploit this pattern even when not writing GUIs. In Section 6.4 we saw the need to avoid holding locks when making calls to certain OSGi API methods, and in the example code we had to slightly reorder the operations to allow us to safely perform those OSGi calls outside of a lock. This led to a small change in the behaviour of the method — brief periods of time in which two Mailbox services are registered.

Suppose we are under some constraint which means we simply cannot have two Mailbox services registered at the same time. This means we *must* first unregister the existing service before registering the new one. Now we have a problem: without wrapping the two operations in a synchronized block, we cannot make them atomic, so a race condition could occur in which two threads both simultaneously unregister the existing service and then both create and register new services... the very situation we were trying to avoid!

There is a solution which satisfies both the requirement not to lock and the requirement to update atomically: simply perform all of the updates from a single thread. We could do this by handing updates to a thread that we create specifically for the purpose of updating our Mailbox service. In Java 5 this is easily done with an `Executor` as shown in Listing 6.16.

Of course, this solution also produces a subtle change to the behaviour of the service. Now the update to the service doesn't happen synchronously during the call to `registerMailbox`, but asynchronously shortly afterwards.

Listing 6.13 Updating a Swing Control in Response to a Service Event

```
1  package org.osgi.book.reader.tutorial;
2
3  import java.awt.EventQueue;
4
5  import javax.swing.JLabel;
6  import javax.swing.SwingUtilities;
7
8  import org.osgi.book.reader.api.Mailbox;
9  import org.osgi.framework.BundleContext;
10 import org.osgi.framework.ServiceReference;
11 import org.osgi.util.tracker.ServiceTracker;
12
13 public class JLabelMailboxCountTracker extends ServiceTracker {
14
15    private final JLabel label;
16    private int count = 0;
17
18    public JLabelMailboxCountTracker(JLabel label,
19          BundleContext context) {
20       super(context, Mailbox.class.getName(), null);
21       this.label = label;
22    }
23
24    @Override
25    public Object addingService(ServiceReference reference) {
26       int displayCount;
27       synchronized (this) {
28          count++;
29          displayCount = count;
30       }
31       updateDisplay(displayCount);
32       return null;
33    }
34
35    @Override
36    public void removedService(ServiceReference reference,
37          Object service) {
38       int displayCount;
39       synchronized (this) {
40          count--;
41          displayCount = count;
42       }
43       updateDisplay(displayCount);
44    }
45
46    private void updateDisplay(final int displayCount) {
47       Runnable action = new Runnable() {
48          public void run() {
49             label.setText("There are " + displayCount + " mailboxes");
50          }
51       };
52       if(EventQueue.isDispatchThread()) {
53          action.run();
54       } else {
55          SwingUtilities.invokeLater(action);
56       }
57    }
58 }
```

Listing 6.14 A Scala Utility to Execute a Closure in the Event Thread

```
def inEDT(action: => Unit) =
  if (EventQueue.isDispatchThread())
    action
  else
    SwingUtilities.invokeLater(new Runnable() {def run = action})
```

Listing 6.15 Updating a Swing Control — Scala Version

```
class JLabelMailboxCountTracker(ctx: BundleContext, label: JLabel)
    extends ServiceTracker(ctx, classOf[Mailbox].getName, null) {

  private var count = 0: int

  override
  def addingService(ref: ServiceReference): AnyRef = {
    var displayCount = 0
    synchronized { count += 1; displayCount = count }

    inEDT(label.setText("foo"))
    null
  }

  override
  def removedService(ref: ServiceReference, service: AnyRef) = {
    var displayCount = 0
    synchronized { count -= 1; displayCount = count }

    inEDT(label.setText("foo"))
  }
}
```

Listing 6.16 Single-threaded execution

```
1  package org.osgi.book.concurrency;

3  import java.util.HashMap;
4  import java.util.Map;
5  import java.util.Properties;
6  import java.util.concurrent.ExecutorService;
7  import java.util.concurrent.Executors;

9  import org.osgi.book.reader.api.Mailbox;
10 import org.osgi.framework.BundleContext;
11 import org.osgi.framework.ServiceRegistration;

13 public class SingleThreadedMailboxRegistrationService implements
14       MailboxRegistrationService {

16    private final Map<String, ServiceRegistration> map
17       = new HashMap<String, ServiceRegistration>();
18    private final BundleContext context;

20    private final ExecutorService executor =
21          Executors.newSingleThreadExecutor();

23    public SingleThreadedMailboxRegistrationService(
24          BundleContext context) {
25       this.context = context;
26    }

28    public void registerMailbox(final String name,
29                                final Mailbox mailbox) {
30       Runnable task = new Runnable() {
31          public void run() {
32             ServiceRegistration priorReg = map.get(name);
33             priorReg.unregister();

35             Properties props = new Properties();
36             props.put(Mailbox.NAME_PROPERTY, name);
37             ServiceRegistration reg = context.registerService(
38                   Mailbox.class.getName(), mailbox, props);
39             map.put(name, reg);
40          }
41       };
42       executor.execute(task);
43    }

45    public void cleanup() {
46       executor.shutdown();
47    }
48 }
```

This is essentially a trade-off that we cannot escape — the requirement to avoid holding a lock forces us to either reorder operations (as in the previous solution) or execute asynchronously. In most cases the reordering solution is preferable.

There is another problem with the code in Listing 6.16: it always creates its own thread irrespective of any application-wide management policies with respect to thread creation. Threads are somewhat expensive in Java, and if too many services create their own single-purpose threads then we will have a problem managing our application. Note also that we need to remember to cleanup this service to ensure the thread is shutdown when no longer needed.

In this case we can employ a useful form of executor called the `SerialExecutor`, the code for which appears in Listing 6.17. This executor ensures that at most one task is running at any time, but it achieves this without creating a thread or locking anything during the execution of a task. The trick is an internal *work queue*. If a call to `execute` arrives while the executor is idle, then the thread making that call becomes the "master" and it immediately executes the task... note that the task is executed *synchronously* in the calling thread. However if a call to `execute` arrives while a master is already running then we simply add the task to the end of the work queue and immediately return to the caller. The task will be executed by the master thread, which ensures that the work queue is emptied before returning to its caller. As soon as the master thread returns from `execute`, the executor is idle and the next thread to call `execute` will become the new master.

Listing 6.18 shows the Mailbox registration service yet again, using `SerialExecutor`. In this version we can get rid of the `cleanup` method since there is no thread to shutdown, but we need to switch the map to a thread-safe version for visibility, since any thread can become the master thread.

The `SerialExecutor` class is useful, and you should consider keeping it handy in your OSGi toolbox, but it is also not a panacea. We must still consider it asynchronous, since non-master threads will return to their callers before the work is completed by the master thread. Also it is completely unsuitable when calls to `execute` are made with too high frequency, since any thread unlucky enough to be nominated the master will be stuck executing the work of many other threads before it is finally able to return to its caller. A possible enhancement would be to allow `SerialExecutor` to spin off a thread if the master thread decided that enough was enough.

Another useful pattern is to have a bundle export one or more executors as services. In doing this we resolve the dilemma of having bundles that wish to spin off threads to perform work asynchronously versus the desire to manage creation of threads at an application level. We can create a single "work manager" bundle that creates thread pools in various configurations — single threads, fixed size pools, resizable pools, etc. — and then any bundle wishing

Listing 6.17 The `SerialExecutor` class

```
1  package org.osgi.book.utils;
2
3  import java.util.ArrayList;
4  import java.util.List;
5  import java.util.concurrent.Executor;
6
7  import org.osgi.service.log.LogService;
8
9  public class SerialExecutor implements Executor {
10    private final List<Runnable> queue = new ArrayList<Runnable>();
11    private final ThreadLocal<Integer> taskCount;
12    private final LogService log;
13
14    public SerialExecutor() {
15       this(null);
16    }
17
18    public SerialExecutor(LogService log) {
19       this.log = log;
20       taskCount = new ThreadLocal<Integer>() {
21          protected Integer initialValue() {
22             return 0;
23          }
24       };
25    }
26
27    public void execute(Runnable command) {
28       Runnable next = null;
29       int worked = 0;
30
31       // If the queue is empty, I am the master and my next task is to
32       // execute my own work. If the queue is non-empty, then I simply
33       // add my task to the queue and return.
34       synchronized (this) {
35          next = queue.isEmpty() ? command : null;
36          queue.add(command);
37       }
38
39       while (next != null) {
40          // Do the work!
41          try {
42             next.run();
43             worked++;
44          } catch (Exception e) {
45             logError("Error processing task", e);
46          }
47
48          // Get more work if it exists on the queue
49          synchronized (this) {
50             queue.remove(0); // Head element is the one just processed
51             next = queue.isEmpty() ? null : queue.get(0);
52          }
53
54          taskCount.set(worked);
55       }
56    }
```

Listing 6.17 (continued)

```
    /**
     * Returns the number of tasks executed by the last call to
     * <code>execute</code> from the calling thread.
     */
    public int getLastTaskCount() {
        return taskCount.get();
    }

    private void logError(String message, Exception e) {
        if (log != null) {
            log.log(LogService.LOG_ERROR, message, e);
        }
    }
}
```

Listing 6.18 The Mailbox registration service using `SerialExecutor`

```
package org.osgi.book.concurrency;

import java.util.HashMap;
import java.util.Map;
import java.util.Properties;
import java.util.concurrent.ConcurrentHashMap;
import java.util.concurrent.Executor;

import org.osgi.book.reader.api.Mailbox;
import org.osgi.book.utils.SerialExecutor;
import org.osgi.framework.BundleContext;
import org.osgi.framework.ServiceRegistration;

public class SerialMailboxRegistrationService implements
        MailboxRegistrationService {

    private final Map<String, ServiceRegistration> map =
            new ConcurrentHashMap<String, ServiceRegistration>();
    private final BundleContext context;

    private final Executor executor = new SerialExecutor();

    public SerialMailboxRegistrationService(BundleContext context) {
        this.context = context;
    }

    public void registerMailbox(final String name,
                final Mailbox mailbox) {
        Runnable task = new Runnable() {
            public void run() {
                ServiceRegistration priorReg = map.get(name);
                if(priorReg != null) priorReg.unregister();

                Properties props = new Properties();
                props.put(Mailbox.NAME_PROPERTY, name);
                ServiceRegistration reg = context.registerService(
                        Mailbox.class.getName(), mailbox, props);

                map.put(name, reg);
            }
        };
        executor.execute(task);
    }
}
```

to execute tasks can submit them via the service interface. Listing 6.19 shows an example of registering such an executor; note that we register a wrapper object around the thread pool rather than registering the thread pool directly, as this prevents clients from casting the service to its `ExecutorService` interface and calling methods such as `shutdown` on it.

Listing 6.19 Registering a Thread Pool as an Executor Service

```
1  package org.osgi.book.concurrency;
2
3  import java.util.Properties;
4  import java.util.concurrent.Executor;
5  import java.util.concurrent.ExecutorService;
6  import java.util.concurrent.Executors;
7
8  import org.osgi.framework.BundleActivator;
9  import org.osgi.framework.BundleContext;
10 import org.osgi.framework.ServiceRegistration;
11
12 public class ExecutorServiceActivator implements BundleActivator {
13
14    private static final int POOL_SIZE = 20;
15
16    private volatile ExecutorService threadPool;
17    private volatile ServiceRegistration svcReg;
18
19    public void start(BundleContext context) {
20       threadPool = Executors.newFixedThreadPool(POOL_SIZE);
21
22       Executor wrapper = new Executor() {
23          public void execute(Runnable command) {
24             threadPool.execute(command);
25          }
26       };
27
28       Properties props = new Properties();
29       props.put("kind", "fixed");
30       props.put("poolSize", POOL_SIZE);
31       svcReg = context.registerService(Executor.class.getName(),
32             wrapper, props);
33    }
34
35    public void stop(BundleContext context) {
36       svcReg.unregister();
37       threadPool.shutdown();
38    }
39 }
```

6.7. Interrupting Threads

As we discussed in Chapter 2, there is no general way to force a thread to stop in Java, except by shutting down the whole Java Virtual Machine. This is by design: if such a mechanism existed then threads could be terminated while mutating data structures, and would be likely to leave them in an inconsistent state. Instead, Java offers a co-operative mechanism whereby we *ask* a thread

to stop. The implementation code for a thread must be able to respond to such a request and perform the appropriate clean-up.

Unfortunately, in traditional (non-OSGi) Java development, these issues are rarely considered. Often threads are simply allowed to run to completion, or in some cases a thread is created during start-up of the application and assumed to run for as long as the Java process itself is running. For example a web server application would create a thread for accepting socket connections, and that thread would run until the server is shutdown. In these scenarios, there is no need to handle termination requests: the thread will be automatically stopped when Java stops.

In OSGi we don't have this luxury. Our bundle can be shutdown by a call to the `stop` method of its activator, and when that happens we *must* cleanup all artefacts that have been created on behalf of our bundle, including threads, sockets and so on. We cannot rely on our threads being forcibly ended by the termination of the JVM.

So how do we ask a thread to stop? Sadly Java doesn't even have a single consistent way to do this. There are a number of techniques, but we must choose the correct one based on what the thread is doing when we wish it to stop. Therefore in most cases we must tailor the termination code to the thread implementation code.

The code we saw in Chapter 2 (Listing 2.7) used the *interruption* mechanism, which is a simple booolean status that can be set on a thread by calling the `interrupt` method. It is possible for a thread to explicitly check its interruption status by calling `Thread.interrupted`, and also certain library methods (like `Thread.sleep` and `Object.wait`) are aware of their calling thread's interruption status and will exit immediately with an `InterruptedException` if the thread is interrupted while they are running.

Unfortunately, not all blocking library methods respond to interruption. For example most blocking I/O methods in `java.io` package simply ignore the interruption status; these methods continue to block, sometimes indefinitely, even after the thread has been interrupted. To wake them up we need some knowledge of what is blocking and why.

Listing 6.20 shows a server thread that accepts client connections. When a client connects, the server immediately sends the message "Hello, World!" to the client and then closes the connection to that client.

The activator here looks identical to `HeartbeatActivator` from Chapter 2: to stop the thread, we call its `interrupt` method. However in this case we have overridden the normal `interrupt` method with our own implementation that closes the server socket in addition to setting the interruption status. If the thread is currently blocking in the `ServerSocket.accept` method, closing the socket will cause it to exit immediately with an `IOException`. This pattern

Listing 6.20 Server Activator

```
1  package org.osgi.book.concurrency;

3  import java.io.IOException;
4  import java.io.PrintStream;
5  import java.net.ServerSocket;
6  import java.net.Socket;

8  import org.osgi.framework.BundleActivator;
9  import org.osgi.framework.BundleContext;

11 public class ServerActivator implements BundleActivator {

13    private HelloServer serverThread = new HelloServer();

15    public void start(BundleContext context) throws Exception {
16       serverThread.start();
17    }

19    public void stop(BundleContext context) throws Exception {
20       serverThread.interrupt();
21    }

23 }

25 class HelloServer extends Thread {

27    private static final int PORT = 9876;

29    private volatile ServerSocket socket = null;

31    public void interrupt() {
32       super.interrupt();
33       try {
34          if (socket != null) {
35             socket.close();
36          }
37       } catch (IOException e) {
38          // Ignore
39       }
40    }

42    public void run() {
43       try {
44          socket = new ServerSocket(PORT);

46          while (!Thread.currentThread().isInterrupted()) {
47             System.out.println("Accepting connections...");
48             Socket clientSock = socket.accept();
49             System.out.println("Client connected.");
50             PrintStream out = new PrintStream(clientSock
51                   .getOutputStream());
52             out.println("Hello, World!");
53             out.flush();
54             out.close();
55          }
56       } catch (IOException e) {
57          System.out.println("Server thread terminated.");
58       }
59    }

61 }
```

works for most of the `java.io` library: to interrupt a blocked I/O method, we can close the underlying socket or stream that it is blocking on.

As a rule of thumb, when you see a blocking library method that does *not* declare `InterruptedException` in its `throws` clause, it generally means it does not respond to interruption, and you must find another way to force it awake.

Much more discussion of these issues can be read in [**?**].

6.8. Exercises

1. Write a `ServiceTracker` that tracks a `Mailbox` services and stores them in a `Map<String, List<Mailbox>>`. The map should be keyed on the location of the publishing bundle (available from `ServiceReference.getBundle().getLocation()`) and the values should be the list of all `Mailbox` services published by that bundle. Also provide a `getMailboxes(String)` method that returns a snapshot of the services for a given bundle location. Ensure that any `synchronized` blocks used are as short as possible.
2. See the service interface in Listing 6.21. Write an implementation of this service which creates a thread pool of the size specified in the `size` field and registers it as an `Executor` service. The first time this method is called it should simply create and register the thread pool. On subsequent calls it should create and register a new thread pool, and also unregister and shutdown the old thread pool. Assume that your service class takes a `BundleContext` in its constructor.

.

Listing 6.21 Exercise 2: Thread Pool Manager Interface

```
1 package org.osgi.book.concurrency;

3 public interface ThreadPoolManager {
4    void updateThreadPool(int size);
5 }
```

7. The Whiteboard Pattern and Event Admin

So far our Mailbox Reader application is extremely simple; we can only display the headers of messages that were available in a mailbox at the time that the table was displayed. To be even remotely useful, the reader would also need to notify us when new messages arrive... otherwise we would have to restart the application to check for new messages!

The mechanism for new message notification variable across different types of mailboxes. With some communication protocols — such as POP3 email or RSS over HTTP — we must poll the server occasionally to ask if there are any new messages. Under other protocols such as SMS and IMAP, the server is able to "push" messages to our client as soon as they are available. Naturally the details of the notification mechanism need to be hidden inside the mailbox service implementation rather than exposed to clients of the service, such as our Reader GUI.

7.1. The Classic Observer Pattern

Most Java programmers will recognise the above as a problem that can be solved with the Observer pattern, also sometimes called the Listener pattern. And they would be correct, however OSGi adds a twist to the standard Java approach.

The classic pattern as defined by the "Gang of Four" [10] involves two entities: an Observable, which is the source of events; and a Listener, which consumes those events. The listener must be registered with an observable in order to receive events from it. Then the observable would notify all registered listeners of each event as it occurred. This is shown in Figure 7.1.

To use this pattern in our Mailbox Reader application, we could define a listener interface named `MailboxListener`. We would also need to extend the `Mailbox` interface to make it act as an observable, by adding methods to register and unregister listeners. These two interfaces are shown together in Listing 7.1.

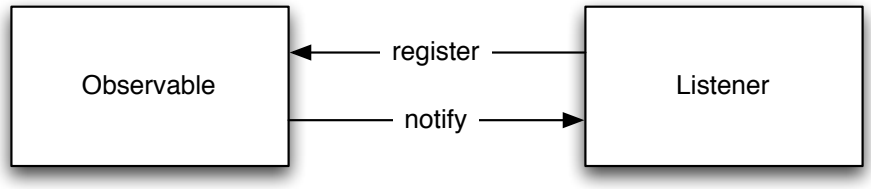


Figure 7.1.: The Classic Observer Pattern

Listing 7.1 Mailbox Listener and Observable Mailbox Interfaces

```
1 package org.osgi.book.reader.api;

3 public interface MailboxListener {
4    void messagesArrived(String mailboxName, Mailbox mailbox, long[] ids);
5 }
```

```
1 package org.osgi.book.reader.api;

3 public interface ObservableMailbox extends Mailbox {
4    void addMailboxListener(MailboxListener listener);
5    void removeMailboxListener(MailboxListener listener);
6 }
```

7.2. Problems with the Observer Pattern

The observer pattern is very familiar, especially in GUI programming, but it has a number of problems. Most of these problems are general but they are particularly bad when we translate the pattern into an OSGi context.

The first problem is keeping track of dynamic observable objects. In our example, the source of events is a service that — like all services — has a life-cycle and can come and go many times during the life of the application. But when a new mailbox service appears, no listeners will be registered with it. Therefore any listener needs to hold open a `ServiceTracker` so it can be notified when new mailbox services appear and register itself, and likewise unregister itself when the service is going away. We must also find a way to deal with "lost updates", i.e. events that occur between the mailbox appearing and the listener(s) registering themselves with it. If there are many listeners then they will not all register at the same time, so some listeners may see events that others do not see.

A second problem is that, just as the mailboxes can come and go, so can the listeners. Each observable therefore needs to manage a changing set of listeners, and make sure it does not attempt to deliver events to listeners that have disappeared. Since the registering and unregistering of listeners can occur in a different thread from the firing of an event, proper synchronization must

be used to avoid concurrency bugs.

There is a memory management problem here also: the observer pattern is one of the principal causes of memory leaks in Java applications. When a listener is registered with an observable, the internal implementation of the observable will typically add that listener to collection field. But now there is a strong reference to the listener object merely because it exists inside the collection, preventing it from being cleaned up by the garbage collector. Even if the listener is not useful any more in its original context, it will live on in memory until the observer dies.

Therefore it is very important to clean up listeners when they are no longer required. But this is very difficult to verify, and is often not done correctly. The problem is that it is very easy to detect a problem with the set-up phase — if done incorrectly, the application simply does not work — but problems with cleaning up do not directly break the application, instead causing "out of memory" conditions much later on.

7.3. Fixing the Observer Pattern

None of the problems above are insurmountable, of course. By writing both our listeners and observers carefully and correctly we can avoid all concurrency bugs and memory leaks. The remaining problem, however, is that there is quite a lot of complex code to write each time we want to use the observer pattern, and it is not good practice to repeat such code many times. Therefore we need to look for a way to centralise that code in a single reusable component or library, so that we can go back to writing "dumb" observers and listeners.

One possible solution is to create some kind of event broker that sits between the event sources and the event listeners. The listeners could register themselves against the broker rather than directly with each observable, and the event sources would simply publish events to the broker and allow it to deliver those events to all registered listeners. Figure 7.2 shows the general idea.

How does this simplify matters for the listeners? Instead of having to track each observer and register and unregister themselves each time an observer appears and disappears, the listeners would simply need to register themselves *once* with the broker and unregister once when no longer needed. We can assume the broker will have a much longer life than the individual observers or listeners, so we will not have to worry about the broker itself going away.

And for the event sources? They now have a much easier job. Rather than managing a variable size collection of listeners, they can simply publish their events to a single place. In fact we no longer need special observable interfaces such as `ObservableMailbox` from Listing 7.1. Now, *any* object can publish

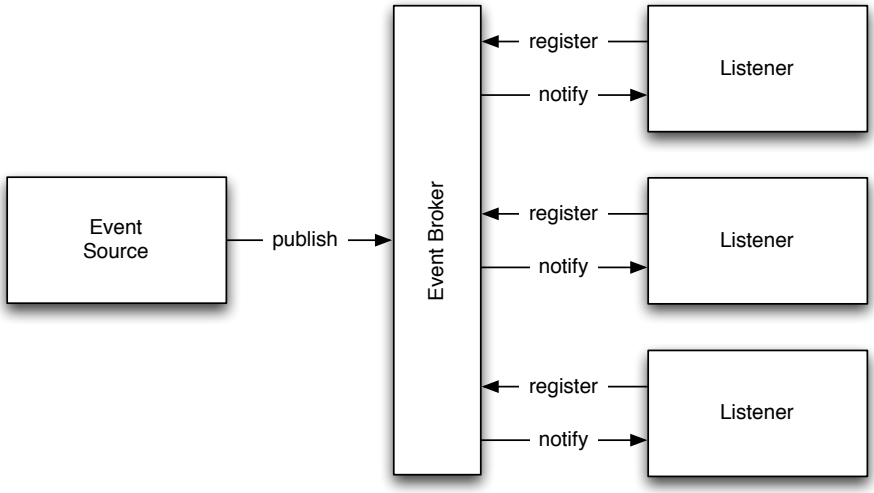


Figure 7.2.: An Event Broker

events to all of the current listeners for that event type simply by calling the event broker. So, both event sources and listeners can now be written with dumb code.

An alternative to the event broker is a "listener directory", as in Figure 7.3. This would take responsibility for managing the listeners for a particular kind of event, but it would not actually deliver the events itself. Instead it would provide a snapshot of the current set of listeners, enabling the event source to deliver the events. This is slightly more flexible, since the event source could choose to follow a custom delivery strategy. For example it may wish to deliver events to multiple listeners in parallel using a thread pool.

Of course, either the event broker or the listener directory implementations will be quite complex and must be written with great care, although fortunately they need only be written once. However, in OSGi we are lucky because an implementation of both these patterns already exists! In fact the listener directory is simply the Service Registry that we are already familiar with.

7.4. Using the Whiteboard Pattern

The listener directory pattern we have just described is more poetically known as the Whiteboard Pattern. We will see why this name fits in a moment, but let's look at how we can use the Service Registry as a directory of listeners.

To register a listener, we simply take that listener object and publish it to the

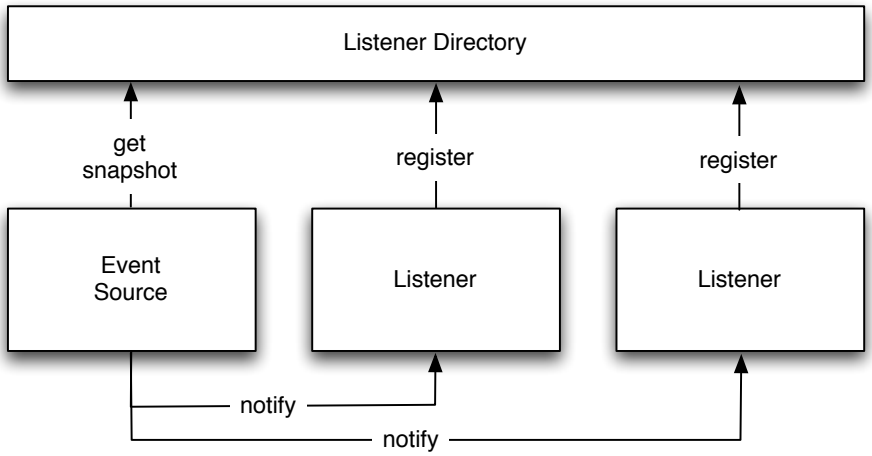


Figure 7.3.: A Listener Directory

service registry under its specific `Listener` interface. For example, Listing 7.2 shows a bundle activator that registers an implementation of the `MailboxListener` interface. This is literally *all we have to do* on the listener side.

Listing 7.2 Registering a Mailbox Listener

```
1  package org.osgi.book.reader.gui;

3  import org.osgi.book.reader.api.MailboxListener;
4  import org.osgi.framework.BundleActivator;
5  import org.osgi.framework.BundleContext;

7  public class MailboxListenerActivator implements BundleActivator {

9      public void start(BundleContext context) throws Exception {
10         MailboxListener listener = new MyMailboxListener();
11         context.registerService(MailboxListener.class.getName(),
12                                 listener, null);
13     }

15     public void stop(BundleContext context) throws Exception {
16         // No need to explicitly unregister the service
17     }
18 }
```

This simplicity explains the name “Whiteboard Pattern”. We do not need to actively seek out event sources and explicitly register against them, instead we merely declare the existence of the listener and wait for event sources to find it. This is similar to writing your name on a whiteboard in order to declare your interest in joining an activity, rather than finding the organiser of that activity and registering your interest directly.

Then how does an event source interact with the whiteboard pattern? Well, it must do more than just registering a service, but not a lot more. Typically it would use a `ServiceTracker` and call the `getServices` method to get a snapshot of all the currently registered listeners, which it can then iterate over. An example based on the `MailboxListener` interface is shown in Listing 7.3. Here we subclass `ServiceTracker` and offer a `fireMessagesArrived` method that performs the work.

Listing 7.3 Mailbox Listener Tracker

```
1  package org.osgi.book.reader.asyncmailbox;
2
3  import org.osgi.book.reader.api.Mailbox;
4  import org.osgi.book.reader.api.MailboxListener;
5  import org.osgi.framework.BundleContext;
6  import org.osgi.util.tracker.ServiceTracker;
7
8  public class MailboxListenerTracker extends ServiceTracker {
9
10    public MailboxListenerTracker(BundleContext context) {
11       super(context, MailboxListener.class.getName(), null);
12    }
13
14    public void fireMessagesArrived(String mailboxName,
15          Mailbox mailbox, long[] ids) {
16       Object[] services = getServices();
17       for (int i = 0; services != null && i < services.length; i++) {
18          ((MailboxListener) services[i]).messagesArrived(mailboxName,
19                mailbox, ids);
20       }
21    }
22 }
```

This `MailboxListenerTracker` class is an obvious candidate for reuse: all mailbox implementations that have the capability to receive messages asynchronously will probably want to use this class or something very similar. So, we could pull it out into a shared library or even supply it as a helper class in the base Mailbox API.

However, we can do better than this. There is an even more general pattern at work in this class, which readers of "Gang of Four" will recognise as the Visitor pattern. Given this observation we can write a general purpose helper class that can be reused every time we implement the whiteboard pattern. This is shown in Listing 7.4, in which we define a `WhiteboardHelper` class that takes instances of a `Visitor` interface. To use the helper in our Mailbox example we can simply pass in an instance of `Visitor<MailboxListener>` that makes a call to the `messagesArrived` method. Java 5 generics are used to make the pattern type-safe but it can also work under Java 1.4 using casts.

Listing 7.4 Visitor Interface and Whiteboard Helper Class

```
1 package org.osgi.book.utils;

3 public interface Visitor<T> {
4    void visit(T object);
5 }
```

```
1 package org.osgi.book.utils;

3 import org.osgi.framework.BundleContext;
4 import org.osgi.util.tracker.ServiceTracker;

6 public class WhiteboardHelper<T> extends ServiceTracker {

8    public WhiteboardHelper(BundleContext context, Class<T> svcClass) {
9       super(context, svcClass.getName(), null);
10   }

12   public void accept(Visitor<? super T> visitor) {
13      Object[] services = getServices();
14      if (services != null) {
15         for (Object serviceObj : services) {
16            @SuppressWarnings("unchecked")
17            T service = (T) serviceObj;

19            visitor.visit(service);
20         }
21      }
22   }
23 }
```

7.4.1. Registering the Listener

Now let's have a look at how to introduce the whiteboard pattern into our example Mailbox Reader application, starting with the listener side. In the Mailbox Reader application, each table displaying the contents of a mailbox is likely to be interested in receiving notifications of new messages. In a real-world application, other parts of the GUI are also likely to be interested in these notifications — for example we might want to show an indicator icon in the "System Tray" that pops up messages when a new message arrives. The advantage of the whiteboard approach is that components of the GUI can independently register `MailboxListener` services and receive notifications while remaining oblivious to the existence of other listeners. This enhances the modularity of the application.

In this example we will implement the functionality to update the mailbox table display when new messages arrive. We first need to decide which object will implement the `MailboxListener` interface: it should be one which is close to the model and is able to respond to updates to that model and notify the Swing GUI framework. The obvious choice here is the `MailboxTableModel` class, which inherits several such notification methods from its superclass `AbstractTableModel`. The one we will want to call when new messages arrive

is `fireTableRowsInserted`. Listing 7.5 shows only the new code we need to add to the class.

Listing 7.5 Adding the `MailboxListener` Interface to `MailboxTableModel`

```
1  package org.osgi.book.reader.gui;

3  // Imports omitted ...

5  public class MailboxTableModel extends AbstractTableModel implements
6        MailboxListener {

8     // Previously defined methods omitted ...


2     public void messagesArrived(String mailboxNam, Mailbox mailbox,
3           long[] ids) {
4        if (mailbox != this.mailbox) {
5           // Ignore events for other mailboxes
6           return;
7        }

9        List<Message> newMessages;
10       try {
11          newMessages = Arrays.asList(mailbox.getMessages(ids));
12       } catch (MailboxException e) {
13          newMessages = Collections.emptyList();
14       }

16       final int firstNew, lastNew;
17       synchronized (this) {
18          firstNew = messages.size(); // Index of the first new row
19          messages.addAll(newMessages);
20          lastNew = messages.size() - 1; // Index of the last new row
21       }

23       SwingUtilities.invokeLater(new Runnable() {
24          public void run() {
25             fireTableRowsInserted(firstNew, lastNew);
26          }
27       });
28    }
29 }
```

The next problem is how we register the table model as a service under the `MailboxListener` interface. It makes sense to do this inside `MailboxPanel` because that is the class in which the table model is created and held. We simply add `registerListener` and `unregisterListener` methods to the class. The former needs to take the `BundleContext` as a paramater, but the latter takes no parameter; it simply calls `unregister` on the `ServiceRegistration` object that was created in the `registerListener` method.

The final modification we must make is to include calls to `registerListener` and `unregisterListener` from the `addingService` and `removedService` methods of `ScannerMailboxTracker`. We will not repeat the entire code of the tracker here, since we must only insert two lines:

- In `addingService`, after construction of the new `MailboxPanel` object

Listing 7.6 Mailbox Panel, with `MailboxListener` Registration

```
1  package org.osgi.book.reader.gui;

3  // Imports omitted...

5  public class MailboxPanel extends JPanel {

7      private final MailboxTableModel tableModel;
8      private volatile ServiceRegistration svcReg;

10     // Existing constructor omitted...

1      public void registerListener(BundleContext context) {
2          svcReg = context.registerService(
3                  MailboxListener.class.getName(), tableModel, null);
4      }

6      public void unregisterListener() {
7          if(svcReg == null) throw new IllegalStateException();
8          svcReg.unregister();
9      }
10 }
```

during the `Callable.call` method, we insert:

```
panel.registerListener(context);
```

- In `removedService`, before removing the `MailboxPanel` from the tabbed pane during the `Runnable.run` method, we insert:

```
panel.unregisterListener();
```

7.4.2. Sending Events

Now we look at implementing the event source side of the whiteboard pattern: a mailbox that sends events when new messages arrive.

However, first we need to implement a mailbox which actually does receive new messages! Our previous sample mailbox was fixed in size, with a hard-coded list of initial messages, so now we need a mailbox that can grow. A realistic implementation would involve network sockets and polling and other complexity, so again we will keep things simple and implement a mailbox that grows by one message every five seconds. To achieve this we drive a timer thread from our bundle activator, in a very similar way to the `HeartbeatActivator` example from Chapter 2.

Listing 7.7 shows the activator with the timer thread. We assume that we have a mailbox implementation class `GrowableMailbox` that has an `addMessage` method, allowing messages to be added programmatically. `GrowableMailbox` also takes a `WhiteboardHelper` in its constructor, so we create this in the activator's `start` method before constructing the mailbox.

Listing 7.7 Growable Mailbox Activator and Timer Thread

```
1  package org.osgi.book.reader.asyncmailbox;
2
3  import java.util.Date;
4  import java.util.Properties;
5
6  import org.osgi.book.reader.api.Mailbox;
7  import org.osgi.book.reader.api.MailboxListener;
8  import org.osgi.book.utils.WhiteboardHelper;
9  import org.osgi.framework.BundleActivator;
10 import org.osgi.framework.BundleContext;
11 import org.osgi.framework.ServiceRegistration;
12
13 public class GrowableMailboxActivator implements BundleActivator {
14
15    private static final String MAILBOX_NAME = "growing";
16    private WhiteboardHelper<MailboxListener> whiteboard;
17    private GrowableMailbox mailbox;
18    private Thread messageAdderThread;
19    private ServiceRegistration svcReg;
20
21    public void start(BundleContext context) throws Exception {
22       whiteboard = new WhiteboardHelper<MailboxListener>(context,
23                                          MailboxListener.class);
24       whiteboard.open(true);
25
26       mailbox = new GrowableMailbox(whiteboard, MAILBOX_NAME);
27       messageAdderThread = new Thread(new MessageAdder());
28       messageAdderThread.start();
29
30       Properties props = new Properties();
31       props.put(Mailbox.NAME_PROPERTY, MAILBOX_NAME);
32       svcReg = context.registerService(Mailbox.class.getName(),
33                                        mailbox, props);
34    }
35
36    public void stop(BundleContext context) throws Exception {
37       svcReg.unregister();
38       messageAdderThread.interrupt();
39       whiteboard.close();
40    }
41
42    private class MessageAdder implements Runnable {
43       public void run() {
44          try {
45             while (!Thread.currentThread().isInterrupted()) {
46                Thread.sleep(5000);
47                mailbox.addMessage("Message added at " + new Date(),
48                      "Hello again");
49             }
50          } catch (InterruptedException e) {
51             // Exit quietly
52          }
53       }
54    }
55 }
```

To implement the "growable" mailbox itself, we subclass it from `FixedMailbox` to inherit the internal storage of messages. We just need to add a protected `addMessage` method to be called from the timer thread. This is shown in Listing 7.8. Note the `synchronized` block surrounding access to the internal `messages` field. The superclass must also synchronise accesses to that field, which is the reason why all the methods of `FixedMailbox` were declared `synchronized` (this was left unexplained when that class was first introduced in Chapter 3).

Here is where we see the power of the whiteboard pattern at work. Having added a new message to its internal data structure, `GrowableMailbox` creates a visitor object that calls `messagesArrived` against each listener that it visits.

Listing 7.8 Growable Mailbox

```
1  package org.osgi.book.reader.asyncmailbox;

3  import org.osgi.book.reader.api.Mailbox;
4  import org.osgi.book.reader.api.MailboxListener;
5  import org.osgi.book.reader.api.Message;
6  import org.osgi.book.reader.fixedmailbox.FixedMailbox;
7  import org.osgi.book.reader.fixedmailbox.StringMessage;
8  import org.osgi.book.utils.Visitor;
9  import org.osgi.book.utils.WhiteboardHelper;

11 public class GrowableMailbox extends FixedMailbox {

13    private final WhiteboardHelper<MailboxListener> whiteboard;
14    private final String mailboxName;

16    public GrowableMailbox(WhiteboardHelper<MailboxListener> wb,
17          String mailboxName) {
18       this.whiteboard = wb;
19       this.mailboxName = mailboxName;
20    }

22    protected void addMessage(String subject, String text) {
23       final int newMessageId;

25       synchronized (this) {
26          newMessageId = messages.size();
27          Message newMessage = new StringMessage(newMessageId,
28                subject, text);
29          messages.add(newMessage);
30       }

32       final long[] newMessageIds = new long[] { newMessageId };
33       final Mailbox source = this;
34       Visitor<MailboxListener> v = new Visitor<MailboxListener>() {
35          public void visit(MailboxListener l) {
36             l.messagesArrived(mailboxName, source, newMessageIds);
37          }
38       };
39       whiteboard.accept(v);
40    }
41 }
```

7.5. Event Admin

The whiteboard pattern works well in many situations, such as the one described above, however in some cases it can add overhead that is undesirable.

For example, the event sources must implement all of the code to iterate over the listener services. The `WhiteboardHelper` is rather simplistic: on the first error it stops delivering messages to other listeners that might be registered, and if any listener takes a long time to process an event it will hold up the execution of the event source (i.e., the `addMessage` method of `GrowableMailbox` will not return until all listeners have processed the newly added message). Also, the event sources are still somewhat coupled to the consumers, since they can only deliver events to implementers of a specific interface. Therefore both the event source and the consumer must depend on the same version of that interface.

Sometimes we would like to create an event source that sends many fine-grained events to many different listeners. For example, a financial trading application may need to deliver quotes (e.g., stock quotes, foreign exchange rates, etc) to many consumers of that data. The whiteboard pattern does not scale well in that scenario. Instead we should look at the other pattern described in Section 7.3: the *Event Broker*. OSGi provides us with an implementation of this pattern called the Event Admin Service, which is a service specified in the OSGi Service Compendium. Event Admin implements a form of *publish/subscribe*, which is popular in many message-based systems.

The operation of Event Admin is slightly different from the diagram in Figure 7.2. In that diagram, the listeners registered themselves directly with the event broker, but Event Admin actually uses the whiteboard pattern: listeners register as services under the `EventHandler` interface, and the "broker" tracks them. The diagram for Event Admin is therefore a combination of both Figures 7.2 and 7.3, and is shown in Figure 7.4.

7.5.1. Sending Events

Because it is based on the whiteboard pattern, using Event Admin from the consumer end looks very similar to what we have already seen. So we will first look at the event source end, which does look substantially different. Rather than tracking each listener service and iterating over them each time an event must be sent, when using Event Admin our sources simply need to make a single method call to the Event Admin service.

For our example we will model a financial trading application which continually receives "market data", i.e., prices of various tradable assets. Again, to avoid the incidental complexity of doing this for real we will simulate it by generating

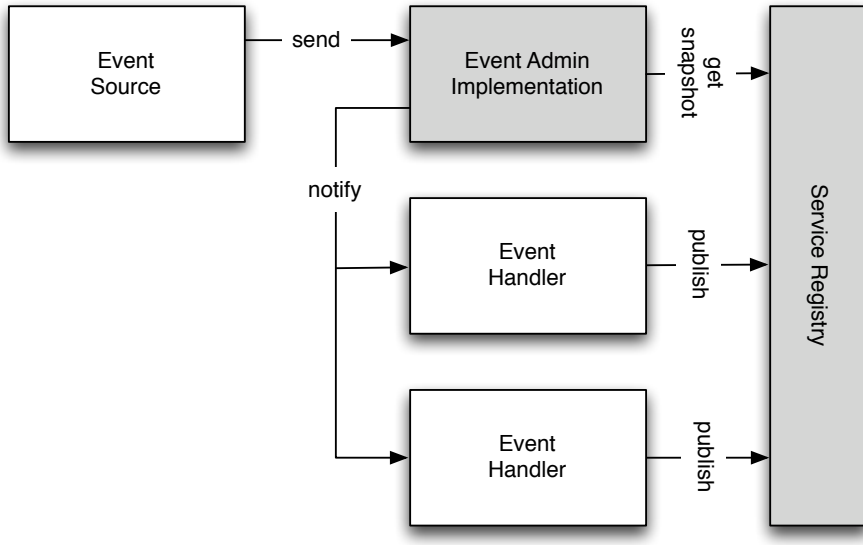


Figure 7.4.: The Event Admin Service

random data on a timer thread. Listing 7.9 shows the implementation of a `Runnable` that does just this: it sends stock quotes for the symbol `MSFT` (Microsoft Corporation) with a starting price of 25 and randomly adjusting upwards or downwards every 100 milliseconds.

This class requires an instance of `EventAdmin` to be passed in its constructor. A good way to provide this is by following the pattern used in the `LogTracker` class from Chapter 4. That is, we subclass `ServiceTracker` and implement the `EventAdmin` interface with delegating calls to the result of calling `getService`. The result is `EventAdminTracker` shown in Listing 7.10

Listing 7.11 shows the activator which manages the `EventAdminTracker` and the timer thread.

7.5.2. The Event Object

The `Event` object what we pass to the Event Admin service is an immutable object that consists of a topic and an arbitrary set of properties.

The topic defines the logical type of the event, and its primary purpose is to act as a filter to determine which handlers should receive which events. It consists of a sequence of tokens separated by slashes, which serve to form a hierarchy. This allows consumers to filter out at any level of the hierarchy, so in the preceding example a consumer could choose to receive: all prices;

Listing 7.9 Random Price Generator

```
1  package org.osgi.book.trading.feeds;
2
3  import java.util.Properties;
4  import java.util.Random;
5
6  import org.osgi.service.event.Event;
7  import org.osgi.service.event.EventAdmin;
8
9  public class RandomPriceFeed implements Runnable {
10
11    private static final String TOPIC = "PRICES/STOCKS/NASDAQ/MSFT";
12
13    private final EventAdmin eventAdmin;
14
15    public RandomPriceFeed(EventAdmin eventAdmin) {
16       this.eventAdmin = eventAdmin;
17    }
18
19    public void run() {
20       double price = 25;
21       Random random = new Random();
22
23       try {
24          while (!Thread.currentThread().isInterrupted()) {
25             // Create  and send the event
26             Properties props = new Properties();
27             props.put("symbol", "MSFT");
28             props.put("time", System.currentTimeMillis());
29             props.put("price", price);
30             eventAdmin.sendEvent(new Event(TOPIC, props));
31
32             // Sleep 100ms
33             Thread.sleep(100);
34
35             // Randomly adjust price by upto 1.0, + or -
36             double nextPrice = random.nextBoolean()
37                                ? price + random.nextDouble()
38                                : price - random.nextDouble();
39             price = Math.max(0, nextPrice);
40          }
41       } catch (InterruptedException e) {
42          // Exit quietly
43       }
44    }
45 }
```

Listing 7.10 Event Admin Tracker

```
1  package org.osgi.book.utils;
2
3  import org.osgi.framework.BundleContext;
4  import org.osgi.service.event.Event;
5  import org.osgi.service.event.EventAdmin;
6  import org.osgi.util.tracker.ServiceTracker;
7
8  public class EventAdminTracker extends ServiceTracker
9                                  implements EventAdmin {
10
11    public EventAdminTracker(BundleContext context) {
12       super(context, EventAdmin.class.getName(), null);
13    }
14
15    public void postEvent(Event event) {
16       EventAdmin evtAdmin = (EventAdmin) getService();
17       if(evtAdmin != null) evtAdmin.postEvent(event);
18    }
19
20    public void sendEvent(Event event) {
21       EventAdmin evtAdmin = (EventAdmin) getService();
22       if(evtAdmin != null) evtAdmin.sendEvent(event);
23    }
24 }
```

Listing 7.11 Bundle Activator for the Random Price Generator

```
1  package org.osgi.book.trading.feeds;
2
3  import org.osgi.book.utils.EventAdminTracker;
4  import org.osgi.framework.BundleActivator;
5  import org.osgi.framework.BundleContext;
6
7  public class RandomPriceFeedActivator implements BundleActivator {
8
9     private EventAdminTracker evtAdmTracker;
10    private Thread thread;
11
12    public void start(BundleContext context) throws Exception {
13       evtAdmTracker = new EventAdminTracker(context);
14       evtAdmTracker.open();
15
16       thread = new Thread(new RandomPriceFeed(evtAdmTracker));
17       thread.start();
18    }
19
20    public void stop(BundleContext context) throws Exception {
21       thread.interrupt();
22       evtAdmTracker.close();
23    }
24 }
```

all *stock* prices; just prices for stocks traded on NASDAQ; or just prices for Microsoft Corporation.

Data describing the actual event should appear in the properties, which is actually an instance of `Dictionary`. So in this example the time and price of the quote are included, as well as the stock symbol. Note that in general, data describing the event should *not* be added to the topic. However a field that is likely to be used as a primary filter (such as the stock symbol in this example) could appear in both the topic and the properties. Only one such field should be used this way, though.

The immutability of the `Event` class is a very important feature. If it were mutable, then any handler could change the content of an event, and therefore other handlers receiving it subsequently would see the altered content instead of the original event created by the sender. To support the immutability of `Event`, the specification for Event Admin states that only instances of String and the eight primitive types (i.e., `int`, `float`, `double` etc). or their Object wrappers (`Integer`, `Float`, `Double`) may be added to the set of properties for an event.

This may seem rather restrictive, and it is tempting to ignore the rule and add more complex objects directly to the event's `Dictionary`. For example we may have a `Quote` class implemented as a standard JavaBean with getters and setters. There is nothing to stop us adding such an object[1] but it is not a good idea to do so. If `Quote` is mutable then the event itself will be effectively mutable. More seriously, we would introduce some coupling between the event sources and consumers: only consumers that import the `Quote` class (and furthermore, the same *version* of it) will be able to conveniently access the contents of the event. Other consumers could use Java reflection to look at the quote object but the code to do this is far more cumbersome and less efficient than simply accessing a `Dictionary`.

Another reason for using only simple values in the event properties is because we can then apply a secondary filter to events based on those values. We will see an example in the next section.

7.5.3. Receiving Events

Since we have already implemented the whiteboard pattern, we already know how to implement an event handler for use with Event Admin: simply register services under the `EventHandler` interface.

However, we must supply at least a topic declaration as a service property along with the registration. This tells the Event Admin broker which topics we are interested in, and we can either specify the topic in full or use a wildcard in

[1] At compile time; some Event Admin implementations *may* enforce the rule at runtime.

order to receive messages on all topics beneath a certain level in the hierarchy. For example specifying `PRICES/STOCKS/*` will give us all stock prices and `PRICES/STOCKS/NYSE/*` will give us just prices published on the New York Stock Exchange. We can also specify `*` alone to receive *all* events on all topics. Note that we cannot place a `*` in the middle of the topic string, i.e., `PRICES/*/NYSE...` is not allowed.

The name for the topic property is "`event.topics`", but from Java code we tend to use a static constant, `EventConstants.EVENT_TOPIC`. This property is mandatory: `EventHandler` services that do not specify a topic will not receive any events.

Another property we can set is "`event.filter`" or `EventConstants.EVENT_FILTER`. This property is optional but it allows us to apply an additional filter on the contents of the event properties, using an LDAP-style query.

Listing 7.12 shows an example of registering an `EventHandler`, which prints stock quotes to the console. It uses a filter to print only prices greater than or equal to 20.

7.5.4. Running the Example

As Event Admin is a compendium service, we need to compile against the compendium API JAR, `osgi.cmpn.jar`. That JAR also needs to be installed in Felix as a bundle to provide the API to our bundles at runtime.

To get the example working we will also need to install an implementation of the Event Admin service. Felix supplies one (as do all the other OSGi frameworks) but it is not included in the default download. However we can install and start it directly from the Felix console as follows (NB: do not include the line break in the middle of the URL):

```
-> install http://www.apache.org/dist/felix/org.apache.felix
    .eventadmin-1.0.0.jar
Bundle ID: 26
-> start 26
->
```

Now we can verify that the Event Admin service is running by typing the `services` commmand. We should see the following somewhere in the output:

```
Apache Felix EventAdmin (26) provides:
--------------------------------------
org.osgi.service.event.EventAdmin
```

To build the sender and receiver bundles, use the two bnd descriptors shown in Listing 7.13.

After building we can install and start the bundles:

Listing 7.12 Stock Ticker Activator

```
1  package org.osgi.tutorial;
2
3  import java.util.Properties;
4
5  import org.osgi.framework.BundleActivator;
6  import org.osgi.framework.BundleContext;
7  import org.osgi.service.event.Event;
8  import org.osgi.service.event.EventConstants;
9  import org.osgi.service.event.EventHandler;
10
11 public class StockTickerActivator implements BundleActivator {
12
13    private static final String STOCKS_TOPIC = "PRICES/STOCKS/*";
14
15    public void start(BundleContext context) throws Exception {
16       EventHandler handler = new EventHandler() {
17          public void handleEvent(Event event) {
18             String symbol = (String) event.getProperty("symbol");
19             Double price = (Double) event.getProperty("price");
20
21             System.out.println("The price of " + symbol
22                   + " is now " + price);
23          }
24       };
25
26       Properties props = new Properties();
27       props.put(EventConstants.EVENT_TOPIC, STOCKS_TOPIC);
28       props.put(EventConstants.EVENT_FILTER, "(price>=20)");
29       context.registerService(EventHandler.class.getName(),
30                               handler, props);
31    }
32
33    public void stop(BundleContext context) throws Exception {
34    }
35
36 }
```

Listing 7.13 Bnd Descriptors for the Random Price Feed and Stock Ticker

```
1 # random_feed.bnd
2 Private-Package: org.osgi.book.trading.feeds,org.osgi.book.utils
3 Bundle-Activator: org.osgi.book.trading.feeds.RandomPriceFeedActivator
```

```
1 # ticker.bnd
2 Private-Package: org.osgi.tutorial
3 Bundle-Activator: org.osgi.tutorial.StockTickerActivator
```

```
-> install file:random_feed.jar
Bundle ID:27
-> install file:ticker.jar
Bundle ID: 28
-> start 27
-> start 28
The price of MSFT is now 25.194525474465635
The price of MSFT is now 24.547478302312108
The price of MSFT is now 25.173540992244572
The price of MSFT is now 25.906669217574922
The price of MSFT is now 25.41915022996729
The price of MSFT is now 26.04457130652444
The price of MSFT is now 26.29259735036186
The price of MSFT is now 25.64594680159028
The price of MSFT is now 26.279434082391102
The price of MSFT is now 27.012694572863026
...
```

7.5.5. Synchronous versus Asynchronous Delivery

Event Admin includes a much more sophisticated event delivery mechanism than the simplistic `WhiteboardHelper` class. One of the most important differences is that Event Admin can optionally deliver events to handlers *asynchronously*.

Most examples of the whiteboard pattern — including our `WhiteboardHelper` class — deliver events synchronously, meaning that each listener is called in turn from the event source thread, and therefore the event source cannot continue any other processing until all of the listeners have finished processing the event, one by one.

Event Admin does support synchronous processing, and that is what we used in the previous example, however it also supports asynchronous processing, which means that the call to the broker will return immediately, allowing the event source to continue with other processing. The events will be delivered to the listeners in one or more threads created by Event Admin for this purpose. To use asynchronous processing we simply call the `postEvent` method of `EventAdmin` rather than `sendEvent`. Nothing else (in our code) needs to change.

Using `postEvent` to request asynchronous delivery makes Event Admin behave much more like a true event broker, albeit an in-JVM one. For the event source it is "fire and forget", allowing us to quickly post an event or message and let Event Admin worry about delivery to the end consumers. Therefore you should in general use `postEvent` as a preference, unless you have a particular need to wait until an event has been delivered to all consumers, in which case you should use `sendEvent`.

Note that using `sendEvent` does *not* guarantee that the `handleEvent` method of all handlers will actually be called in the same thread that the event source used to call `sendEvent`. Event Admin may choose to use several threads to

deliver the event to several handlers concurrently, even when we are using synchronous delivery — the guarantee given by `sendEvent` is merely that it will return to the caller only after the event has been fully delivered to all handlers. As a consequence, we cannot make any assumption about the thread on which handlers will receive events, even if we believe we know which thread is sending them.

Another reason to favour `postEvent` over `sendEvent` is we can induce deadlocks if we hold a lock when calling `sendEvent`, as described at length in Chapter 6. However it is practically impossible to cause deadlock by holding a lock when calling `postEvent`.

7.5.6. Ordered Delivery

Whether we are using synchronous or asynchronous delivery, Event Admin promises to deliver events to each handler in the same order in which they arrived at the broker. That is, if a single thread sends or posts events A, B and C in that order to Event Admin, then each handler will see events A, B and C arriving in that order. However if multiple threads are sending or posting events to Event Admin, the order in which those events arrive at the broker depends on low level timing factors and so the delivery order to handlers is not guaranteed. That is, if a second thread sends events D, E and F in that order, then each handler may see the order $ABCDEF$ or perhaps $DEFABC$, $ADBECF$, $DAEBFC$ or some other interleaving. But they will not see $CABDEF$, $ADCFBE$ etc., since these examples violate the internal ordering of messages within each thread.

7.5.7. Reliable Delivery

Event Admin attempts to make the delivery of events reliable when faced with misbehaving handlers. If a handler throws an exception then Event Admin will catch it and log it the OSGi Log Service, if it is available. Then it will continue delivering the event to other handlers. Note it does not attempt to catch Errors such as `OutOfMemoryError`, `LinkageError` etc. Also because the `EventHandler.handleEvent` method does not declare any checked exceptions in a `throws` clause, we can only throw subclasses of `RuntimeException` from our handlers.

Some implementations of Event Admin may also attempt to detect handlers that have stalled, for example in an infinite loop or deadlock. They may choose "blacklist" particular misbehaving handlers so they no longer receive any events. However, this feature is optional and not all Event Admin implementations support it.

7.6. Exercises

1. The notification mechanism shown in Section 7.4 is somewhat inefficient because all mailbox events are sent to all registered mailbox listeners, but many of the listeners (e.g., the message tables) are only interested in the events for a *single* mailbox. Extend this code to implement filtering based on mailbox name. Each listener should be able to listen to just one mailbox or all mailboxes. Hint: `WhiteboardHelper` will need to be extended to accept a filter expression.

8. The Extender Model

We saw in Chapter 4 how services can be used to extend the functionality of an application at runtime with new Java classes. Sometimes though we want extensibility of another sort: the ability to add artifacts other than executable code.

A good example is the help system of our Mailbox Reader sample application. We are able to extend the functionality of the application by plugging in new bundles, such as bundles providing new mailbox types, which are naturally implemented with services. But we also need to provide documentation for these new features. Therefore our help system needs to be extensible, too.

Let's assume that help documents will be in HTML format. Documentation is usually static, so a plain HTML file will suffice, i.e. the content of the HTML document does not need to be generated on-the-fly. Now, we *could* use a service to register the existence of a file, by declaring an interface as in Listing 8.1.

Listing 8.1 Help Provider Service Interface

```
1 public interface HelpDocumentProvider {
2     /**
3      * Return the URL of an HTML file.
4      */
5     URL getHelpDocumentURL();
6 }
```

But this would be a very cumbersome way to accomplish such a simple task. Any bundle wishing to provide help would have to implement the interface, and also implement `BundleActivator` in order to instantiate the class and register it as a service... all this just to provide a simple URL!

It would be a lot easier if we didn't need to "register" the document at all, but instead the help system simply found it inside our bundle and processed it automatically. Perhaps we could give a hint to the help system about which HTML files are intended for its use, e.g. by placing those files in a `help` subdirectory of the bundle.

It turns out this is quite easy to achieve, because bundles can see the contents of other bundles. So we can write a bundle that scans other bundles, looking

for help documentation and adding it to the central index. We call this kind of bundle an *extender*.

An extender bundle is one which scans the contents and/or headers of other bundles and performs some action on their behalf. This is a very useful and common pattern in OSGi.

8.1. Looking for Bundle Entries

Let's take a look a the code required to look at the contents of another bundle. We have two tools for doing this, both of them methods on the `Bundle` interface. The first is `getEntry`, which takes a path string such as `help/index.html` and returns a URL reference to that entry in the bundle if it exists; the contents of the entry can be read from the URL by calling `openStream` on it. The second is `getEntryPaths` which takes a prefix string and returns an enumeration of all the bundle entry paths starting with that prefix.

The method in Listing 8.2 will scan a bundle and return a list of URLs pointing to the HTML documents we are interested in, i.e. those that have been placed under the `help` subdirectory. Notice, again, the necessity of doing a null-check on the result of the query method: OSGi never returns an empty array or collection type. However our method *does* return an empty List of URLs when no matches are found.

Listing 8.2 Scanning a Bundle for Help Documents

```
1   private List<URL> scanForHelpDocs(Bundle bundle) {
2       List<URL> result;
3       Enumeration<?> entries = bundle.getEntryPaths("help");
4       if (entries != null) {
5           result = new ArrayList<URL>();
6           while (entries.hasMoreElements()) {
7               String entry = (String) entries.nextElement();
8               if (entry.endsWith(".html")) {
9                   result.add(bundle.getEntry(entry));
10              }
11          }
12      } else {
13          result = Collections.emptyList();
14      }
15      return result;
16  }
```

This code works just fine but is a little limited. The problem is we need to take the information returned by `scanForHelpDocs` and use it to create an index page in our help system, but all we know about each document is its filename. Therefore all we could show in the help index would be a list of filenames, which is not likely to be very helpful.

We could remedy this by asking help providers to list all the HTML documents they provide explicitly and supply a title using a simple properties file as shown in Listing 8.3.

Listing 8.3 A Help Index File, `index.properties`

```
introduction=Introduction
first_steps=First Steps in OSGi
dependencies=Bundle Dependencies
intro_services=Introduction to Services
```

We can interpret this as follows: the file named `help/introduction.html` has the title "Introduction"; the file named `help/first_steps.html` has the title "First Steps in OSGi"; and so on.

Let's assume that this properties file can be found at the location `help/index.properties`. The code in Listing 8.4 is an improved version of the scanning method, which now returns not just a list of document URLs but a list of URLs and titles:

Listing 8.4 Scanning a Bundle for Help Documents with Titles (1)

```
1   private List<Pair<URL, String>> scanForHelpDocsWithTitles(
2           Bundle bundle) throws IOException {
3       // Find the index file entry; exit if not found
4       URL indexEntry = bundle.getEntry("help/index.properties");
5       if (indexEntry == null) {
6           return Collections.emptyList();
7       }
8
9       // Load the index file as a Properties object
10      Properties indexProps = new Properties();
11      InputStream stream = null;
12      try {
13          stream = indexEntry.openStream();
14          indexProps.load(stream);
15      } finally {
16          if (stream != null)
17              stream.close();
18      }
19
20      // Iterate through the files
21      List<Pair<URL, String>> result =
22              new ArrayList<Pair<URL, String>>(indexProps.size());
23      Enumeration<?> names = indexProps.propertyNames();
24      while (names.hasMoreElements()) {
25          String name = (String) names.nextElement();
26          String title = indexProps.getProperty(name);
27
28          URL entry = bundle.getEntry("help/" + name + ".html");
29          if (entry != null) {
30              result.add(new Pair<URL, String>(entry, title));
31          }
32      }
33
34      return result;
35  }
```

In order to represent a document URL together with a title, we defined a simple `Pair` class, shown in Listing 8.5. This uses Java 5 Generics to hold two objects of arbitrary types, similar to a *tuple* type found in other languages.

Listing 8.5 The `Pair` Class

```
1  package org.osgi.book.utils;
2
3  public class Pair<A, B> {
4      private final A first;
5      private final B second;
6
7      public Pair(A first, B second) {
8          this.first = first;
9          this.second = second;
10     }
11
12     public A getFirst() {
13         return first;
14     }
15
16     public B getSecond() {
17         return second;
18     }
19
20     // Omitted: hashCode, equals and toString implementations
21 }
```

8.2. Inspecting Headers

The previous example is already very convenient for help providers, but a little inflexible. What if the `help` subdirectory of a bundle is already used for something else, or if we want to give the index file a name other than `index.properties`? It would be helpful if we could somehow provide the path to the index file, so we didn't have to assume a fixed path. But where can we put this information? If we put it in another properties file, then we still need to find *that* file.

However, there is already one file that must appear in the bundle at a fixed location, and that is `META-INF/MANIFEST.MF`. It is also flexible: we are free to add new headers to the manifest so long as their name does not clash with an existing Java or OSGi header. So, we can ask help providers to insert a reference to their index file using a new header name that we define. The provider's manifest might therefore look like something like Listing 8.6.

Fortunately we don't need to parse the manifest ourselves because just like the contents of a bundle, manifest headers are accessible through the `Bundle` interface. In this case we can call the `getHeaders` method to get a dictionary of all the headers. We can then use this to pick out the value of the `Help-Index` header — note that all the other headers can also be read using this

Listing 8.6 `MANIFEST.MF` for a Help Provider

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-SymbolicName: org.foo.help
Bundle-Version: 1.0.0
Help-Index: docs/help.properties
```

method, including those defined by OSGi and others.

Listing 8.7 shows yet another version of the scanning method. There is still one assumption implicit in this code: the HTML documents are expected to be in the same directory as the index file. However we could remove this assumption by adding more information to either the index or the value of the `Help-Index` header. It might also help to use a more structured file format for the index, such as XML or JSON.

8.3. Bundle States

Somewhere we need to actually call one of the above scanner methods, and do something with the results. Since they all take a `Bundle` object as input, it seems we need to iterate over the set of current bundles. But we should know by now that this is not enough: we also need to be informed when a new bundle appears or when a bundle we were previously looking at goes away.

Consider again the lifecycle of a bundle, which we first saw in Section 2.8 and Figure 2.3. Should we allow a bundle in *any* state to contribute help documents, or should we restrict ourselves to a subset of states? This is an important question, for which all extender bundles need to come up with a good answer. Bundle states are not really exclusive: some bundle states can be considered to "include" other states. For example an ACTIVE bundle is also RESOLVED and INSTALLED. In other words, an extender that is interested in RESOLVED bundles should be interested in STARTING, ACTIVE and STOPPING bundles as well. The inclusion relationships of all the states are illustrated in Figure 8.1.

If we include bundles that are merely INSTALLED then *every* bundle will be involved in our scan, so users needn't do anything at all beyond installing a bundle in order for it to appear in the index. But this can have negative consequences. For example, some bundles may not able to resolve because of missing dependencies or other unsatisfied constraints, but they will still be picked by an extender that includes the INSTALLED state. This isn't always a problem, but in our help system it might be. Merely providing documents doesn't need any dependencies, but suppose our documentation is embedded in a bundle that provides actual functionality as well. If the functionality

Listing 8.7 Scanning a Bundle for Help Documents with Titles (2)

```
1   private static final String HELP_INDEX_BUNDLE_HEADER = "Help-Index";

3   private List<Pair<URL, String>> scanForHelpDocsWithTitle(
4         Bundle bundle) throws IOException, HelpScannerException {
5      @SuppressWarnings("unchecked")
6      Dictionary<String, String> headers = bundle.getHeaders();

8      // Find the index file entry; exit if not found
9      String indexPath = headers.get(HELP_INDEX_BUNDLE_HEADER);
10     if (indexPath == null)
11        return Collections.emptyList();
12     URL indexEntry = bundle.getEntry(indexPath);
13     if (indexEntry == null)
14        throw new HelpScannerException("Entry not found: " + indexPath);

16     // Calculate the directory prefix
17     int slashIndex = indexPath.lastIndexOf('/');
18     String prefix = (slashIndex == -1)
19           ? "" : indexPath.substring(0, slashIndex);

21     // Load the index file as a Properties object
22     Properties indexProps = new Properties();
23     InputStream stream = null;
24     try {
25        stream = indexEntry.openStream();
26        indexProps.load(stream);
27     } finally {
28        if (stream != null) stream.close();
29     }

31     // Iterate through the files
32     List<Pair<URL, String>> result =
33           new ArrayList<Pair<URL, String>>(indexProps.size());
34     Enumeration<?> names = indexProps.propertyNames();
35     while (names.hasMoreElements()) {
36        String name = (String) names.nextElement();
37        String title = indexProps.getProperty(name);

39        URL entry = bundle.getEntry(prefix + "/" + name + ".html");
40        if (entry != null) result.add(new Pair<URL, String>(entry, title));
41     }
42     return result;
43  }
```

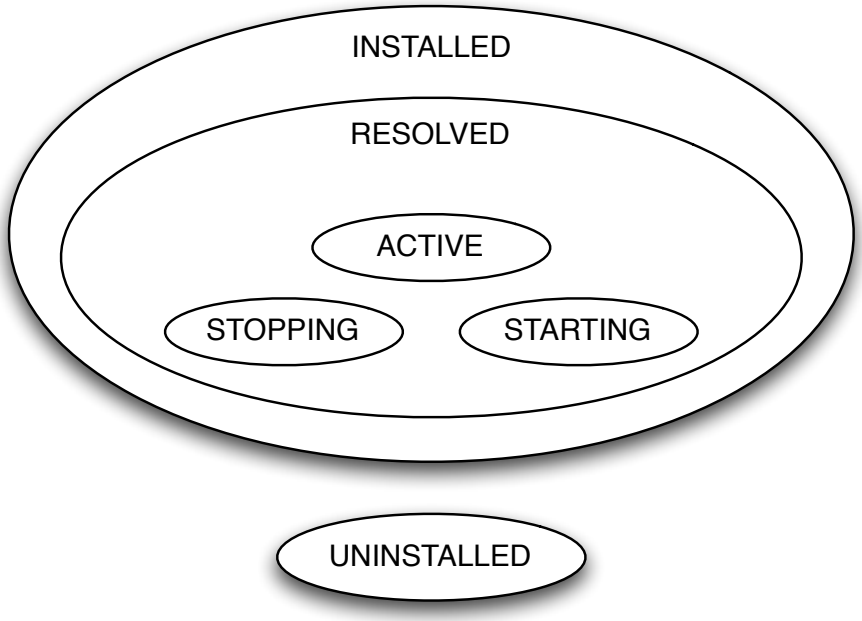


Figure 8.1.: Inclusion Relationships of Bundle States

of the bundle is not available due to a missing dependency, then it could be confusing to display help for it.

So, as an alternative, we can exclude the INSTALLED state and include only RESOLVED (along with STARTING, ACTIVE and STOPPING) bundles. This ensures only bundles with satisfied dependencies are included by our extender.

But both INSTALLED and RESOLVED have another problem: it is difficult to exit from those states. Suppose we want the ability for certain bundles to be explicitly removed from consideration by an extender. If the extender is looking for INSTALLED bundles, there is obviously only one way to take a bundle out of consideration: uninstall it. But if the extender is looking for RESOLVED bundles, we are still required to uninstall the bundle. The reason for this is there is no explicit "unresolve" method or command in OSGi; since we cannot ask for the bundle to be moved from RESOLVED back to INSTALLED, so we have to use the heavy-handed approach of uninstalling it completely.

The last alternative is to look at ACTIVE bundles only. This has the advantage that we can easily move bundles in and out of consideration by the extender simply by starting and stopping them. For this reason most examples of the extender pattern use the ACTIVE state (with one notable exception, as we will see in Section 8.6).

8.4. Using a Bundle Tracker

For the Help system example, we choose to look at the ACTIVE state only. Therefore we need to know about both the bundles already in ACTIVE state when our extender bundle starts, and also keep track of subsequently activated/deactivated bundles. Just as `ServiceTracker` made our lives simpler for tracking services, the class `BundleTracker` simplifies the task of tracking bundles, and by design it closely resembles `ServiceTracker` in usage. `BundleTracker` is a new part of the OSGi Compendium specification since Release 4.2 that explicitly supports the development of extender bundles.

Listing 8.8 shows the full code for the help system extender (with the exception of the scanning method which was already given in Listing 8.7), and it illustrates most of the points that need to be made about `BundleTracker`.

First, we can can create our own tracker customisation either by extending `BundleTracker` or by implementing `BundleTrackerCustomizer`. This is analogous `ServiceTracker` and `ServiceTrackerCustomizer`. In this case we extend `BundleTracker` itself because it makes the activator code slightly shorter.

Listing 8.8 The `HelpExtender` Class

```
1  package org.osgi.book.help.extender;
2
3  import java.io.IOException;
4  import java.io.InputStream;
5  import java.net.URL;
6  import java.util.*;
7  import java.util.concurrent.ConcurrentHashMap;
8
9  import org.osgi.book.utils.Pair;
10 import org.osgi.framework.*;
11 import org.osgi.service.log.LogService;
12 import org.osgi.util.tracker.BundleTracker;
13
14 public class HelpExtender extends BundleTracker {
15
16    private final Map<Long, List<Pair<URL, String>>> documentMap
17          = new ConcurrentHashMap<Long, List<Pair<URL, String>>>();
18    private final LogService log;
19
20    public HelpExtender(BundleContext context, LogService log) {
21       super(context, Bundle.ACTIVE, null);
22       this.log = log;
23    }
24
25    public List<Pair<URL, String>> listHelpDocs() {
26       List<Pair<URL, String>> result = new ArrayList<Pair<URL, String>>();
27       for (List<Pair<URL, String>> list : documentMap.values())
28          result.addAll(list);
29       return result;
30    }
31
32    @Override
33    public Object addingBundle(Bundle bundle, BundleEvent event) {
34       Bundle result = null;
35       long id = bundle.getBundleId();
36       try {
37          List<Pair<URL, String>> docs = scanForHelpDocsWithTitle(bundle);
38          if (!docs.isEmpty()) {
39             documentMap.put(id, docs);
40             result = bundle;
41          }
42       } catch (IOException e) {
43          log.log(LogService.LOG_ERROR, "IO error in bundle "
44                + bundle.getLocation(), e);
45       } catch (HelpScannerException e) {
46          log.log(LogService.LOG_ERROR, "Error in bundle "
47                + bundle.getLocation(), e);
48       }
49       return result;
50    }
51
52    @Override
53    public void removedBundle(Bundle bundle, BundleEvent event, Object obj) {
54       documentMap.remove(bundle.getBundleId());
55    }
56
57    // Omitted: scanForHelpDocsWithTitle method from previous section
58    // ...
```

The second parameter of `BundleTracker`'s constructor is a filter in the form of a bundle state mask. We can select multiple bundle states by performing a bitwise-OR of all the desired states. Then the tracker will call `addingBundle` only for those bundles that are either in one of the specified states or in the process of moving to one of them. The `removedBundle` method will be called when a tracked bundle leaves the desired state(s), and `modifiedBundle` is called when a tracked bundle moves from one state to another but still matches the state mask.

The return type of `addingBundle` is `Object` and we can return anything we like; that object will then be given back to us in the `removedBundle` and `modifiedBundle` methods. Conventionally we return the `Bundle` object. If we return `null` then the bundle will no longer be tracked; we use this effect in our `addingBundle` method, which only returns non-null when a bundle actually has help documents declared in it.

Like `ServiceTracker`, a `BundleTracker` needs to be opened by calling the `open` method before it will do anything. Forgetting to open the tracker is a significant source of bugs.

8.4.1. Testing the Help Extender

To test this extender, we will write a shell command that prints a list of the available help documents; this is shown in Listing 8.9. The corresponding activator and `bnd` descriptor are in Listing 8.10.

If we install and start the resulting bundle, we should now be able to call the `helpDocs` command:

```
osgi> helpDocs
0 documents(s) found
```

Of course, we haven't built a bundle yet that provides any help documentation! Let's do that now. If we create a directory in our project called `resources/help_sample`, we can put our index file and help HTML there and include them in a bundle using the `Include-Resource` instruction to `bnd`, as shown in Listing 8.11. Note the assignment-style format of the `Include-Resource` header; this tells `bnd` to copy the contents of the directory on the right hand side into a directory named `docs` in the bundle. If we omitted "`docs=`" then the files would be copied to the root of the bundle.

Now if we install and start the `help_sample` bundle and rerun the `helpDocs` command, we should see the following output:

```
osgi> helpDocs
4 document(s) found
Introduction (bundleentry://3.fwk15131397/docs/introduction.html)
First Steps in OSGi (bundleentry://3.fwk15131397/docs/first_steps.html)
Bundle Dependencies (bundleentry://3.fwk15131397/docs/dependencies.html)
...
```

Listing 8.9 Shell Command for Testing the Help Extender

```
1  package org.osgi.book.help.extender;
2
3  import java.net.URL;
4  import java.util.List;
5
6  import org.eclipse.osgi.framework.console.CommandInterpreter;
7  import org.eclipse.osgi.framework.console.CommandProvider;
8  import org.osgi.book.utils.Pair;
9
10 public class HelpListCommand implements CommandProvider {
11
12    private final HelpExtender extender;
13
14    public HelpListCommand(HelpExtender extender) {
15       this.extender = extender;
16    }
17
18    public String getHelp() {
19       return "\t" + "helpDocs - List currently available help docs";
20    }
21
22    public void _helpDocs(CommandInterpreter ci) {
23       List<Pair<URL,String>> docs = extender.listHelpDocs();
24       ci.println(docs.size() + " document(s) found");
25       for (Pair<URL, String> pair : docs) {
26          ci.println(pair.getSecond() + " (" + pair.getFirst() + ")");
27       }
28    }
29 }
```

8.5. Bundle Events and Asynchronous Listeners

The `BundleTracker` class is how *most* extender bundles should be implemented. However, there is unfortunately a small risk associated with the use of `BundleTracker`s because every tracker is notified of every bundle state change *synchronously*, i.e. in the same thread in which the state change is being made. This means two problems can arise: first, if a tracker performs its work too slowly in the `addingBundle` method then this can slow down the entire system; second, the bundle tracker must be multi-thread safe, since bundle events can happen on any thread.

Sometimes we wish to know about the state of bundles in the system but we do not necessarily need to know instantaneously about every change. Also we may not care about the transitory states `STARTING` and `STOPPING`. In that case we may wish to implement an asynchronous `BundleListener` that is notified of the gross changes in state on a separate thread shortly after they happen.

Asynchronous `BundleListener`s have the advantage that the framework guarantees never to call the `bundleChanged` method concurrently from multiple threads, so they can be written without worrying about locking fields or performing operations atomically. However it does *not* guarantee to always call `bundleChanged` from the *same* thread, so we still need to follow safe publi-

Listing 8.10 Activator & Bnd Descriptor for the Help Extender Bundle

```
1  package org.osgi.book.help.extender;

3  import org.eclipse.osgi.framework.console.CommandProvider;
4  import org.osgi.book.utils.LogTracker;
5  import org.osgi.framework.*;

7  public class HelpExtenderActivator implements BundleActivator {

9     private volatile LogTracker log;
10    private volatile HelpExtender extender;
11    private volatile ServiceRegistration cmdSvcReg;

13    public void start(BundleContext context) throws Exception {
14       log = new LogTracker(context);
15       log.open();
16       extender = new HelpExtender(context, log);
17       extender.open();

19       HelpListCommand command = new HelpListCommand(extender);
20       cmdSvcReg = context.registerService(CommandProvider.class.getName(),
21             command, null);
22    }

24    public void stop(BundleContext context) throws Exception {
25       cmdSvcReg.unregister();
26       extender.close();
27       log.close();
28    }

30 }
```

```
update 1# helpextender.bnd
Private-Package: org.osgi.book.help.extender, org.osgi.book.utils
Bundle-Activator: org.osgi.book.help.extender.HelpExtenderActivator
```

Listing 8.11 Bnd Descriptor for a Sample Help Provider

```
# help_sample.bnd
Help-Index: docs/index.properties
Include-Resource: docs=resources/help_sample
```

cation idioms as decribed in Section 6.3. Another advantage is that, since the callback to a `BundleListener` is executing in a thread dedicated to that purpose, we can be a little more liberal about performing blocking operations and other computations that would be too long-running for execution by a `BundleTracker`. We still need to be cautious though, since we could hold up the delivery of bundle events to other listeners, so truly long-running operations should be performed in an thread that we explicitly create.

Another listener interface exists, `SynchronousBundleListener`, which as the name suggests delivers the bundle events synchronously. The `BundleTracker` class is implemented as a `SynchronousBundleListener`. To illustrate the difference between synchronous and asychronous event delivery, see Figures 8.2 and 8.3, which show UML-like sequence diagrams for `SynchronousBundleListener` and (asynchronous) `BundleListener` respectively.

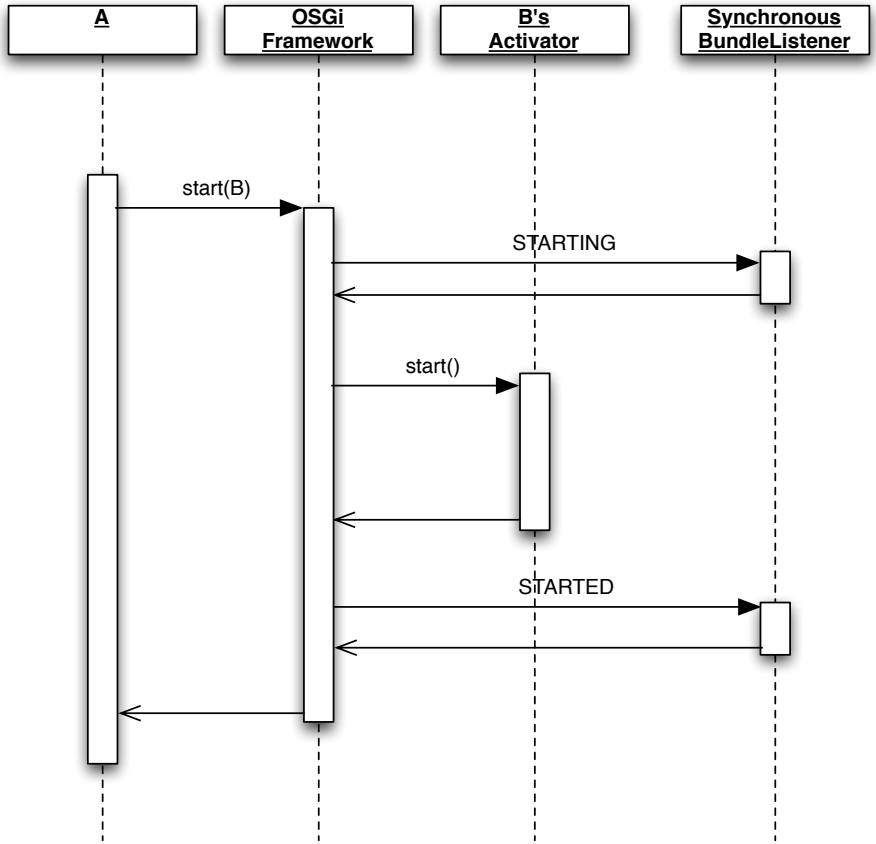


Figure 8.2.: Synchronous Event Delivery when Starting a Bundle

A disadvantage of `BundleListeners` is that they can lead us to act on stale

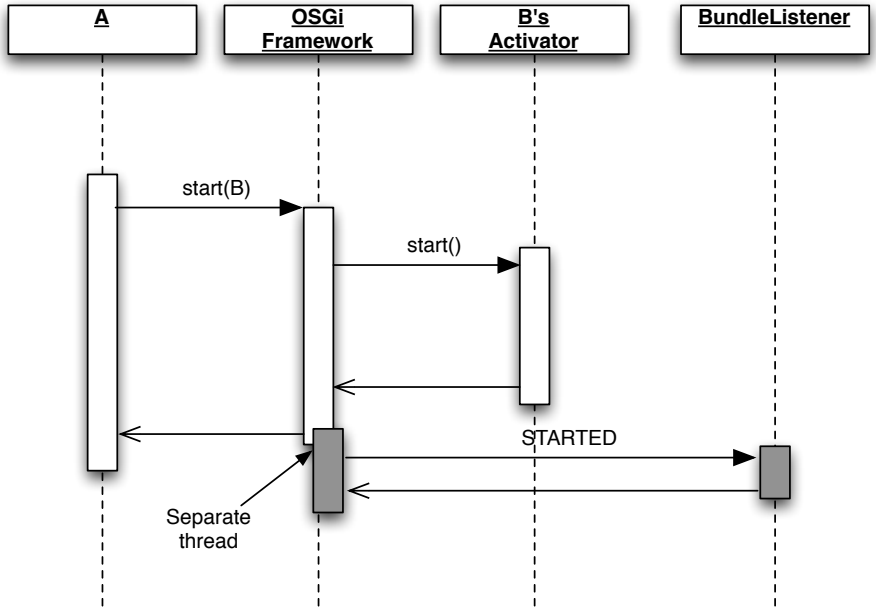


Figure 8.3.: Asynchronous Event Delivery after Starting a Bundle

information. If we had built our Help extender as a `BundleListener` rather than using `BundleTracker`, we would only hear about a bundle being uninstalled sometime after it was uninstalled. Therefore a user might request a help document from a bundle which no longer exists. This would lead to an error which we would need to catch and display appopriately to the user.

8.6. The Eclipse Extension Registry

As mentioned in Section **??**, the Eclipse IDE and platform are based on OSGi. However, Eclipse currently makes very little use of services, mainly for historical reasons.

Eclipse did not always use OSGi: in fact it only started using OSGi in version 3.0, which was released in 2004, nearly three years after the first release. Until version 3.0, Eclipse used its own custom-built module system which was somewhat similar to OSGi, but substantially less powerful and robust. Also it used it's own late-binding mechanism called the "extension registry", which achieves roughly the same goal as OSGi's services but in a very different way. When Eclipse switched to OSGi, it threw out the old module system, but it did *not* throw out the extension registry, because to do so would have rendered

almost all existing Eclipse plug-ins useless. By that time there were already many thousands of plug-ins for Eclipse, and there was no feasible way to offer a compatibility layer that could commute extension registry based code into services code. Therefore the extension registry continues to be the predominant model for late-binding in Eclipse plug-ins and RCP applications.

Today, the extension registry is implemented as an extender bundle. Bundles are able to declare both *extension points* and *extensions* through the special file `plugin.xml` which must appear at the root of a bundle. This file used to be the central file of Eclipse's old module system, as it listed the dependencies and exports of a plug-in, but today its role is limited to the extension registry[1]

Listing 8.12 An Eclipse `plugin.xml` File

```
<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.2"?>
<plugin>

   <extension-point id="org.foo.commands"
                    name="Commands"
                    schema="schema/org.foo.commands.exsd"/>

   <extension point="org.foo.commands">
      <command id="org.bar.mycommand"
               class="org.bar.MyCommand">
      </command>
   </extension>

</plugin>
```

Listing 8.12 shows an example of a `plugin.xml` file, which in this case declares both an extension point and an extension in the same bundle. An extension *point* is a place where functionality can be contributed, and an *extension* is the declaration of that contributed functionality. They are like a socket and a plug, respectively. In this example the extension contributes functionality into the `org.foo.commands` extension point, which is defined in the *same* bundle: there is nothing to stop this and it can be useful in certain situations.

Usually one does not directly edit `plugin.xml` as XML in a text editor. Instead there are powerful tools in Eclipse PDE to edit the extensions and extension points graphically. They are shown in Figures 8.4 and 8.5.

We will not go into much detail on how the extension registry is used, as this subject is well documented in various books and articles about Eclipse. However we should already be able to guess how the registry bundle works by scanning bundles for the presence of a `plugin.xml` file, reading the extension and extension point declarations therein, and merging them into a global map. It is not really so different from the help extender that we implemented in the previous section.

[1] One still occasionally sees a `plugin.xml` that includes the old module system declarations; they are supported by a compatibility layer.

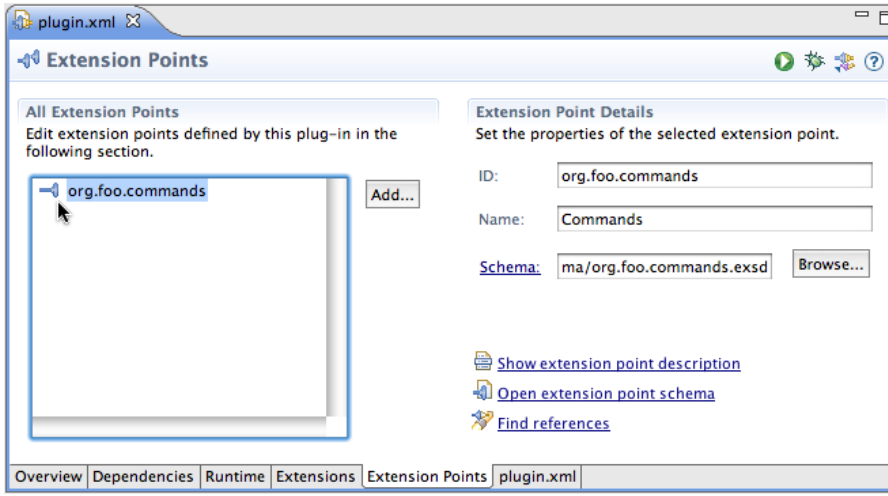


Figure 8.4.: Editing an Extension Point in Eclipse PDE

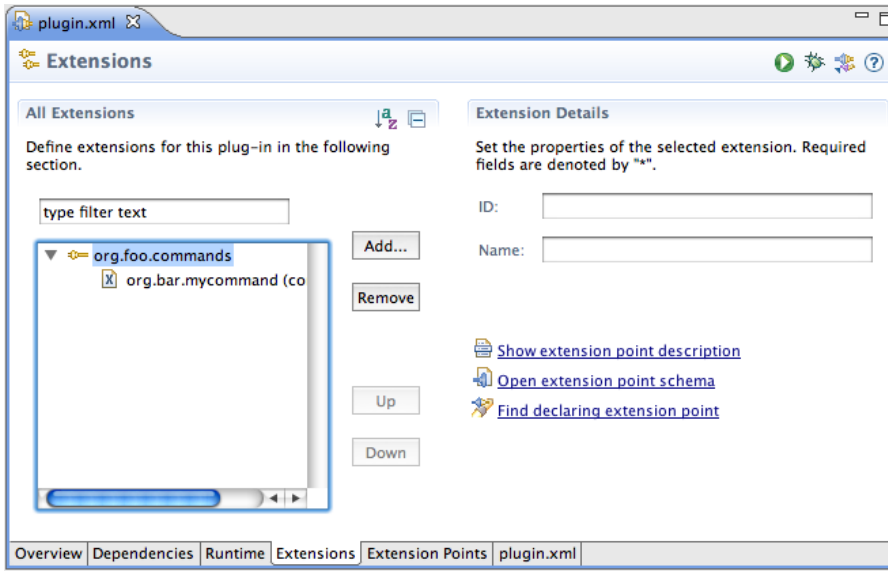


Figure 8.5.: Editing an Extension in Eclipse PDE

However, rather than considering only ACTIVE bundles, the extension registry considers all RESOLVED bundles. The result — as we would expect from the discussion in Section ?? — is that we don't have a lot of control over the content of the registry: *all* extensions in all RESOLVED bundles will contribute to the running application, and the only way we can remove those contributed features is by fully uninstalling the bundle.

8.7. Impersonating a Bundle

In Section 2.10 we learned that the only way to obtain a `BundleContext` and interact with the framework was by implementing a `BundleActivator`. However we also learned in a footnote that that was not strictly true: another way is to define an extender bundle that interacts with the framework *on behalf of* another bundle. By doing this we can define standard patterns of interaction with the framework.

For example, suppose we have a lot of bundles that register Mailbox services. The activator code to create the Mailbox instances and register them with the service registry can be quite repetitive, and it might be useful to use the extender model to define an alternative based on declarations. Thus we could offer a Mailbox service simply by adding to our `MANIFEST.MF`:

```
Mailbox-ServiceClass: org.example.MyMailbox
```

But there is a catch: if we register the Mailbox services using the `BundleContext` of our extender bundle, then they will all appear to be services offered by that extender, not by the real bundle that contains the Mailbox implementation. This would be wrong. However, there is a solution: in OSGi Release 4.1 a new method was added to the `Bundle` interface called `getBundleContext`. This method allows our extender to register services as if they were registered by the target bundle — in other words it allows us to impersonate bundles.

It should be noted that this is an advanced technique. There are many factors that need to be taken into account when writing an impersonating extender, so it should *not* be undertaken lightly. Also note that the example in this section is simply a limited version of the standard OSGi facility called Declarative Services (which we will look at in detail Chapter ??) so we would not wish to do this for real. Nevertheless, it is useful to understand what is going on when *using* an impersonating extender.

Listing 8.13 shows the code for the extender. Although it is quite long, the bulk is taken up with using Java reflection to load the specified Mailbox class by name, instantiating it, and handling the myriad checked exceptions that Java gives us whenever we use reflection. Note that we must ask the *target* bundle to perform the class load, by calling the `Bundle.loadClass` method, since in

Listing 8.13 Mailbox Service Extender

```
1  package org.osgi.book.extender.service;

3  import java.util.Map;
4  import java.util.concurrent.ConcurrentHashMap;

6  import org.osgi.book.reader.api.Mailbox;
7  import org.osgi.framework.*;
8  import org.osgi.service.log.LogService;
9  import org.osgi.util.tracker.BundleTracker;

11 public class MailboxServiceExtender extends BundleTracker {

13    private static final String SVC_HEADER = "Mailbox-ServiceClass";
14    private final Map<String, ServiceRegistration> registrations
15       = new ConcurrentHashMap<String, ServiceRegistration>();
16    private final LogService log;

18    public MailboxServiceExtender(BundleContext ctx, LogService log) {
19       super(ctx, Bundle.ACTIVE, null);
20       this.log = log;
21    }

23    @Override
24    public Object addingBundle(Bundle bundle, BundleEvent ev) {
25       Bundle result = null;
26       String className = (String) bundle.getHeaders().get(SVC_HEADER);
27       if (className != null) {
28          try {
29             Class<?> svcClass = bundle.loadClass(className);
30             if (!Mailbox.class.isAssignableFrom(svcClass)) {
31                log.log(LogService.LOG_ERROR,
32                      "Declared class is not an instance of Mailbox");
33             } else {
34                Object instance = svcClass.newInstance();
35                ServiceRegistration reg = bundle.getBundleContext()
36                      .registerService(Mailbox.class.getName(), instance,
37                            null);
38                registrations.put(bundle.getLocation(), reg);
39                result = bundle;
40             }
41          } catch (ClassNotFoundException e) {
42             log.log(LogService.LOG_ERROR, "Error creating service", e);
43          } catch (InstantiationException e) {
44             log.log(LogService.LOG_ERROR, "Error creating service", e);
45          } catch (IllegalAccessException e) {
46             log.log(LogService.LOG_ERROR, "Error creating service", e);
47          }
48       }
49       return result;
50    }

52    @Override
53    public void removedBundle(Bundle bundle, BundleEvent ev, Object ebj) {
54       ServiceRegistration reg;
55       reg = registrations.remove(bundle.getLocation());
56       if (reg != null) reg.unregister();
57    }
58 }
```

general the extender bundle will not know anything about the Mailbox implementation class. Also we assume that the specified class has a zero-argument constructor; if that is not the case then we will get an `InstantiationException`.

The activator and `bnd` descriptor for this extender is shown in Listing 8.14. In order to give the extender itself access to an instance of `LogService`, we use the `LogTracker` class from Section 4.10.1.

Listing 8.14 Activator and Bnd Descriptor for the Mailbox Service Extender

```
1  package org.osgi.book.extender.service;

3  import org.osgi.book.utils.LogTracker;
4  import org.osgi.framework.BundleActivator;
5  import org.osgi.framework.BundleContext;

7  public class MailboxServiceExtenderActivator
8        implements BundleActivator {

10    private volatile LogTracker logTracker;
11    private volatile MailboxServiceExtender extender;

13    public void start(BundleContext context) throws Exception {
14       logTracker = new LogTracker(context);
15       logTracker.open();

17       extender = new MailboxServiceExtender(context, logTracker);
18       extender.open();
19    }

21    public void stop(BundleContext context) throws Exception {
22       extender.close();
23       logTracker.close();
24    }

26 }
```

```
# mbox_svc_extender.bnd
Private-Package: org.osgi.book.extender.service,\
   org.osgi.book.utils
Bundle-Activator:\
    org.osgi.book.extender.service.MailboxServiceExtenderActivator
```

Now we can write a simple bundle that contains a minimal Mailbox implementation and declares it through a declaration in its `MANIFEST.MF`. Listing 8.15 shows the mailbox implementation and `bnd` descriptor we will use.

When we install and start the `sample_svc_extender` bundle, we should be able to see the registered mailbox service by typing the `services` command:

```
osgi> services
...
{org.osgi.book.reader.api.Mailbox}={service.id=30}
  Registered by bundle: sample_svc_extender_0.0.0 [6]
  No bundles using service.
```

Listing 8.15 Minimal Mailbox Class and Bnd Descriptor

```
1  package org.osgi.book.extender.service.sample;

3  import org.osgi.book.reader.api.*;

5  public class MyMailbox implements Mailbox {

7     public long[] getAllMessages() {
8        return new long[0];
9     }
10    public Message[] getMessages(long[] ids) {
11       return new Message[0];
12    }
13    public long[] getMessagesSince(long id) {
14       return new long[0];
15    }
16    public void markRead(boolean read, long[] ids) {
17    }
18 }
```

```
# sample_svc_extender.bnd
Private-Package: org.osgi.book.extender.service.sample
Mailbox-ServiceClass: org.osgi.book.extender.service.sample.MyMailbox
```

Here we see that a Mailbox service has been registered, apparently by the `sample_svc_extender` bundle, although it was really registered by the extender on behalf of that bundle.

To reiterate, impersonating other bundles is an advanced technique that must be used with care. In come cases it can be used to implement common patterns in a single location; however the example in this chapter could also have been implemented with straightforward Java inheritance, i.e. a simple activator that can be used as a base class.

8.8. Conclusion

The extender model is a very useful technique for allowing an application to be extended via resources or declarations, rather than via programmatic services. In some cases extenders can be used to implement common bundle implementation patterns by acting on behalf of other bundles.

9. Configuration and Metadata

In the previous chapter on the Extender Model, we saw one way to extend an application without writing executable code. There is another major aspect to application construction which also does not usually involve writing programmatic code: *configuration*. In this chapter we look at how to make OSGi-based applications configurable.

Why is it necessary to consider this subject in a separate chapter? We could imagine using the Extender Model to configure our applications: configuration data would be stored in flat files or XML documents embedded in bundles, and then read by the bundles that need the data. Unfortunately there are two significant problems with using such an approach:

1. Many parts of our application will need be configured. Each of the bundles that contains configurable elements would have to implement the Extender Model, e.g. by subclassing `BundleTracker`, which would be complex and create lots of overhead, and potentially inconsistency if bundles written by other developers use different file format or
2. Configuration files embedded in bundles are not easily accessible to users or administrators. Generating bundle JARs is a task best left to developers; administrators want to modify external resources stored in an agreed location in the file system, or perhaps in database records etc.

We therefore need to find a solution that makes it easy for many parts of our application to obtain configuration data, and also allows those configuration data to be stored in arbitrary ways outside of the bundles that comprise our application. This chapter discusses a standard solution that fulfils both requirements.

9.1. Configuration Admin

The OSGi Service Compendium specifies a standard service for configuration, called the *Configuration Admin*[1] service. Configuration Admin is often further abbreviated "CM" for historical reasons.

[1] The OSGi specification uses the abbreviation "admin" throughout, as if it were an actual word.

CM's primary task is to make it easy for many parts of an OSGi-based application to obtain configuration data. Many conventional approaches to this problem require each component to actively load its own configuration data, and though they typically provide an API that abstracts the physical mechanism involved, this still requires each part of the application to repeat largely the same code. CM reverses these responsibilities: components are required merely to advertise their interest in receiving configuration data. CM will then "push" configuration data to them[2]. This has two benefits: firstly, bundles need very little code to receive configuration data; and secondly, the physical loading and saving of data is be done in one place only and can easily be changed.

How does a bundle "advertise" its interest in receiving configuration data? Simply by publishing a service under one of two interfaces recognised by the Configuration Admin implementation. Therefore the design of CM is yet another example of the Whiteboard Pattern in use (Chapter 7). The objects that are advertised in this way are called configuration *targets*, and they may have other roles in the system such as exposing service interfaces or interacting with the user.

On the other side of the equation, CM needs a way to load and save configuration data in many different formats and on many different storage media. The requirements here are extremely diverse. In an enterprise server setting, the data is likely to be stored in one or more property files on the local disk in a system-wide location, but a desktop application may store its data as XML files in the user's home directory, or as entries in the Windows Registry. A mobile phone application might receive configuration data over the air in a highly compressed format from the cellular service provider. There is no way for CM to directly support all of these scenarios, so it supports *none of them* and instead offers a programmatic API for manipulating abstract configuration objects. That API is accessed by a so-called "Management Agent": a bundle that is responsible for loading and saving data using the specific format and storage medium employed by the application.

At its heart, CM is a broker responsible for matching up configuration objects with configuration targets. The matching is performed on the basis of a persistent identifier or PID, which uniquely identifies both the object and the target. When CM detects that a configuration object has the same PID as a configuration target, it supplies the data therein to the target. This is illustrated in Figure 9.1.

An important aspect of CM is that it uses persistence to allow the management agent and the configuration targets to be disconnected across time. An agent may create a configuration at a certain time and, whether or not a configura-

[2]This "backwards" approach is sometimes called Inversion of Control (IoC), or more fancifully the *Hollywood Principle*: "don't call us, we'll call you".

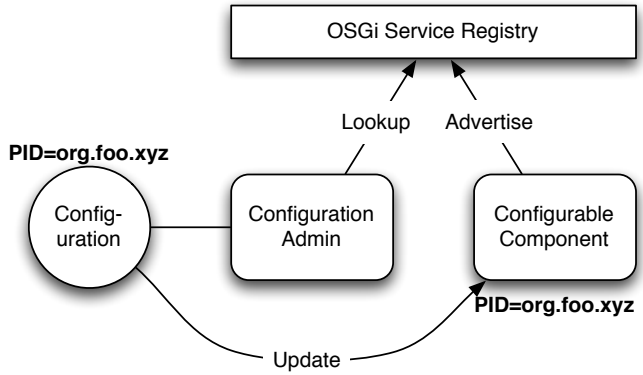


Figure 9.1.: Overview of the Configuration Admin Service

tion target exists at the that time with a matching PID, the configuration will be saved by CM[3]. A configuration target may appear much later, potentially even after the management agent has been uninstalled or the OSGi framework has been shutdown and restarted, and it will at that time receive the data.

9.1.1. Audiences

There are two audiences for this chapter. The first comprises developers who wish to write configurable objects, i.e. objects which receive configuration data from CM. This audience needs to learn how to advertise objects to CM and how to process the data received.

The second audience comprises those developers who need to supply configuration data to CM using a specific data format and storage medium – in other words, those who need to develop a management agent. This audience, which should be far smaller than the first, needs to learn how to create, update and remove configurations using the programmatic API offered by CM.

Since the first audience is so much bigger, its needs will be discussed first. In order to test the code we write, we still need a management agent, but we will use one that is freely available "off-the-shelf".

[3]The location to which CM saves this data is an internal detail of the CM implementation bundle. Most CM implementations simply use the persistent storage area provided by the OSGi framework via `BundleContext.getDataFile`.

9.2. Building Configurable Objects

9.2.1. Configured Singletons

Suppose we have a single instance of an object that needs some kind of configuration data, for example a `ServerConnection` object that opens and manages a connection to a server. Which server should it connect to, and how? At the very least we need to know the address and the port of the remote server, and optionally we may need to know a user name and password, local address and port, and so on. All of these are configuration data.

As described above, configurable components simply advertise themselves to Configuration Admin by publishing as a service. The interface they publish under is called `ManagedService` and it specifies just one method: `updated`, which takes a `Dictionary` full of properties and is permitted to throw a `ConfigurationException`.

The `ManagedService` interface is used for objects that are logically singletons. We do not use the term "singleton" in the strict sense of the coding pattern defined by the Gang of Four [10], whereby a private constructor and static field are used to ensure that only one instance of a particular class can possibly be created. Rather we mean that the instances are created and registered one at a time, under the control of our code. This distinction will be clearer when we look at "non-singleton" configured objects in a later section.

Our `ServerConnection` object could be implemented as shown in Listing 9.1, and registered as a service by the bundle activator as shown in Listing 9.2. We provide a persistent identifier (PID) by setting the `SERVICE_PID` property on the service — the value of the PID is arbitrary but must be globally unique, so it's a good idea to use a hierarchical namespace similar to Java package naming conventions. In this example, since there will only be one instance of the object, the fully-qualified class name is a good choice for the PID.

9.2.2. Running the Example with FileInstall

The example from the previous section will do nothing until two other things are in place: we need an active implementation of the Configuration Admin runtime, and we need a management agent that will supply the actual data.

Equinox supplies a bundle that implements the Configuration Admin runtime. It's name is `org.eclipse.equinox.cm` and it is included with the archive downloaded in Chapter 2, or it can be obtained separately from the main Equinox download site. Before proceeding, ensure this bundle is installed and ACTIVE. We also need the OSGi compendium bundle present, if it is not already, to supply the API packages for CM.

Listing 9.1 Configured Server Connection Singleton

```
1  package org.osgi.book.configadmin;
2
3  import java.util.Dictionary;
4
5  import org.osgi.service.cm.ConfigurationException;
6  import org.osgi.service.cm.ManagedService;
7
8  public class ServerConnection implements ManagedService {
9
10    private static final String PROP_HOST = "host";
11    private static final String PROP_PORT = "port";
12
13    private volatile String host;
14    private volatile Integer port;
15
16    public void updated(Dictionary properties)
17          throws ConfigurationException {
18       System.out.println("ServerConnection.updated: " + properties);
19       if (properties == null) {
20          host = null;
21          port = null;
22          System.out.println("Unconfigured server connection");
23       } else {
24          host = (String) properties.get(PROP_HOST);
25          if (host == null) {
26             throw new ConfigurationException(PROP_HOST,
27                   "Mandatory field");
28          }
29          String portStr = (String) properties.get(PROP_PORT);
30          if (portStr == null) {
31             throw new ConfigurationException(PROP_PORT,
32                   "Mandatory field");
33          }
34          try {
35             port = Integer.parseInt(portStr);
36          } catch (NumberFormatException e) {
37             throw new ConfigurationException(PROP_PORT,
38                   "Invalid number", e);
39          }
40          System.out.println("Configured server connection for host "
41                + host + ", port " + port);
42       }
43    }
44
45    public void openConnection() {
46       // ....
47    }
48 }
```

Listing 9.2 Registering the Server Connection Singleton

```
1  package org.osgi.book.configadmin;

3  import java.util.Properties;

5  import org.osgi.framework.BundleActivator;
6  import org.osgi.framework.BundleContext;
7  import org.osgi.framework.Constants;
8  import org.osgi.service.cm.ManagedService;

10 public class ServerConnectionActivator implements BundleActivator {

12    public void start(BundleContext context) throws Exception {
13       ServerConnection svrConn = new ServerConnection();

15       Properties props = new Properties();
16       props.put(Constants.SERVICE_PID,
17                 ServerConnection.class.getName());

19       context.registerService(ManagedService.class.getName(),
20                               svrConn, props);
21    }

23    public void stop(BundleContext context) throws Exception {
24    }
25 }
```

```
osgi> install file:osgi.cmpn.jar
Bundle id is 1

osgi> install file:org.eclipse.equinox.cm_1.0.100.v20090520-1800.jar
Bundle id is 2

osgi> start 2

osgi> install file:serverconn.jar
Bundle id is 3

osgi> start 3
Unconfigured server connection
```

What happened here is that the CM bundle started but could not find any configurable objects (that is, objects advertised under the `ManagedService` interface), and so it did nothing. Then we installed and started our configurable `ServerConnection` – CM detected this, but could not find any matching configuration data, so it called the `updated` method with a `null` parameter.

In order to test what happens when configuration data exists, we need a management agent to supply some configuration data. There is a simple yet very useful management agent available for us to use: FileInstall, which was originally developed by Peter Kriens but is now maintained by the Apache Felix project[4]. It can be downloaded from the following URL:

[4]Despite being a Felix project, it is a standard OSGi bundle that can run on any OSGi R4 framework, including Equinox.

http://felix.apache.org/site/downloads.cgi

Once downloaded it can be installed into Equinox and started as follows:

```
osgi> install file:org.apache.felix.fileinstall-1.0.0.jar
Bundle id is 4

osgi> start 4

osgi> felix.fileinstall.poll (ms)   2000
felix.fileinstall.dir             /home/neil/workspace/config/./load
felix.fileinstall.debug           -1
felix.fileinstall.bundles.new.start          true
```

FileInstall reads configuration data from files in the standard Java properties file format, and it works by polling a directory in the filesystem for files with a specific pattern in their name. By default, it looks in the `load` directory relative to the current working directory, and it polls every 2 seconds. As we can see it prints the full path of the directory it is polling, along with the polling interval and some other information, when it starts.

The names of the files that FileInstall looks for should be of the form `something.cfg`, where "something" is the PID with which the configuration data will be associated. So in order to create a configuration record with a PID of `org.osgi.book.configadmin.ServerConnection` — which will cause CM to provide it to our `ServerConnection` object — we should create a file named `org.osgi.book.configadmin.ServerConnection.cfg`. Checking the code for the `ServerConnection` object we see that the property names it expects are `host` and `port`. Therefore to connect to the host `example.org` on port 1234 we should create a file with the following contents:

```
host=example.org
port=1234
```

As soon as this file is created we should see the following message on our OSGi console, meaning that the `update` method has been called with the configuration data inside the file:

```
osgi> Configured server connection for host example.org, port 1234
```

We have to be a little careful how we create this configuration file. If we first create the file before entering its contents, then FileInstall will read the empty file and cause CM to pass an empty dictionary to our `update` method. This will trigger the error condition on lines 24 to 27:

```
2009-06-14 01:43:52.867 Error: host : Mandatory field
org.osgi.service.cm.ConfigurationException: host : Mandatory field
   at org.osgi.book.configadmin.ServerConnection.updated(...)
```

It may be better to create the file outside of the `load` directory and then copy it in when completed. But if course it's also important to code our configurable objects robustly in the face of bad configuration data: throw

`ConfigurationException` for missing or invalid mandatory properties, and use sensible defaults wherever possible.

We can change the file contents and FileInstall will detect the change and update our object with the new contents. It checks the "last modified" time stamp of the file, so the UNIX `touch` command (which updates th time stamp of a file without altering its content) can be used to force a configuration to be reloaded. Finally, deleting the file causes the the configuration to be deleted, and our `updated` method will be called again with a `null` dictionary.

9.2.3. Configured Singleton Services

We can configure any object using the technique shown in the previous section, but sometimes the object that needs to be configured is a service. For example, a mailbox service that connects to a database would need to know the connection parameters — e.g., host, port, user and password — for the database server. In this case, we can implement the `ManagedService` interface in addition to the existing service interface, as shown in Listing 9.3, and publish to the service registry under both interfaces, as shown in Listing 9.4.

Listing 9.3 A Configured Service

```
1  package org.osgi.book.configadmin;
2
3  import java.util.Dictionary;
4
5  import org.osgi.book.reader.api.Mailbox;
6  import org.osgi.book.reader.api.MailboxException;
7  import org.osgi.book.reader.api.Message;
8  import org.osgi.service.cm.ConfigurationException;
9  import org.osgi.service.cm.ManagedService;
10
11 public class ConfiguredDbMailbox implements Mailbox, ManagedService {
12
13    private static final String PROP_JDBC_URL = "jdbcUrl";
14
15    public void updated(Dictionary properties)
16          throws ConfigurationException {
17       if (properties != null) {
18          String url = (String) properties.get(PROP_JDBC_URL);
19          // Configure using the supplied properties...
20
21       }
22    }
23
24    public long[] getAllMessages() {
25       // ...
26       return null;
27
28    // Rest of the Mailbox API omitted...
29 }
```

This example shows the simplest approach to creating a configurable service,

Listing 9.4 Activator for the Configured Service

```
1  package org.osgi.book.configadmin;

3  import java.util.Properties;

5  import org.osgi.book.reader.api.Mailbox;
6  import org.osgi.framework.BundleActivator;
7  import org.osgi.framework.BundleContext;
8  import org.osgi.framework.Constants;
9  import org.osgi.service.cm.ManagedService;

11 public class ConfiguredDbMailboxActivator implements BundleActivator {

13    public void start(BundleContext context) throws Exception {
14       ConfiguredDbMailbox mailbox = new ConfiguredDbMailbox();

16       Properties props = new Properties();
17       props.put(Mailbox.NAME_PROPERTY, "cfgDbMailbox");
18       props.put(Constants.SERVICE_PID, ConfiguredDbMailbox.class
19             .getName());

21       String[] interfaces = new String[] { Mailbox.class.getName(),
22             ManagedService.class.getName() };
23       context.registerService(interfaces, mailbox, props);
24    }

26    public void stop(BundleContext context) throws Exception {
27    }
28 }
```

but sometimes this approach will not work. There is a problem caused by the timing of the call to `updated` by CM.

On line 23 of `ConfiguredDbMailboxActivator` (Listing 9.4), we register our service under both the `Mailbox` and the `ManagedService` interfaces at the same instant — this instant marks the start of a race. On the one hand, CM will notice the new `ManagedService` instance and try to find a configuration record with a matching PID; if it finds one, it will call the `updated` method to supply the configuration data. On the other hand, a client of the mailbox API — such as the GUI we developed in Chapter 5 — may see the new `Mailbox` service and immediately call a method on it such as `getAllMessages`. There is no way to predict which of these calls will happen first, so we have a problem: how does a mailbox that reads messages from a database respond to a request such as `getAllMessages` *before* it even knows which database to use?

More generally, the problem is how a configured object should behave in the absence of configuration data, and sometimes we can dodge the issue by assuming sensible default values. A polling component is a good example: the polling interval may be controlled by configuration, but in the absence of configuration we default to an interval of, say, 5 seconds. Sometimes though, there is just no sensible default value, and the database-backed mailbox is an example — we should not assume any particular address and port and hard-code them into our service, because any such assumptions are likely to be invalid

more often than they are valid.

There are various ways of solving this problem. We could block the thread that called the Mailbox method until CM calls `updated`, but this could easily result in deadlocks, and anyway CM may *never* call `updated` if there is no configuration record matching the PID of our `ManagedService`. So this option is dangerous and not recommended. Another issue is that CM might call `updated` with a parameter of `null`, indicating that the configuration data has been deleted or is no longer valid. In this case the service should be "de-configured", but it's possible that clients may call a mailbox method *after* the `updated(null)` call. Blocking doesn't help here either.

Another way is to return an error to the client that got in too early (or too late), e.g, by throwing a `MailboxException`. This is safer than blocking, but rather unfriendly: it is not the client's fault that the mailbox is not configured, and indeed the client has no way to tell the difference between a configured Mailbox service and an unconfigured one. As discussed in the section on composed services in Chapter 4, we should not register services that are "broken" due to unsatisfied dependencies. In that chapter we were talking about dependencies on other services; now we are talking about dependencies on configuration data. The principle is the same. So, a better way to handle this problem is to register the mailbox under the `Mailbox` interface *after* it receives configuration data from Configuration Admin, and not before. This implies a two-stage registration: first we register under `ManagedService` and then later register under `Mailbox`.

Listing 9.5 A Simplified Configured Mailbox Service

```
1  package org.osgi.book.configadmin;

3  import org.osgi.book.reader.api.Mailbox;
4  import org.osgi.book.reader.api.MailboxException;
5  import org.osgi.book.reader.api.Message;

7  public class ConfiguredDbMailbox2 implements Mailbox {

9      private final String jdbcUrl;

11     public ConfiguredDbMailbox2(String jdbcUrl) {
12         this.jdbcUrl = jdbcUrl;
13     }


16     // Mailbox API methods omitted...
17 }
```

Listing 9.5 shows how this can be done. All of the code for handling configuration has been removed from the mailbox implementation class, which now simply receives the configuration data it needs via a constructor parameter (though in reality it would probably need more parameters than this). The

configuration code has now moved to a separate class, `ConfiguredDbMailboxManager`, which is shown along with an activator that registers it in Listing 9.6.

In the manager, we instanqtiate a new mailbox each time the configuration changes, which means we must re-register with the service registry. The unregistration of the previous mailbox is done in a finally block as this ensures it is always unregistered, even when we throw a `ConfigurationException` due to bad configuration data. We also use an `AtomicReference` to save the `ServiceRegistration` object, which highlights some of the lessons learned about concurrency in Chapter 6.

Notice another useful aspect of this approach: configuration data can now be used to control the service properties of the mailbox service registration. In this example we set the "name" property of using a value loaded from the configuration.

9.2.4. Multiple Configured Objects

We referred to the above examples as logical "singletons" because they were created and registered individually. This was possible because we knew in advance that there should be one `ServerConnection`, one `ConfiguredDbMailbox` and so on. In general we can use the same technique for any number of instances so long as we know in advance how many there should be: if we need ten server connections, then we create and register ten `ServerConnection` instances.

However sometimes we cannot know in advance how many configured objects to create, because that fact is itself part of the configuration.

Consider for example an HTTP server which is able to listen on many ports, and each port listener may serve a different set of HTML pages. We might configure a listener with the port number for it to listen on, and the base directory of the HTML pages it should serve. But how many listeners should there be? This is a configuration parameter also. We can ship our server with a single pre-configured listener for port 80 (the default HTTP port), but an administrator might add an arbitrary number of additional listeners, for ports 8080, 8081 and so on. This scenario is difficult to achieve using the `ManagedService` interface because we need to know how many configured objects to create *before* any of them receive any configuration from CM.

Therefore CM offers an additional interface called `ManagedServiceFactory`, which can be used an an alternative to `ManagedService`. This interface is designed for creating arbitrarily many configured objects associated with a single "factory" PID. For our HTTP server example, we would register a single

Listing 9.6 Activator and Manager for the Simplified Configured Service

```
1  package org.osgi.book.configadmin;
2
3  import java.util.Dictionary;
4  import java.util.Properties;
5  import java.util.concurrent.atomic.AtomicReference;
6
7  import org.osgi.book.reader.api.Mailbox;
8  import org.osgi.framework.BundleActivator;
9  import org.osgi.framework.BundleContext;
10 import org.osgi.framework.Constants;
11 import org.osgi.framework.ServiceRegistration;
12 import org.osgi.service.cm.ConfigurationException;
13 import org.osgi.service.cm.ManagedService;
14
15 public class ConfiguredDbMailboxActivator2 implements BundleActivator {
16    public void start(BundleContext context) throws Exception {
17       ConfiguredDbMailboxManager manager =
18          new ConfiguredDbMailboxManager(context);
19       Properties props = new Properties();
20       props.put(Constants.SERVICE_PID, ConfiguredDbMailboxManager.class
21             .getName());
22       context.registerService(ManagedService.class.getName(),
23             manager, props);
24    }
25    public void stop(BundleContext context) throws Exception {
26    }
27 }
28
29 class ConfiguredDbMailboxManager implements ManagedService {
30
31    private static final String PROP_JDBC_URL = "jdbcUrl";
32    private static final String PROP_MBOX_NAME = "mboxName";
33
34    private final BundleContext context;
35    private final AtomicReference<ServiceRegistration> registration
36        = new AtomicReference<ServiceRegistration>();
37
38    public ConfiguredDbMailboxManager(BundleContext context) {
39       this.context = context;
40    }
41    public void updated(Dictionary dict) throws ConfigurationException {
42       ServiceRegistration newRegistration = null;
43       try {
44          if(dict != null) {
45             String jdbcUrl = (String) dict.get(PROP_JDBC_URL);
46             if(jdbcUrl == null)
47                throw new ConfigurationException(PROP_JDBC_URL,
48                                                 "Mandatory field");
49             String mboxName = (String) dict.get(PROP_MBOX_NAME);
50             if(mboxName == null)
51                throw new ConfigurationException(PROP_MBOX_NAME,
52                                                 "Mandatory field");
53             ConfiguredDbMailbox2 mailbox = new ConfiguredDbMailbox2(
54                                                          jdbcUrl);
55             Properties props = new Properties();
56             props.put(Mailbox.NAME_PROPERTY, mboxName);
57             newRegistration = context.registerService(
58                   Mailbox.class.getName(), mailbox, props);
59          }
60       } finally {
61          ServiceRegistration oldRegistration =
62             registration.getAndSet(newRegistration);
63          if(oldRegistration != null) oldRegistration.unregister();
64       }
65    }
66 }
```

instance of `ManagedServiceFactory` with a single factory PID, and it would create zero or more instances of the port listener objects.

Listing 9.7 The `ManagedServiceFactory` interface

```
1 public interface ManagedServiceFactory {

3     public String getName();

5     public void updated(String pid, Dictionary properties)
6               throws ConfigurationException;

8     public void deleted(String pid);
9 }
```

Listing 9.7 shows the methods of `ManagedServiceFactory`. It is a little more complex than `ManagedService`, but still fairly simple. Let's look at the methods in turn.

The first method `getName` is something of an anomaly — it should return a human-readable name for the factory.

The second method `updated` is essentially the same as the `updated` method of `ManagedService` in that it provides configuration data to us in the form of a dictionary. However there is an additional PID parameter, which is a unique PID for an individual configured object. In our HTTP server example, there would be one of these PIDs for *each* port listener. It's important to distinguish between this PID and the "factory" PID that we attach to the `ManagedServiceFactory` registration: the factory PID is supplied by us, but the individual PIDs are generated as unique strings by the CM implementation.

The third method `deleted` also takes an individual PID argument, and CM uses this method to instruct us to delete any configured objects that we have been created for the specified PID.

Note that there is no `create` method on this interface! The `updated` method serves for both creating new configured objects and updating existing ones. Our implementation of `ManagedServiceFactory` is required to atomically create and configure a new object if `updated` is called with a PID that we have not previously seen.

All of the above strongly suggests that we need to use some kind of map with the individual PID as the key. Indeed almost all implementations of `ManagedServiceFactory` do exactly that. Listing 9.8 shows a factory that creates instances of HTTP port listeners; the `HttpPortListener` class itself is omitted as it is not interesting. It is sufficient to know that it has a constructor taking a port number and base directory as parameters, and it has `start` and `stop` methods. We must be careful to stop the previously configured port listener for the associated PID, as it is not possible to have two listeners on the same port simultaneously. Of course another listener with a different PID

may be configured to use the same port, but we do not try to detect this error here, it will be handled in the `start` method of `HttpPortListener`.

On line 27 we check whether the dictionary is `null`. It is valid for a CM implementation to pass a `null` dictionary to the `updated` method of a factory, and this has a slightly different meaning from calling the `deleted` method. In our example we treat `null` in the same way as we treat deletion, but we must always be prepared to receive a `null`.

The activator for this example is omitted as it is trivial. It is required only to register an instance of `HttpPortListenerFactory` under the `ManagedServiceFactory` interface and supply a service PID. We again use the fully qualified class name, i.e. `org.osgi.book.configadmin.HttpPortListenerFactory`, as the PID.

9.2.5. Multiple Configured Objects with FileInstall

We can now use FileInstall to test the factory, and again we create a file named `something.cfg` in the standard properties file format. Now, each file corresponds to a single individual configuration record, i.e. the configuration for one `HttpPortListener`, with the filename prefixed by the factory PID. But we must be able to create multiple files for the same factory PID, so we add a hyphen and a suffix.

This suffix is referred to as the *alias*, so the name of the filename will be of the form `<factoryPid>-<alias>.cfg`. The alias is an identifier that is used by FileInstall to associate the file with a particular individual configuration record[5]. We supply the alias as an arbitrary string, and it can be something as simple as a number, but it may be better to use string which helps us to remember something about what the configuration record is for.

After installing and starting our HTTP listener bundle, we can create a file in the `load` directory with the following contents:

```
port=80
baseDir=./pages
```

This can be saved in a file named `org.osgi.book.configadmin.HttpPortListenerFactory-main.cfg`, because this is configuration for the "main" listener on port 80. When FileInstall detect this file it will instruct our factory to create a new object:

```
osgi> install file:http_listeners.jar
Bundle id is 5
```

[5]The alias is *not* the same as the PID of the individual configuration record. The PID is generated by CM and, as we will see shortly, FileInstall has no control over the generated values.

Listing 9.8 A `ManagedServiceFactory` for HTTP Port Listeners

```
1  package org.osgi.book.configadmin;

3  import java.io.File;
4  import java.util.Dictionary;
5  import java.util.Map;
6  import java.util.concurrent.ConcurrentHashMap;

8  import org.osgi.service.cm.ConfigurationException;
9  import org.osgi.service.cm.ManagedServiceFactory;

11 public class HttpPortListenerFactory implements ManagedServiceFactory {

13    private static final String PROP_PORT = "port";
14    private static final String PROP_DIR = "baseDir";

16    private final Map<String, HttpPortListener> map
17       = new ConcurrentHashMap<String, HttpPortListener>();

19    public String getName() {
20       return "HTTP Port Listener Factory";
21    }

23    public void updated(String pid, Dictionary dict)
24          throws ConfigurationException {
25       HttpPortListener newListener = null;
26       try {
27          if(dict != null) {
28             String portStr = (String) dict.get(PROP_PORT);
29             if(portStr == null)
30                throw new ConfigurationException(PROP_PORT,
31                      "Mandatory field");
32             int port = Integer.parseInt(portStr);

34             String dirStr = (String) dict.get(PROP_DIR);
35             if(dirStr == null)
36                throw new ConfigurationException(PROP_DIR,
37                      "Mandatory field");
38             File dir = new File(dirStr);
39             if(!dir.isDirectory())
40                throw new ConfigurationException(PROP_DIR,
41                      "Base directory does not exist: " + dir);

43             newListener = new HttpPortListener(port, dir);
44          }
45       } catch(NumberFormatException e) {
46          throw new ConfigurationException(PROP_PORT, "Invalid port");
47       } finally {
48          HttpPortListener oldListener = (newListener == null) ?
49                map.remove(pid) : map.put(pid, newListener);
50          if(oldListener != null) oldListener.stop();
51          if(newListener != null) newListener.start();
52       }
53    }

55    public void deleted(String pid) {
56       HttpPortListener oldListener = map.remove(pid);
57       if(oldListener != null) oldListener.stop();
58    }

60 }
```

```
osgi> start 5

osgi> STARTED listener, port=80, directory=./pages
```

The output on the last line comes from the `start` method of `HttpPortListener`. If we change the value of the `baseDir` property and save the file, then we see the following output:

```
Updating configuration from org.osgi.book.configadmin.HttpPortLis↩
tenerFactory-main.cfg
STOPPED listener, port=80, directory=./pages
STARTED listener, port=80, directory=/home/www/html
```

Suppose we create another file named `org.osgi.book.configadmin.HttpPortListenerFactory-secondary.cfg`:

```
STARTED listener, port=8080, directory=./MyWebPages
```

Finally, suppose we delete the file `org.osgi.book.configadmin.HttpPortListenerFactory-main.cfg`; the corresponding listener will be stopped but the "secondary" one will keep going:

```
STOPPED listener, port=80, directory=/home/www/html
```

9.2.6. A Common Mistake

When implementing the `ManagedServiceFactory` interface to create multiple configured objects, there is a very common misconception that the individual objects should implement `ManagedService` and be registered as services under that interface.

This is a very understandable error, since `ManagedService` is used for configuring objects after they are created so it seems to make sense that the objects created by a `ManagedServiceFactory` should implement `ManagedService` in order to continue receiving new configuration data after creation... and perhaps we should use the PID supplied by the factory for the PID of our `ManagedService`?

However this is wrong, and it is important to understand why. When using a `ManagedServiceFactory`, configuration changes to existing objects are always passed via the factory, which is solely responsible for reconfiguring the objects it previously created. Thus the name "factory" only describes half of what it is really responsible for!

Here is another way to put it: CM neither knows nor cares what kind of objects are created by a `ManagedServiceFactory`, so when one of those objects needs to be reconfigured, CM does not attempt to do so directly. Instead, it asks the factory that originally created the object to find and reconfigure it. This is why the factory needs to store any object it creates in a map.

We assume that a factory always knows how to reconfigure any object that it has itself created. In some cases the objects cannot be directly reconfigured — e.g., if they are immutable and take configuration only via constructor parameters, as in the example of the `HttpPortListener`s — these must be destroyed and recreated by the factory. In other cases there may be mutator methods on the objects that allow the factory to reconfigure them without destroying and recreating. These are implementation details of the objects and the factory; CM itself does not care.

What would happen if we ignored this warning and did register each object created from a factory under the `ManagedService` interface? In fact this is forbidden by the Configuration Admin specification because it would force two configurations to have the same PID. The CM implementation would ignore the `ManagedService` registration and log an error.

9.2.7. Multiple Configured Service Objects

On the other hand, it is quite valid and indeed common for the objects we create from a `ManagedServiceFactory` to be registered as some other kind of service.

Let's return to the database-backed mailbox example. Previously we created a single instance of this mailbox connected to a single database, but there may in fact be many databases that we could use as mailboxes. The user or an administrator could configure one mailbox service for each database, therefore the number of services is determined by the configuration. We will use a `ManagedServiceFactory` that both creates the mailboxes and registers them under the `Mailbox` service, as shown in Listings 9.9 and 9.10.

Notice the *very* strong similarity between this factory code and the code in Listing 9.6 for the singleton `ConfiguredDbMailboxManager`. Both classes manage instances of the `ConfiguredDbMailbox2` class; but the factory version stores them in a map rather than a reference field. Thus we arrive at essentially the same idiom for implementing both `MangedService` and `ManagedServiceFactory`[6]!

9.2.8. Configuration Binding

When a configuration object is created by FileInstall or some other management agent, it is usually created "unbound". This means it is available to be consumed by any bundle that publishes a configuration target (a `ManagedService` or `ManagedServiceFactory`) with the correct PID.

[6] In a sense, a `ConcurrentHashMap` is just a multi-entry `AtomicReference`. Or if you prefer, an `AtomicReference` is just a single-entry `ConcurrentHashMap`...

Listing 9.9 Activator for Multiple Configured Services

```
1  package org.osgi.book.configadmin;
2
3  import java.util.Properties;
4
5  import org.osgi.framework.BundleActivator;
6  import org.osgi.framework.BundleContext;
7  import org.osgi.framework.Constants;
8  import org.osgi.service.cm.ManagedServiceFactory;
9
10 public class ConfiguredDbMailboxActivator3 implements BundleActivator {
11    public void start(BundleContext context) throws Exception {
12       Properties props = new Properties();
13       props.put(Constants.SERVICE_PID,
14             ConfiguredDbMailboxFactory.class.getName());
15
16       context.registerService(ManagedServiceFactory.class.getName(),
17             new ConfiguredDbMailboxFactory(context), props);
18    }
19
20    public void stop(BundleContext context) throws Exception {
21    }
22 }
```

However, as soon as CM matches up the configuration with a consumer bundle and passes it to the `updated` method of a configuration target published by that bundle, the configuration becomes "bound", meaning it cannot be supplied to any other bundle, even one that publishes another configuration target with the same matching PID. Technically the configuration is bound to the *location* of the bundle that consumes it, i.e. the URL from which the bundle was installed, and it cannot subsequently be used by another bundle with a different location. The location is thus used as a kind of ID for the bundle. The configuration will remain bound until the consuming bundle is uninstalled — at that point it will revert to the unbound state and can again be supplied to any bundle offering a configuration target with the correct PID.

Sometimes configurations are created already in the bound state and never become unbound. However this is less common because unbound configurations are more generally useful. This topic will be discussed in more detail when we look at how bundles create and update configurations.

9.3. Describing Configuration Data

We have now seen essentially all of the ways for our application to *use* configuration data, at least with the lowest level of API available to us. We have also seen how to use a simple existing management agent to *provide* configuration data to the application.

However, there is a missing piece of the puzzle. In order to provide configu-

Listing 9.10 Factory for Multiple Configured Services

```
1  package org.osgi.book.configadmin;
2
3  import java.util.Dictionary;
4  import java.util.Map;
5  import java.util.Properties;
6  import java.util.concurrent.ConcurrentHashMap;
7
8  import org.osgi.book.reader.api.Mailbox;
9  import org.osgi.framework.BundleContext;
10 import org.osgi.framework.ServiceRegistration;
11 import org.osgi.service.cm.ConfigurationException;
12 import org.osgi.service.cm.ManagedServiceFactory;
13
14 public class ConfiguredDbMailboxFactory implements ManagedServiceFactory {
15
16    private static final String PROP_JDBC_URL = "jdbcUrl";
17    private static final String PROP_MBOX_NAME = "mboxName";
18
19    private final BundleContext context;
20    private final Map<String,ServiceRegistration> registrations
21       = new ConcurrentHashMap<String, ServiceRegistration>();
22
23    public ConfiguredDbMailboxFactory(BundleContext context) {
24       this.context = context;
25    }
26
27    public String getName() {
28       return "Database Mailbox Factory";
29    }
30
31    public void updated(String pid, Dictionary dict)
32             throws ConfigurationException {
33       ServiceRegistration newRegistration = null;
34       try {
35          if(dict != null) {
36             String jdbcUrl = (String) dict.get(PROP_JDBC_URL);
37             if(jdbcUrl == null)
38                throw new ConfigurationException(PROP_JDBC_URL,
39                                                 "Mandatory field");
40             String mboxName = (String) dict.get(PROP_MBOX_NAME);
41             if(mboxName == null)
42                throw new ConfigurationException(PROP_MBOX_NAME,
43                                                 "Mandatory field");
44             ConfiguredDbMailbox2 mailbox = new ConfiguredDbMailbox2(
45                                                            jdbcUrl);
46             Properties props = new Properties();
47             props.put(Mailbox.NAME_PROPERTY, mboxName);
48             newRegistration = context.registerService(
49                   Mailbox.class.getName(), mailbox, props);
50          }
51       } finally {
52          ServiceRegistration oldRegistration = (newRegistration == null)
53                ? registrations.remove(pid)
54                : registrations.put(pid, newRegistration);
55          if(oldRegistration != null) oldRegistration.unregister();
56       }
57    }
58
59    public void deleted(String pid) {
60       ServiceRegistration oldRegistration = registrations.remove(pid);
61       if(oldRegistration != null) oldRegistration.unregister();
62    }
63 }
```

ration data via FileInstall, we had to know the PIDs of all the configurable objects and the names and meanings of all the fields they expected to read from their configuration, along with knowledge of which fields were optional and what their default values were. How is a user or administrator supposed to know all of this?

We could write a thick manual to accompany our program, but this is not a very satisfactory answer. As all developers know, documentation easily gets out of date with respect to features of the software. Also it creates a huge burden on our users, who must thoroughly read a large list of PIDs and field names before they can make any change. This is not how software should be configured in the 21st century.

If we could make our configurable object self-describing, then it would be possible to offer configuration tools to the end user which are far more powerful and usable than a plain old text editor. Such a tool could tell the user:

- Which objects in the system are configurable.
- What fields are supported by each object, and what the meaning of those fields is.
- What the expected data type of each field is.
- What the range or list of expected values is.

Even better, if we could describe this information in a standard form then we would not need to build tools specifically for each application, but instead tools from specialist vendors could be used with multiple applications.

The OSGi Compendium offers just such a standard form for describing the configuration needs of objects — the Metatype Service Specification.

9.3.1. Metatype Concepts

The Metatype Service Specification defines a way to structure the metadata about configured objects in such a way that it can be used to automatically construct a reasonable user interface within a tool. It describes two basic concepts:

Object Class Definitions (OCDs) consist of a name, description, a set of Attribute Definitions, and optionally an icon.

Attribute Definitions (ADs) consist of a name, description, type, cardinality (i.e. how many values are permitted) and optionally a list of allowed values.

The Metatype Service is not in fact tied only to Configuration Admin, it can be used for publishing metadata about many kinds of artefact, but nevertheless it maps very neatly onto Configuration Admin concepts. An OCD can be mapped onto a `ManagedService` or `ManagedServiceFactory` object, and an AD can map onto a dictionary key passed to those objects. So when we create new configuration objects using CM, we should also create metadata for them using the Metatype Service.

There are two ways to do this. We can publish a service under the `MetaType-Provider` interface, which allows us to dynamically respond to metadata queries, or we can create an XML file in a well-known location inside our bundle.

Which way is best? Usually the file-based approach should be preferred, because except in rare circumstances the service publication approach results in repetitive boilerplate code. XML also has the advantage that it can be inspected by tools while our application is offline, whereas the service-based would allow us only to configure our application while it is running.

Whichever way is chosen to expose metadata, there is an API that can be used within the OSGi runtime to inspect all available objects and attributes. The `MetaTypeService` is published by a special bundle and it provides a convenient mechanism for looking up metadata in the currently running OSGi framework. However it cannot offer any help with inspecting metadata while the framework is offline, nor can it be used to query metadata from another instance of OSGi, e.g. over a network.

9.3.2. Creating a Metadata File

The metadata XML files must be created in the `OSGI-INF/metatype` directory of a bundle.

TODO

9.4. Building a Configuration Management Agent

Now we turn to the other audience for this chapter: those who wish to write management agents that will allow an OSGi application to be configured in a particular way, i.e with a specific format and storage mechanism for the configuration data.

First let's discuss the term "management agent". Recall that we applied this term to the FileInstall bundle that we worked with in the the previous section, but what does it really mean? The idea of a management agent in OSGi is in

fact very general, and the task of managing configuration is only one aspect. There are many other tasks that a management agent *could* do, and we will discuss these tasks in a later chapter, but not all management agents need to do everything. Therefore, they can vary enormously in size and complexity. FileInstall is a very small and simple example.

One area in which FileInstall is weak is its handling of data types in configuration data. Since it works entirely with properties files and does not look at metadata published by the Metatype Service, it cannot infer the type of data in each field; therefore it treats everything as a string. This is why in our `HttpPortListener` example it was necessary to parse the port number into an integer. Other file formats exist which offer more information about the type of the data they contain, for example XML would allow us to add an attribute to each field element that specified its type, as follows:

```
<configuration>
   <field name="baseDir" type="String" value="./pages"/>
   <field name="port" type="Integer" value="1234"/>
</configuration>
```

However XML may be considered too verbose. An interesting alternative is JSON (JavaScript Object Notation) which has a range of built-in primitive data types — strings, numbers and booleans — where the type of a field is indicated directly by the syntax. For example the same data could be written in JSON like this:

```
{
   "baseDir" : "./pages",
   "port"    : 1234
}
```

Here the double quotes around the value for the `baseDir` field clearly indicate that it is a string type, whereas the lack of quotes and numeric content for the `port` value indicate it is a number. If we built a management agent that understood JSON data, then it could supply numeric fields – such as the port number in this example – directly to the configuration targets as `Integer` objects rather than encoded in as strings.

So for our example, we will build exactly that. However we will avoid the fiddly task of polling the filesystem, and instead create some console commands that allow the user to load, update, view and delete configurations. Creating console commands was first discussed in Section **??**, you may wish to refer back to that section to refresh some of the details.

9.4.1. Listing and Viewing Configurations

First we will write a command for viewing existing configurations. Operations on the state of configuration must be done by calling methods on the `ConfigurationAdmin` service, which is published by the CM bundle. The simplest

operation is listing configurations, so we will first build a console command for viewing existing configurations and use it to look at the configurations that have been created by FileInstall.

As `ConfigurationAdmin` is a service, we need a `ServiceTracker` to access it. Listing 9.11 shows the bundle activator for our command bundle, which creates a tracker, passes it to a new command provider and registers the command provider.

Listing 9.11 Activator for the JSON Configuration Bundle

```
1  public class JsonConfigCommandsActivator implements BundleActivator {
2
3      private ServiceTracker cfgAdmTracker;
4      private ServiceRegistration reg;
5
6      public void start(BundleContext context) throws Exception {
7          cfgAdmTracker = new ServiceTracker(context,
8                  ConfigurationAdmin.class.getName(), null);
9          cfgAdmTracker.open();
10
11         reg = context.registerService(CommandProvider.class.getName(),
12                 new JsonConfigCommands(cfgAdmTracker), null);
13     }
14
15     public void stop(BundleContext context) throws Exception {
16         reg.unregister();
17         cfgAdmTracker.close();
18     }
19 }
```

The command provider itself will be relatively large when it is completed, so we will attack it piecemeal. The constructor and `getHelp` method are uninteresting and so not shown here. Instead we will look first at two utility methods to output the contents of a configuration record in either summary or detailed form. We will use the summary form for printing listings of multiple configurations, and the detailed form for printing single configurations.

Now we will write the `_listConfigs` method, as shown in Listing 9.13. This command takes an optional argument that can be used to narrow the list of configurations to a specific PID; the argument can contain wildcards, or omitted entirely in which case all configurations will be listed. It works by calling the `listConfigurations` method on the `ConfigurationAdmin` service, which takes a filter string in the usual format, or `null`.

Our command can now be used with the existing configurations created by FileInstall in the previous section:

```
osgi> listConfigs
PID=org.osgi.book.configadmin.HttpPortListenerFactory-1245012586509-0↩
,factoryPID=org.osgi.book.configadmin.HttpPortListenerFactory

osgi> listConfigs *HttpPortListener*
PID=org.osgi.book.configadmin.HttpPortListenerFactory-1245012586509-0
Factory PID=org.osgi.book.configadmin.HttpPortListenerFactory
```

Listing 9.12 Utility Methods for Viewing Configurations

```
1   void showConfigSummary(Configuration config, StringBuffer buffer) {
2      buffer.append("PID=").append(config.getPid());
3      String factoryPid = config.getFactoryPid();
4      if(factoryPid != null)
5         buffer.append(", factoryPID=").append(factoryPid);
6   }

8   void showConfigDetailed(Configuration config, StringBuffer buffer) {
9      buffer.append("PID=").append(config.getPid()).append('\n');

11     String factoryPid = config.getFactoryPid();
12     if(factoryPid != null)
13        buffer.append("Factory PID=").append(factoryPid).append('\n');

15     String location = config.getBundleLocation();
16     location = (location != null) ? location : "<unbound>";
17     buffer.append("Bundle Location:").append(location).append('\n');

19     buffer.append("Contents:\n");
20     Dictionary dict = config.getProperties();
21     if(dict != null) {
22        Enumeration keys = dict.keys();
23        while(keys.hasMoreElements()) {
24           Object key = keys.nextElement();
25           buffer.append('\t').append(key).append('=')
26              .append(dict.get(key)).append('\n');
27        }
28     }
29  }
```

Listing 9.13 Listing Configurations

```
1   public void _listConfigs(CommandInterpreter ci) throws IOException,
2         InvalidSyntaxException {
3      ConfigurationAdmin cfgAdmin =
4         (ConfigurationAdmin) cmTracker.getService();
5      if(cfgAdmin == null) {
6         ci.println("ConfigurationAdmin service not available");
7         return;
8      }

10     String filter = null;
11     String pid = ci.nextArgument();
12     if(pid != null)
13        filter = "(service.pid=" + pid + ")";
14     Configuration[] configs = cfgAdmin.listConfigurations(filter);
15     if(configs == null || configs.length == 0) {
16        ci.println("No configurations found");
17     } else if(configs.length == 1) {
18        StringBuffer buffer = new StringBuffer();
19        showConfigDetailed(configs[0], buffer);
20        ci.println(buffer);
21     } else {
22        for (int i = 0; i < configs.length; i++) {
23           StringBuffer buffer = new StringBuffer();
24           showConfigSummary(configs[i], buffer);
25           ci.println(buffer);
26        }
27     }
28  }
```

```
Bundle Location:file:http_listeners.jar
   baseDir=./load
   felix.fileinstall.filename=org.osgi.book.configadmin.HttpPortListe↩
nerFactory-1.cfg
   port=8080
   service.factoryPid=org.osgi.book.configadmin.HttpPortListenerFactory
   service.pid=org.osgi.book.configadmin.HttpPortListenerFactory-1245↩
012586509-0
```

9.4.2. Creating and Updating Configurations

Now we will add a command to the same command provider class for creating and updating configurations. Somewhat surprisingly, the principal method on `ConfigurationAdmin` that we need to use is named `getConfiguration`! The reason for this is the need to avoid race conditions: if there were a `createConfiguration` method and two clients were to call it at around the same time with the same PID, then there is a risk that we would get two configurations with the same PID, which is not allowed. So `getConfiguration` works by returning an existing `Configuration` object if one exists with the specified PID, or creating a new one if not.

On the other hand, creating a configuration for a *factory* does not suffer from race conditions: it is permitted to have multiple configurations for the same factory PID, and in this case the individual PIDs will be generated by the CM implementation. Therefore we have a `createFactoryConfiguration` method which takes the factory PID and always creates a new `Configuration` object.

Before writing this method we will need to consider how to map JSON structures to CM configurations. JSON allows for deeply nested structures which are more complex than CM can represent, so we must limit the complexity of the files loaded by our command. If a file contains nested objects or arrays then we will reject it by throwing a `ConfigurationException`. On the other hand, we can handle an array of top-level objects if we assume that they map to individual configurations for a factory. Therefore we employ the following simple rule: if the input file contains a single top-level object then we assume it is a singleton configuration, and if the input file contains an array at the top-level then we assume the members of that array are factory configurations. Following this assumption means we don't have to write separate methods for creating singleton and factory configurations.

To parse the JSON data we can use one of the many open source parser libraries available, such as Jackson[**?**]. The code to interface with the parser is omitted here because it is verbose and not directly relevant to OSGi, but let us assume that we have a method available named `parseJson` which takes a `java.io.File` as input and returns an array of `Dictionary` objects.

Given this method, we can write our `_loadConfig` command as shown in Listing 9.14.

Listing 9.14 Loading JSON Data

```
1   public void _loadConfig(CommandInterpreter ci)
2       throws JsonParseException, IOException, ConfigurationException {
3       String fileName = ci.nextArgument();
4       String pid = ci.nextArgument();
5       if(fileName == null || pid == null) {
6           ci.println("Usage: loadConfig <file> <pid> - load JSON-format" +
7                   " configuration data from file into the specified PID");
8           return;
9       }
11      ConfigurationAdmin cfgAdmin = (ConfigurationAdmin) cmTracker.getService();
12      if(cfgAdmin == null) {
13          ci.println("ConfigurationAdmin service not available");
14          return;
15      }
17      Dictionary[] dicts = parseJson(new File(fileName));
18      if(dicts.length == 1) {
19          System.out.println("--> getting configuration");
20          Configuration config = cfgAdmin.getConfiguration(pid, null);
21          System.out.println("--> updating configuration");
22          config.update(dicts[0]);
23      } else {
24          for (Dictionary dict : dicts) {
25              System.out.println("--> creating factory configuration");
26              Configuration config = cfgAdmin.createFactoryConfiguration(pid, null
27              System.out.println("--> updating configuration");
```

Creating or updating a configuration record is a two-step procedure: first we obtain a `Configuration` object via `getConfiguration` or `createFactoryConfiguration`; then we call its `updated` method with the dictionary of configuration properties. It is on the second step that the `ManagedService` or `ManagedServiceFactory` — if any exist with a matching PID — are called by the CM runtime.

9.4.3. Creating Bound and Unbound Configurations

As shown in Listing 9.14, the methods used in CM to create configurations are `getConfiguration` (for singleton configurations) and `createFactoryConfiguration`. Both of these are shown taking a PID for the first argument and `null` for the second.

What is the `null` second argument? It is the bundle location to which the new configuration should be bound. As described in Section 9.2.8, configurations can be bound to specific bundles based on location, but Management Agents should normally create unbound configurations that can be consumed by any bundle offering a configuration target with the correct PID. To create an unbound configuration we simply pass a location of `null`.

Both of these methods also have single-argument versions that only take a PID as input. These single-argument methods are used to create configuration ob-

jects bound to the location of the calling bundle; in other words, configuration objects that can only be consumed by the same bundle that created them. However it is not so useful for a bundle to configure itself in this way. The main purpose of the CM spec is to allow the task of loading and managing configuration to be decoupled from components which need to receive configuration, but if a bundle loads its own configurations then these tasks are not effectively decoupled.

The reason for having these two versions is to allow administrators to use security to control which bundles are allowed to act as Management Agents. The dual-argument versions of `getConfiguration` and `createFactoryConfiguration` are intended for use by Management Agents, and Java security can be used to control which bundles are permitted to call these methods. When security is enabled, only bundles designated as Management Agents are able to supply configuration to other bundles. On the other hand, any bundle should be allowed to use CM to configure *itself* (even if we don't consider it useful), so the single-argument methods do not require special permission in order to call them. However, none of the preceding discussion is relevant if you are not using Java security to lock down your application, and it is quite rare to do so. Just remember that for a Management Agent we need to call the dual-argument methods with `null` as the second argument.

9.5. Creating a Simple Configuration Entry

We saw in the previous section how to receive configuration data. Now we will look at how to load configuration data and create configuration entries. This is done using the `ConfigurationAdmin` interface. Configuration Admin is implemented as a service, so we will need to use a `ServiceTracker` to get hold of the implementation of the service.

Somewhat counter-intuitively, the method we call to create a configuration is called `getConfiguration`! This method either returns an existing configuration with the PID that we specify, or it creates a new configuration with that PID. There is no "`createConfiguration`" method for a very good reason: it guards against race conditions. If there was such a "create" method, then two threads could call it simultaneously and create two configurations with the same PID, which is not allowed.

On calling `getConfiguration` we are returned a `Configuration` object. We can now set the properties by calling `update` on that object and passing a `Dictionary`. It is only then that the `updated` method of any corresponding `ManagedService` is called. The properties are also persisted at that point, so that they can be given to instances of `ManagedService` that may appear later.

Listing 9.15 shows an example of a bundle activator that loads configuration

data from a file, which is assumed to be in the normal Java property file format. There is a lot wrong with this simplistic activator: it assumes that it will be started after the CM bundle, and it uses hard-coded values for the PID and file name. It is intended only as a minimal example to test the `ManagedService` implementations above. In Section **??** we will build a more realistic bundle that manages the link between CM and property files. But before doing that, we switch back to looking at the consumer aspect of CM.

Listing 9.15 Loading Configuration Data from a File

```
1  package org.osgi.book.configadmin;

3  import java.io.FileInputStream;
4  import java.io.IOException;
5  import java.util.Properties;

7  import org.osgi.framework.BundleActivator;
8  import org.osgi.framework.BundleContext;
9  import org.osgi.framework.ServiceReference;
10 import org.osgi.service.cm.Configuration;
11 import org.osgi.service.cm.ConfigurationAdmin;

13 public class SimpleFileConfigActivator implements BundleActivator {

15    private static final String PID =
16       "org.osgi.book.configadmin.ConfiguredDbMailbox";
17    private static final String FILE = "config.properties";

19    public void start(BundleContext context) throws Exception {
20       // Load the properties
21       Properties props = new Properties();
22       FileInputStream in = new FileInputStream(FILE);
23       props.load(in);

25       // Set the configuration
26       ServiceReference ref = context.getServiceReference(
27                                 ConfigurationAdmin.class.getName());
28       if(ref != null) {
29          ConfigurationAdmin cm =
30             (ConfigurationAdmin) context.getService(ref);
31          if(cm != null) {
32             Configuration config = cm.getConfiguration(PID, null);
33             config.update(props);
34          }
35          context.ungetService(ref);
36       }
37    }

39    public void stop(BundleContext context) throws Exception {
40    }
41 }
```

Part II.

Component Oriented Development

10. Introduction to Component Oriented Development

In the introduction to Part I, we discussed the use of modularity techniques in the general engineering disciplines to manage complexity, and drew parallels with the need to modularise software.

Another concept that is very important in traditional engineering is the idea of the *component*.

In the physical world everybody knows what components are and why they are useful. They range from the very simple, such as a standard screw, to the more complex such as an LCD flat panel display. Going back to our example of the Boeing 747, many of its parts are components and some of those components were not necessarily designed to be used in an aircraft: they could equally be used in the construction of a car or even a house. Others will be more specialised but still not limited to the 747, and could be used in other aircraft. Still others *are* specific to the 747, but many copies of the same kind of component will exist throughout a single aircraft.

This is the point of components: *reuse*. If every single one of the six million parts in a 747 had to be individually designed and machined from scratch, then Boeing would probably still be building the first 747 today. By copying the design of components many times we can shorten the time to market for our products.

Some products can even be built entirely out of components. Here the 747 example fails us because aircraft must still be largely custom-designed; a better example would be a house. So many parts of modern houses are standarised that it is quite possible to build one entirely out of off-the-shelf components. The same could be said of a desktop computer. The resulting products can be just as high quality as their custom-designed equivalents but at a fraction of the cost.

In software we would like to use components also, and at least in theory software components have a huge advantage over physical components like screws and doors, because the latter still have a unit cost whereas software components can be reproduced infinitely many times for free. Unfortunately many of the useful properties of true components have been difficult to reproduce in software, as we will soon see.

10.1. What is a Software Component?

The software development world has for more than two decades been dominated by the paradigm known as Object-Oriented Programming, or OOP. Every mainstream programming language supports this paradigm and it is universally taught in Computer Science courses around the world.

...

But first, let's imagine what could be possible if we had true software components. Would this reduce the task of software development to assembling pre-written components? Probably not. Most software products still require significant levels of custom development: they are in this respect closer to aeroplanes than houses. But there are still great potential gains from the use of components in software.

The differences

11. Declarative Services

The OSGi Services model is the lynchpin of OSGi's component model. However as we saw in previous chapters, working with services using the basic APIs available in the OSGi core specification can be fraught with difficulties. It is difficult firstly to write *correct* code – i.e. code that handles concurrent events properly, does not hold on to references longer than it should, avoids deadlocks and remains responsive — and it is even more difficult to consistently write such code as we use services over and over again.

This is not really a problem with the services model itself, it is simply a question of finding the right abstraction level. Dealing with dynamic services that come and go in multiple threads is not an easy task, so we should rely on abstractions that handle the low-level details for is. The `ServiceTracker` class that we have used extensively is just such an abstraction, and as abstractions go it is a very good one. But it is still a very low-level one that requires us to write a lot of boilerplate "glue" code.

11.1. The Goal: Declarative Living

One of the reasons why low-level OSGi code is hard to test is that it mixes up the desired functionality of the code – its "business logic" – with the boilerplate that is necessarily present merely to make the code work. Thus the functionality cannot be separated from the glue and cannot be tested in isolation.

This is an odd and uncomfortable situation for a technology that aims to *improve* the modularity and separation of concerns in our code!

Another problem is that it is difficult to understand the purpose of the glue code, except by carefully reading and analysing it. For example, suppose wish to make service A depend on service B, such that if B is not present then A will not be available either. Although we can achieve this easily with `ServiceTracker`, our simple *intention*, i.e. A-depends-on-B, is hidden behind the mechanics of tracking B, overriding `addingService`, passing the B instance to A, etc.

Therefore we require a solution that allows us to first isolate the real functionality from the OSGi-specific glue code. Second, we would like to replace

imperative glue code with simple declarations about our intent.

11.2. Introduction

The Declarative Services specification ("DS"), which is a part of the OSGi Services Compendium, has exactly these goals. It is an abstraction that allows us to write straightforward Java objects that do not even refer to the OSGi APIs, and then declare the linkages between them using a simple XML-based structure.

DS cannot support absolutely every use-case supported by the low-level APIs, but that is the nature of an abstraction: by sacrificing a small number of unusual use-cases, the majority that remain supported are made *much* easier. Experience suggests that over 90% of cases are supportable by DS, and for the rest we can simply drop down to the APIs.

11.2.1. Summary of Declarative Services Features

What does the DS specification offer in terms of features? The following list is just a summary; don't worry if some are unclear at this stage, they will all be explained in detail later on.

Creation and Management of Component Instances

At its simplest, DS can be used to create and manage declared component instances, which are simply Java objects.

Providing Services

DS components may optionally be published to the service registry under one or more service interfaces. This occurs under the full control of DS's runtime, so that components may be unpublished if they become disabled for any reason, and then published again when they are re-enabled.

An additional and very compelling feature of DS when it is used to publish services is its ability to defer creation of the component until its service is actually *used* by a client.

Referring to Services

Components often need to make use of other components, and in DS this is done by declaring references to the services offered by those components. In fact DS components can refer to any OSGi service, not just those published by other DS components, and each component can refer to many different types of service.

Service references can be customized to be optional or mandatory, unary or multiple. Each reference is independently handled, so a component can have a mandatory unary reference on one type of service along with an optional multiple reference to another type.

Dynamic Lifecycle

DS components can be optionally declared with lifecycle methods, which allow a component to have an "active" state. They may use this state to perform some kind of action within the system such as running a thread or thread pool, opening a socket, or interacting with a user.

Configuration

DS components can be supplied with configuration data in a number of ways. One of these ways is via the Configuration Admin service, and DS interoperates with Configuration Admin (when it is available) to ease the development of configurable components. We never need to implement either the `ManagedService` or `ManagedServiceFactory` interfaces.

POJOs

DS components can be implemented entirely as Plain Old Java Objects, without any reference to the OSGi APIs. This enhances testability and makes them usable in other non-OSGi containers such as JEE or the Spring Framework.

11.2.2. A Note on Terminology and Versions

A note on terminology will be helpful if you have read about DS elsewhere. DS is a *specification*, and a *model* for wiring together components dynamically via services. It is implemented as an extender bundle, and this runtime bundle is referred to as the Service Component Runtime ("SCR"). Unfortunately the two terms DS and SCR are sometimes incorrectly used as interchangeable

terms for the same thing, so this is something to bear in mind when reading other texts on the subject.

This book principally describes version 1.1 of the DS specification, which is part of the OSGi Release 4.2. Version 1.1 has a number of important improvements over version 1.0, however it is still relatively new so it is important to check that the SCR implementation we use is compliant with 1.1.

11.3. Declaring a Minimal Component

Let us first look at a minimal DS example. Listing 11.1 shows the implementation code for a component, which as we can see does not implement any special interfaces or call any OSGi API methods; indeed, none of the `org.osgi.*` packages are even imported, making this is a true POJO, albeit a useless one! We have added a constructor that prints a message, so that we can see when the object is created, but real components would not typically do this.

Listing 11.1 Java Code for the Minimal DS Component

```
1 package org.osgi.book.ds.minimal;

3 public class HelloComponent {
4    public HelloComponent() {
5       System.out.println("HelloComponent created");
6    }
7 }
```

DS uses an XML file to declare components. Listing 11.2 shows the declaration corresponding to our `HelloComponent` class. Note first the declaration of an XML namespace, using the prefix `scr`. We must include this namespace in order for SCR to treat our component using the new DS 1.1 features. It is also useful to include as it allows XML editors to associate the document with a schema and provide help generating the correct structure for the document.

Listing 11.2 Minimal DS Component Declaration

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <!-- "minimal.xml" -->
3 <scr:component xmlns:scr="http://www.osgi.org/xmlns/scr/v1.1.0">
4   <implementation class="org.osgi.book.ds.minimal.HelloComponent"/>
5 </scr:component>
```

What does the declaration in Listing 11.2 actually say? It simply asks the runtime to create a new component as an instance of the `HelloComponent` class. The class we specify in the `class` attribute of the `implementation` element must be visible to the bundle in which we declare the component,

so we can either include it directly in the bundle or import it from another bundle in the usual way. The class does *not* need to implement any particular interfaces but it must have at least an accessible zero-argument constructor.

Our minimal example is complete, but there is a final step before we can build the bundle. Each bundle may declare any number of components, and the corresponding XML files can be stored anywhere inside the bundle. Therefore we need to list all of the declaration files somewhere so that SCR knows about them; of course, this is done using a manifest header. SCR looks for a header named `Service-Component` containing a list of paths to the XML files. We can simply add this header to our `bnd` descriptor as shown in Listing 11.3 because `bnd` will copy it straight through to the generated manifest. We also have to tell `bnd` to include these files in the bundle, which we do with the `Include-Resource` instruction[1].

Listing 11.3 Bnd Descriptor for the Minimal DS Component

```
# minimal_ds.bnd
Private-Package: org.osgi.book.ds.minimal
Service-Component: minimal.xml
Include-Resource: minimal.xml
```

11.4. Running the Example

To run this example we need to obtain an implementation of SCR. Equinox supplies this as a bundle named `org.eclipse.equinox.ds`, but it has a dependency on another bundle `org.eclipse.equinox.util`. Both of these bundles are included with the main Equinox SDK archive or downloadable separately from the Equinox download site[2]. We also need the OSGi compendium bundle, but we assume that this is already installed.

Let's install the minimal example bundle and the SCR bundles at that same time:

```
osgi> install file:minimal_ds.jar
Bundle id is 4

osgi> install file:org.eclipse.equinox.ds_1.1.1.R35x_v20090806.jar
Bundle id is 5

osgi> install file:org.eclipse.equinox.util_1.0.100.v20090520-1800.jar
Bundle id is 6

osgi>
```

[1] It is unfortunate that we must list the same files twice. In fact `bnd` offers a less cumbersome way of working with DS declarations, which we will see later. For now we stick to this approach because it makes the underlying mechanics of using SCR clearer.

[2] Also if you are using Eclipse 3.5 or above as your IDE then you should find a copy of these bundles in your ECLIPSE/plugins directory.

Now let's try starting both our bundle and the `org.eclipse.equinox.ds` bundle. The `org.eclipse.equinox.util` bundle does not need to be started, it is just a library.

```
osgi> start 4

osgi> start 5
HelloComponent created

osgi>
```

Note that nothing happens until we have started *both* the SCR bundle and our own bundle. The SCR bundle need to be active for it to do anything, and our bundle needs to be active because SCR ignores bundles in any other state.

The Equinox SCR implementation provides a pair of commands for inspecting the state of our DS components. These commands are `list`, which gives us a list of all components from all bundles, and `component` which gives us detailed information about a specific component. Both commands have shorter aliases: `ls` and `comp` respectively. Let's try using the `list` command now:

```
osgi> ls
All Components:
ID  State      Component Name                          Located in bundle
1   Satisfied  org.osgi...minimal.HelloComponent       minimal_ds(bid=4)
```

This shows there is just one component and its assigned ID is 1 (the component name field is elided to fit on the page). Try typing the command `component 1` now – it will print extensive information about the component, though most of it will not yet make any sense.

11.5. Providing a Service

The minimal example above is, of course, utterly useless! There are *much* easier ways to merely instantiate a class So we will expand the example by publishing a component as a service. This requires us to define a new service interface, so in this and the next few sections we will work with a very basic logging service interface. Although OSGi does already define a logging service, ours will be much more simplistic. The `Log` interface is shown in Listing 11.4 – this listing also shows the `bnd` descriptor used to wrap the interface in an API bundle named `logapi` that will be used by both the producers and consumers of the API.

Now we write a bundle that provides an implementation of the log API. Again keeping things as simple as possible, the `ConsoleLog` class in Listing 11.5 implements the `Log` interface and writes messages to the standard output or standard error streams, depending on the severity level of the message. As in the `HelloComponent` example, we add a constructor that prints a message when the object is created, just so that we can see when the object is created.

Listing 11.4 A Simplistic Log Service Interface and Bnd descriptor.

```
1  package org.osgi.book.logapi;
2
3  public interface Log {
4
5     public static final int DEBUG   = 0;
6     public static final int INFO    = 1;
7     public static final int WARNING = 2;
8     public static final int ERROR   = 3;
9
10    void log(int level, String message, Throwable t);
11 }
```

```
# logapi.bnd
Export-Package: org.osgi.book.logapi
```

Listing 11.5 The Console Log Class

```
1  package org.osgi.book.ds.log.console;
2
3  import java.io.PrintStream;
4
5  import org.osgi.book.logapi.Log;
6
7  public class ConsoleLog implements Log {
8
9     public ConsoleLog() {
10       System.out.println("ConsoleLog created");
11    }
12
13    public void log(int level, String message, Throwable t) {
14       PrintStream dest;
15       if(level > Log.WARNING) {
16          dest = System.err;
17       } else {
18          dest = System.out;
19       }
20
21       dest.println("LOG: " + message);
22       if(t != null) t.printStackTrace(dest);
23    }
24 }
```

This implementation class is no different from what we would have written in previous chapters. However now, instead of creating and publishing it as a service by writing a bundle activator, we will ask SCR to do the work. Listing 11.6 shows the XML declaration file and `bnd` descriptor.

Listing 11.6 DS Declaration & Bnd Descriptor for the Console Logger

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- "consolelog.xml" -->
<scr:component xmlns:scr="http://www.osgi.org/xmlns/scr/v1.1.0" configuration-po
    <implementation class="org.osgi.book.ds.log.console.ConsoleLog"/>
    <service>
        <provide interface="org.osgi.book.logapi.Log"/>
    </service>
</scr:component>
```

```
# consolelog_ds.bnd
Private-Package: org.osgi.book.ds.log.console
Service-Component: consolelog.xml
Include-Resource: consolelog.xml
```

Now we will run the bundle, assuming that the SCR bundle is still active from the previous section:

```
osgi> install file:consolelog_ds.jar
Bundle id is 7

osgi> install file:logapi.jar
Bundle id = 8

osgi> start 7

osgi> services
...
{org.osgi.book.logapi.Log}={component.name=
     org.osgi.book.ds.log.console.ConsoleLog, component.id=1,
     service.id=31}
  Registered by bundle: consolelog_ds_0.0.0 [7]
  No bundles using service.
```

From the output of the `services` command we can see that the service has been registered as we expected under the `Log` interface. But the really interesting thing is what we *don't* see: the message "ConsoleLog created" which should have been printed when the constructor of the `ConsoleLog` class was called. Since this message was not printed, the constructor cannot have been called, and therefore the object must not have been instantiated. Yet we can see a service registered in the service registry under the expected interface name. So what is actually going on?

11.5.1. Lazy Service Creation

The explanation for the behaviour seen in the previous example is a useful feature of Declarative Services called "delayed services". All services created

by SCR are delayed by default, which simply means that SCR registers the service with the service registry as soon as our bundle becomes active, but it does not create an instance of our class until the service is actually *used* by a client bundle. In other words the service object is created lazily or "on demand".

This largely fixes a weakness in the low-level approach to OSGi services: the pro-active creation of unneeded services. With the conventional approach to services expounded in Chapter 4, we may end up creating a large number of services that are never used by any consumer. This happens because we want the producers and consumers of services to be decoupled from each other, so that the producers know nothing about the consumers and *vice versa*. We certainly succeed at this, and perhaps a little too well because the producers do not know if any consumers even exist! Therefore the producers have to speculatively create services just in case some consumer needs them, which can be wasteful.

As an analogy, imagine a fast-food restaurant with a wide range of menu items, all kept hot and ready to eat at any time. Such items have a limited shelf life: if no customers order them then they must be thrown away, and the time and ingredients used to make them are wasted.

How bad is this problem? Well, services are just Java objects which usually have a very low *marginal* cost of production[3]. However if we just create one instance, then simply loading the Java class becomes a significant part of the cost. Unfortunately in OSGi we also have to create a class loader for the bundle, and although class loaders are not very expensive, neither are they free. So if we activate a bundle merely in order to instantiate a single object and register it as a service – one that nobody even uses – then it will have is a small but measurable impact on memory usage and performance. Scale this up to hundreds of bundles and we see the problem may become acute.

DS's delayed services feature is therefore very important, since it is able to register the service without actually creating the service object[4]. Furthermore, because the SCR lives in a separate extender bundle, it is not even necessary for the framework to create class loader for our bundle until the last moment. Therefore the registration of any number of services becomes essentially free.

There is one condition: we must not have a `BundleActivator` in our bundle. Remember that SCR only looks at bundles in the ACTIVE state, so we must start our bundle; however if the bundle has an activator then starting it will require creation of a class loader so that the activator class can be loaded. So to get the full benefit of lazy service creation, we must avoid declaring an

[3] The marginal cost of producing something is the cost of producing one more, given that we have already produced many.

[4] There is no magic involved in this, nor does it involve the use of expensive proxy objects. SCR simply registers a `ServiceFactory` as described in Section 4.11 on page 102

activator. In fact this is not so much of a problem, since bundles using DS generally do not need activators, as we will see soon.

Of course, the consumers of the services need not worry about any of the above. They simply access the service in the normal way and will be oblivious to the fact that the service object is created lazily.

11.5.2. Forcing Immediate Service Creation

Delayed services are the default in DS, but we can override the default and request that our service objects be created immediately by setting the `immediate` attribute to `true` at the top level of our XML declaration, as shown in Listing 11.7.

Listing 11.7 Declaration for the Console Logger as an "Immediate" Service

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <!-- "consolelog_immediate.xml" -->
3 <scr:component xmlns:scr="http://www.osgi.org/xmlns/scr/v1.1.0"
4         immediate="true">
5     <implementation class="org.osgi.book.ds.log.console.ConsoleLog"/>
6     <service>
7         <provide interface="org.osgi.book.logapi.Log"/>
8     </service>
9     <property name="service.ranking" type="Integer" value="10"/>
10 </scr:component>
```

If we wrap this in a bundle and install and start it, then we should see the expected message to appear on the console immediately:

```
osgi> install file:consolelog_ds_immediate.jar
Bundle id is 9

osgi> start 9
ConsoleLog created
```

Delayed creation can *only* happen for components that are services. In our first minimal example of a DS component, the object was instantiated immediately because the `immediate` attribute is implicitly `true` for non-services. If a component is not a service then there is no way to receive the signal that the component is "needed", so SCR must create it straight away.

11.5.3. Providing Service Properties

When publishing a service we often need to provide additional metadata in the form of properties, and we saw how to do with with the low-level publishing API back in Section 4.5. Clearly we will need a way to do the same in DS, and we would expect it to be done declaratively. Listing 11.8 shows how – in this

example we add a property named `level` with the value `DEBUG`, to indicate the minimum level of messages printed by this log instance.

Listing 11.8 Adding a Simple Service Property

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- "consolelog.xml" -->
<scr:component xmlns:scr="http://www.osgi.org/xmlns/scr/v1.1.0">
    <implementation class="org.osgi.book.ds.log.console.ConsoleLog"/>
    <property name="level" value="DEBUG"/>
    <service>
        <provide interface="org.osgi.book.logapi.Log"/>
    </service>
</scr:component>
```

The `property` element can be repeated many times to set additional properties. The type of the property is assumed to be `String` but we can override that by specifying the `type` attribute, which must take one of the following types: `String`, `Long`, `Double`, `Float`, `Integer`, `Byte`, `Character`, `Boolean` or `Short`. We can also specify arrays of these data types using a slightly odd syntax in which we omit the `value` attribute and place the values in separate lines in the text content of the XML element.

We can ask also SCR to load properties from a file located inside the bundle. This is done using a different element named `properties` with an `entry` attribute pointing to the file, which must be in the standard Java properties file format. All of the values will be implicitly strings, and there is no support for arrays. This feature is useful for adding a set of shared "header" properties to many components in our bundle.

Listing 11.9 Additional Service Properties

```
    <property name="level" value="DEBUG"/>
    <properties entry="OSGI-INF/vendor.properties"/>
    <property name="languages">
        en_US
        en_GB
        de
    </property>
    <property name="service.ranking" type="Integer" value="10"/>
```

Listing 11.9 shows the use of all these options. We can interleave `property` elements with `properties` elements. The order is significant because in the event that there are name clashes the later setting wins; so we can place default settings from a properties file at the top of the descriptor, and override those with individual `property` elements as needed.

11.6. References to Services

We have seen how DS replaces the "glue code" required for publishing a service, now we look at the task of consuming services. Naturally this is also done declaratively.

In our first example we will bring together the logging service from this chapter and the `MailboxListener` service from Chapter 7 to build a `LogMailboxListener`. This class listens for changes in any mailbox – e.g. a new message arriving – and prints to the log when such a change occurs. The code for the class is shown in Listing 11.10.

Listing 11.10 A Logging Mailbox Listener

```
1  package org.osgi.book.ds.mboxlistener;
2
3  import org.osgi.book.logapi.Log;
4  import org.osgi.book.reader.api.*;
5
6  public class LogMailboxListener implements MailboxListener {
7
8     private Log log;
9
10    public LogMailboxListener() {
11       System.out.println("LogMailboxListener created");
12    }
13
14    public void setLog(Log log) {
15       this.log = log;
16    }
17
18    // Implements MailboxListener.messagesArrived(...)
19    public void messagesArrived(String mboxName, Mailbox mbox,
20                                long[] ids) {
21       log.log(Log.INFO, ids.length + " message(s) arrived in mailbox "
22             + mboxName, null);
23    }
24 }
```

The first thing to notice about this class is that, just like the previous two DS-based examples, it is a POJO. As before we have added a constructor that prints to the console when an object is created, which helps us to see what SCR is doing, but the class has no dependency on OSGi and closely follows the conventional JavaBeans programming model. The log implementation is supplied through a setter method, meaning this class can be easily unit-tested outside of OSGi, or used in a non-OSGi runtime such as the Spring Framework or Java Enterprise Edition (JEE).

When we are running within OSGi we expect SCR to call the setter method to provide the log service. For this to happen we must declare a new element in the XML descriptor named `reference`, which informs SCR that our component *refers* to a service. Additionally we would like to publish our object as

a service itself under the `MailboxListener` interface, so we include a `service` element also. The complete XML declaration is shown in Listing 11.11.

Listing 11.11 DS Component Declaration for the Logging Mailbox Listener

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <!-- "logmailboxlistener.xml" -->
3 <scr:component xmlns:scr="http://www.osgi.org/xmlns/scr/v1.1.0">
4     <implementation
5       class="org.osgi.book.ds.mboxlistener.LogMailboxListener"/>
6
7     <service>
8       <provide interface="org.osgi.book.reader.api.MailboxListener"/>
9     </service>
10
11    <reference interface="org.osgi.book.logapi.Log" bind="setLog"/>
12 </scr:component>
```

We have seen the `implemention` and `service` elements of this XML file before, so the only new thing is the `reference` element, which tells SCR that our component "refers to" the log service. The `interface` attribute is also mandatory as it indicates which service type we wish to use in the component. We also need to supply the `bind` attribute to indicate the name of the setter method [5].

Consider for a moment how much effort we have saved here. To do this the low-level way, we would have had to write a `ServiceTracker` sub-class to track the `Log` service, create an instance of the `LogMailboxListener`, provide it the log instance, register the `LogMailboxListener` as a service, and finally clean up everything when the `Log` service goes away. All this has been replaced with just a few lines of purely declarative XML. But before we celebrate, let's check that it actually works. Listing 11.12 shows the `bnd` file we need.

Listing 11.12 Bnd Descriptor for the DS-Based Logging Mailbox Listener

```
# logmailboxlistener.bnd
Private-Package: org.osgi.book.ds.mboxlistener
Service-Component: logmailboxlistener.xml
Include-Resource: logmailboxlistener.xml
```

Now we install and start the bundle. We also need the `mailbox_api` bundle from Chapter 3, if it is not already present:

```
osgi> install file:logmailboxlistener.jar
Bundle id is 10

osgi> install file:mailbox_api.jar
Bundle id is 11
```

[5] Names of the form `setLog` to set the "log" field are conventional, to allow for compatibility with JavaBeans, but DS does not expect or enforce any particular naming. Also the `bind` attribute is not strictly mandatory since there is another way to obtain the service, but in this example we do need to include it.

```
osgi> start 10

osgi> services
...
{org.osgi.book.reader.api.MailboxListener}={component.name=
     org.osgi.book.ds.logmailboxlistener.LogMailboxListener,
     component.id=3, service.id=33}
  Registered by bundle: logmailboxlistener_0.0.0 [10]
  No bundles using service.
```

As before the new component is delayed, meaning we see the `MailboxListener` service in the service registry, but the implementation class has not been constructed yet because no consumer bundle has attempted to call the service. We can take a closer look at the state of the component using the `list` and `component` commands:

```
osgi> list
All Components:
ID  State      Component Name           Located in bundle
1   Satisfied  org...HelloComponent     minimal_ds(bid=4)
2   Satisfied  org...ConsoleLog         consolelog_ds_immediate(bid=9)
3   Satisfied  org...LogMailboxListener logmailboxlistener(bid=10)

osgi> component 3
   Component[
   name = org.osgi.book.ds.logmailboxlistener.LogMailboxListener
   activate = activate
   deactivate = deactivate
   modified =
   configuration-policy = optional
   factory = null
   autoenable = true
   immediate = false
   implementation = org...LogMailboxListener
   properties = null
   serviceFactory = false
   serviceInterface = [org.osgi.book.reader.api.MailboxListener]
   references = {
      Reference[name = org.osgi.book.logapi.Log,
         interface = org.osgi.book.logapi.Log, policy = static,
         cardinality = 1..1, target = null, bind = setLog,
         unbind = null] }
   located in bundle = logmailboxlistener_0.0.0 [10] ]
Dynamic information :
  The component is satisfied
  All component references are satisfied
  Component configurations :
    Configuration properties:
      component.name = org...LogMailboxListener
      component.id = 3
      objectClass = Object[org.osgi.book.reader.api.MailboxListener]
    Instances:

osgi>
```

This command has given us a lot of useful information. In particular it's very helpful to see that all the component references are "satisfied", indicating that a matching service is available. Because of this the state of the overall component is "satisfied", meaning it is ready to be used.

An interesting point to note is that the `ConsoleLog` component from the previous example has still not been instantiated (as long as we are not using the "immediate" variant from Section 11.5.2). Although it is being used by the `LogMailboxListener` component to satisfy its reference to a log service, that component is also delayed, so neither of them need to be created yet. When a consumer tries to use the `LogMailboxListener`, SCR will be finally forced to create both components.

To test that, we reuse the "growable" mailbox example from Chapter 7. Recall that in that example we used a timer thread to add messages into a mailbox every five seconds, which in turn caused the mailbox to notify all registered `MailboxListener` services of the arrival of the new message, using the whiteboard pattern. Therefore it should automatically find the `LogMailboxListener` service that we have registered using DS; we just need to install and start the growable mailbox bundle:

```
osgi> install file:growable_mbox.jar
Bundle id is 12

osgi> start 12
LogMailboxListener created
ConsoleLog created

osgi> LOG: 1 message(s) arrived in mailbox growing
LOG: 1 message(s) arrived in mailbox growing
LOG: 1 message(s) arrived in mailbox growing
...
```

At last we see the output from the constructors of both the `ConsoleLog` and `LogMailboxListener` classes. Here's what happened, in sequence:

1. When the growable mailbox bundle was started, it started tracking instances of `MailboxListener`. In the `addingService` method of the tracker, the actual service object was retrieved using a `getService` call.
2. As a result, SCR was forced to instantiate the `LogMailboxListener` class.
3. In order to provide `LogMailboxListener` with an instance of the `Log` service, SCR instantiated the `ConsoleLog` class.
4. The timer thread of the growable mailbox bundle calls `addMessage` on the `GrowableMailbox` class.
5. The mailbox calls `WhiteboardHelper` with a visitor that calls `messagesArrived` on each registered `MailboxListener`, and this included our `LogMailboxListener`.
6. The `LogMailboxListener` receives the call to `messagesArrived` and formats a new log entry, which it passes to the `Log` service.
7. `ConsoleLog` prints the log message to the console.

The output will continue until any part of the chain is broken. Here are some of the ways that the chain might be broken:

- We could stop the growable mailbox bundle (`stop 12`), terminating its thread and cutting off the ultimate source of "message arrival" events.
- We could stop the `logmailboxlistener` bundle (`stop 10`), causing the `MailboxListener` service to be unregistered. The growable mailbox would still be trying to deliver events but it would have no listeners to deliver them to.
- We could stop the `consolelog` bundle (`stop 7`), causing the `Log` service to be unregistered. This would force SCR to unregister the `LogMailboxListener` service, since its reference to a `Log` service can no longer be satisfied. Thus the growable mailbox would have no listeners to deliver events to.

11.6.1. Optional vs Mandatory References

When composing services together using `ServiceTracker`s in Chapter 4, we found there was a big difference in usage patterns between *optional* service dependencies and *mandatory* dependencies. Not only did we have to write significant amounts of glue code, but that glue code was completely different when we wanted a mandatory dependency versus an optional one. It certainly discourages us from changing our minds too often.

As we would hope, DS makes this much easier: we simply need to add a declaration to switch between optional and mandatory.

First, let's remind ourselves what "optional" and "mandatory" mean in the context of service dependencies. An optional dependency means that our component can continue to function without the target service being present; in the example above, it would mean our `LogMailboxListener` should be created even if there is no `Log` service available. A mandatory dependency means that our component *cannot operate* without the presence of the target service, so our `LogMaiboxListener` should not be created when there is no `Log`. Also if a component instance is bound to a service and that service goes away, then that component should be destroyed. Destruction of a component simply means that any services it registers will be unregistered, so it will no longer be available to clients. The component object itself will be discarded by SCR, becoming available for garbage collection some time later. If the referenced service subsequently comes back, then SCR will not attempt to resurrect the same object, but will create a brand new component instance.

In DS, references to services are mandatory by default, so if we were to remove the `Log` service instance provided by `ConsoleLog` then the `LogMailboxListener`

would be unregistered also. We can switch the reference to optional by adding the attribute `cardinality="0..1"` to the `reference` element as shown in Listing 11.13

Listing 11.13 DS Declaration for the Optional Logging Mailbox Listener

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <!-- "optlogmailboxlistener.xml" -->
3 <scr:component xmlns:scr="http://www.osgi.org/xmlns/scr/v1.1.0">
4     <implementation
5       class="org.osgi.book.ds.mboxlistener.OptLogMailboxListener"/>

7     <service>
8       <provide interface="org.osgi.book.reader.api.MailboxListener"/>
9     </service>

11    <reference interface="org.osgi.book.logapi.Log" bind="setLog"
12      cardinality="0..1"/>
13 </scr:component>
```

The value "0..1" simply means that the reference will remain "satisfied" even when zero instances of the service are available. Mandatory references may be indicated with a cardinality of "1..1", but since this is the default it is not necessary to state it explicitly. The "1" on the right hand side after the dots does *not* mean there must be a maximum of only one service available – it means that if there are many services available, the reference will bind to only one of them. As you might have already guessed, we can bind to multiple instances by specifying either "0..*n*" or "1..*n*". Multiple-instance references will be discussed later, in Section 11.8. For now we will discuss only single-instance or unary references.

To see this working, we need to make a small fix to the code for the `LogMailboxListener` so that it can handle the `log` field being `null` without throwing a `NullPointerException`. In the code for the `messagesArrived` method above, it was assumed that the `log` field could not be `null`, which was a valid assumption as long as the reference was mandatory. Now we need to insert an `if` statement to check that the `log` field is non-`null` before calling it. For this reason we use a new class `OptLogMailboxListener`, which is identical to `LogMailboxListener` except for the `messagesArrived` method, shown in Listing 11.14. In order to see what is going on, this version of the method prints a warning if the `log` field is `null`, but the warning is printed directly to the console rather than going via the `ConsoleLog` component.

We can build a new bundle for the optional variant named `optlogmailboxlistener`, for which the `bnd` descriptor is omitted as it is trivially derived `logmailboxlistener`. Next we can stop the mandatory variant and install and start the optional one. Remember, at this point we still have a `Log` service available from the `consolelog_ds` bundle, and the "growable" mailbox is still sending out events every five seconds:

```
osgi> stop 10
```

Listing 11.14 Logging Mailbox Listener with Optional Log Field

```
1    public void messagesArrived(String mboxName, Mailbox mbox,
2                                long[] ids) {
3       if(log != null) log.log(Log.INFO, ids.length +
4             " message(s) arrived in mailbox " + mboxName, null);
5       else System.err.println("No log available!");
6    }
```

```
osgi> install file:optmailboxlistener.jar
Bundle id is 13

osgi> start 13
OptLogMailboxListener created
LOG: 1 message(s) arrived in mailbox growing
LOG: 1 message(s) arrived in mailbox growing
...
```

We continue to see the log output from the `ConsoleLog` component. Now let's remove the `Log` service by stopping the `consolelog_ds` bundle:

```
osgi> stop 7
OptLogMailboxListener created

No log available!
No log available!
...
```

As expected, the log messages now stop because there is no log, but the warning messages tell us that the listener's `messageArrived` method is still being called, proving that the `OptLogMailboxListener` component is still active and registered.

A very useful aspect of DS is that it easy is to mix and match optional and mandatory service references. With `ServiceTracker` it can be difficult to create a dependency on multiple service types, and *very* difficult if we need a mixture of optional and mandatory dependencies. DS takes away this pain, allowing us to add as many references as we like with whatever mix of cardinalities we like. Listing 11.15 shows this flexibility. Here we have a `Mailbox` implementation that reads messages from a database connection, and sends messages to a log. In such a component, the database connection would clearly be mandatory whilst the log would be optional. Listing 11.16 shows an excerpt of the implementation class for this service.

11.6.2. Static vs Dynamic References

You might have noticed something strange in the output from the previous example, when the optional log service was removed. As expected, we continued to see lines of debug output from the `messagesArrived` method, which

Listing 11.15 DS Declaration for a Database Mailbox

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- "dbmailbox.xml" -->
<component xmlns:scr="http://www.osgi.org/xmlns/scr/v1.1.0">
    <implementation
        class="org.osgi.book.ds.dbmailbox.DbMailbox"/>

    <service>
        <provide interface="org.osgi.book.reader.api.Mailbox"/>
    </service>

    <reference interface="java.sql.Connection"
        bind="setConnection"
        cardinality="1..1"/>

    <reference interface="org.osgi.book.logapi.Log"
        bind="setLog"
        cardinality="0..1"/>
</component>
```

Listing 11.16 The Database Mailbox Class (Excerpt)

```
package org.osgi.book.ds.dbmailbox;

import java.sql.*;

import org.osgi.book.logapi.Log;
import org.osgi.book.reader.api.*;

public class DbMailbox implements Mailbox {

   private volatile Connection connection;
   private volatile Log log;

   public void setConnection(Connection connection) {
      this.connection = connection;
   }

   public void setLog(Log log) {
      this.log = log;
   }

   public void markRead(boolean read, long[] ids)
         throws MailboxException {
      int count = ids.length;
      if (log != null)
         log.log(Log.DEBUG, "Marking " + count + " messages read.",
               null);
      try {
         PreparedStatement stmt = connection.prepareStatement(
               "UPDATE msgs SET read=1 WHERE ...");
         stmt.executeUpdate();
      } catch (SQLException e) {
         throw new MailboxException(e);
      }
   }

   // ...
```

meant that the service was still registered. But we saw one line of output that we did not expect. Here it is again:

```
osgi> stop 7
OptLogMailboxListener created

No log available!
No log available!
...
```

On the second line we see the output from the constructor of our `OptLogMailboxListener` class! Why has this appeared? Has SCR created a second instance of our service?

Yes and no. In fact SCR is attempting to protect us from the need to worry about dynamics and concurrency. We know that services can come and go at any time and in any thread, so we would normally have to deal with the `log` field being changed even while we are using it. At the very least, this would make the `null` check in `OptLogMailboxListener`'s `messgesArrived` method unsafe, because the `log` field could become `null` immediately after we check its value. But in fact our code is perfectly safe because SCR by default does not dynamically change the value of a service reference – instead, it discards the component instance and creates a new instance. If the component is also published as a service then the service registration is removed and replaced when this happens.

This is known as the "static policy" for service references, and you might worry that it is inefficient. In fact most of the time it is not too so bad: object creation in Java is cheap, especially when we have already previously created an instance of the class, and in the examples above our components do not require any elaborate initialisation. But of course it is not *always* cheap, so we need a way to override this default. Fortunately there is indeed a way, by enabling the "dynamic policy". But first we will take a look at another consequence of the static policy.

11.6.3. Minimising Churn

In the previous example we saw that removing the `Log` service forced SCR to discard our `OptLogMailboxListener` component and recreate it with the log reference unbound. What happens if we then reinstate the log service? Let's take a look:

```
osgi> start 7

No log available!
No log available!
...
```

Absolutely nothing happens! We can verify with the `services` command that the `Log` service is now available, but our component has not picked it up, and instead it continues to print the warning about there being no log.

This may seem like an error, but it is quite deliberate. As we are using static policy, SCR would have to discard and recreate our component in order to bind to the newly available log service. But our dependency on the log is optional after all, so why go to so much effort? SCR tries to reduce the number of times it must recreate components, as this generates *churn*.

Imagine if we had a large application constructed of DS services, all wired together with references using the static policy, forming an interconnected network. Recreating a component in this network would require recreating any components which depend on it, and any components which depend on them, and so on. Small changes would become amplified into tidal waves of components being discarded and recreated. This is churn and it is expensive, so SCR minimises it by never recreating a component unless it is *forced* to do so.

When a service that was previously bound to a component goes away, SCR has no choice but to recreate the component in order to unbind the service. But if a component's optional reference is unbound and then a service comes along, SCR could choose to either recreate the component in order to bind the service, or do nothing and leave the reference unbound. SCR will always choose to do nothing.

An unfortunate consequence of this is that optional static references are likely to spend much of their life in the unbound state even when services are available to bind to them. Indeed, the only time such a reference will be bound is if it is lucky enough to find a service when the component is first started. Therefore when we want an optional reference it is generally better to use the dynamic policy, which allows an unbound reference to be bound later on.

11.6.4. Implementing the Dynamic Policy

DS allows us to switch between static and dynamic policies by adding the `policy` attribute to the `reference` element. Setting this to `dynamic` simply informs SCR that our component is capable of being updated with changing references. Naturally, this exposes us to the full horror of services being changed in multiple threads while we are using them, but DS still makes it much easier to handle than when we were using the low-level APIs.

Listing 11.17 shows the new XML descriptor. In addition to the new `policy` attribute, there is also an `unbind` attribute that specifies the name of a method that will be called by SCR to notify us when the service we were bound to

goes away[6].

Listing 11.17 DS Declaration for the Dynamic Logging Mailbox Listener

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <!-- "dynlogmailboxlistener.xml" -->
3  <scr:component xmlns:scr="http://www.osgi.org/xmlns/scr/v1.1.0">
4      <implementation
5        class="org.osgi.book.ds.mboxlistener.DynLogMailboxListener"/>

7      <service>
8        <provide interface="org.osgi.book.reader.api.MailboxListener"/>
9      </service>

11     <reference interface="org.osgi.book.logapi.Log" bind="setLog"
12       unbind="unsetLog" policy="dynamic"
13       cardinality="0..1"/>
14 </scr:component>
```

How should we change our mailbox listener class to cope with the dynamics? There are a few ways we might think of doing this safely... first, let's look at the *wrong* answers, i.e. the approaches that will *not* make our code thread-safe:

- Make the `log` field `volatile`. Unfortunately this is not enough to be thread-safe. Volatile fields are good for visibility, as any changes to their value are immediately propagated to other threads. But they cannot offer atomicity, so the `null`-check in the `messagesArrived` method can still fail if the content of the `log` field changes in between checking it is non-`null` and calling it.
- Wrap all accesses to the `log` field in `synchronized` blocks or methods. For example, we could make the `setLog`, `unsetLog` and `messagesArrived` methods all `synchronized`. This gives us atomicity but it reduces the concurrency of the system, as only one thread can call our service at a time. Also we should not hold a lock while calling a service method (see Section 6.4 on page 128).

The best approach is to take a copy of the `log` object into a local variable before we use it. Then we can test the variable and call the service only if its value is non-`null`. Since there is no way for the value of a local variable to be altered by another thread, there is no need for the call to the service to happen inside a `synchronized` block, though we still need the copy itself be synchronized for visibility purposes, and the `setLog` and `unsetLog` methods also need to be synchronized[7]. Listing 11.18 shows the `messagesArrived` method using this approach.

[6] The `unbind` attribute can also be used on references with the static policy but it is generally not useful to do so, because when this method is called the component is about to be discarded anyway.

[7] You may wonder why we don't use a `volatile` field instead of `synchronized` blocks. It's true that, for the `messagesArrived` method, we only need visibility and not atomicity. However the `unsetLog` method will need atomicity also, as we will soon see.

Unfortunately it is rather too easy to forget to perform the copy, which must be done *every time* the field is accessed. We could omit the first two lines of the method in Listing 11.18 and the compiler would not complain since the field is of the same type as the local variable. It would be helpful if we could get the compiler to help us to stay safe.

Listing 11.18 Handling Dynamics by Copying a Field)

```
1    public void messagesArrived(String mboxName, Mailbox mbox,
2                                long[] ids) {
3        Log log;
4        synchronized(this) { log = this.log; }

6        if(log != null) log.log(Log.INFO, ids.length +
7              " message(s) arrived in mailbox " + mboxName, null);
8        else System.err.println("No log available!");
9    }
```

Listing 11.19 shows a safe and lightweight approach using the `AtomicReference` class from Java 5's concurrency package. Atomic references offer visibility guarantees like `volatile`, but they support a few additional atomic operations that cannot be performed safely on `volatile`s without synchronization. The best thing about them though is they are the *wrong type* for us to accidentally use without first taking a copy of the contained value into a local variable. That is, in the `messagesArrived` method it would be impossible to call `logRef.log` directly because `logRef` is of type `AtomicReference<Log>`, and there is no `log` method on the `AtomicReference` class!

11.6.5. Service Replacement

You might be wondering whether the static/dynamic policy decision has any effect on mandatory references. With an optional reference, it is clear that the dynamic policy gives us the ability to bind and unbind the service without recreating the component. But when we have a mandatory reference and the target services goes away, then the component must be destroyed, irrespective of whether it is static or dynamic. So does it make sense to specify the dynamic policy for mandatory references?

The answer is yes, it does. Though both static and dynamic references act the same when a mandatory reference becomes unsatisfied, the results are different when a service is *replaced*.

Suppose that our component is bound to the `Log` service, but there are currently two available implementations of the service, published by different bundles. Let's call them A and B, and our component is bound to A as shown on the left hand side of Figure 11.1. Now service A goes away because its bundle is stopped... what happens next? In this scenario it is not necessary

Listing 11.19 Handling Dynamics with Atomic References (Excerpt)

```
1  package org.osgi.book.ds.mboxlistener;
2
3  import java.util.concurrent.atomic.AtomicReference;
4
5  import org.osgi.book.logapi.Log;
6  import org.osgi.book.reader.api.*;
7
8  public class DynLogMailboxListener implements MailboxListener {
9
10    private final AtomicReference<Log> logRef
11          = new AtomicReference<Log>();
12
13    public DynLogMailboxListener() {
14       System.out.println("DynLogMailboxListener created");
15    }
16
17    public void setLog(Log log) {
18       System.out.println("DynLogMailboxListener.setLog invoked");
19       logRef.set(log);
20    }
21
22    // Implements MailboxListener.messagesArrived(...)
23    public void messagesArrived(String mboxName, Mailbox mbox,
24                                long[] ids) {
25       Log log = logRef.get();
26
27       if(log != null) log.log(Log.INFO, ids.length +
28             " message(s) arrived in mailbox " + mboxName, null);
29       else System.err.println("No log available!");
30    }
```

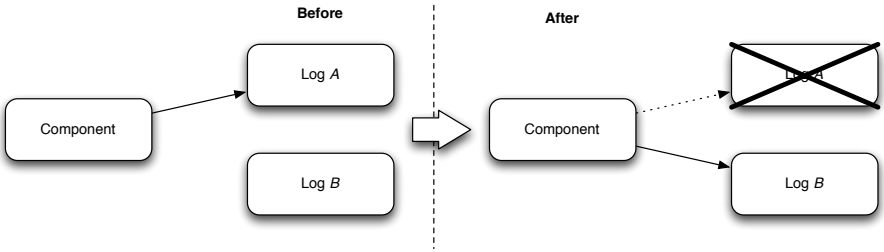


Figure 11.1.: Service Replacement, Before and After

to disable the component, because there is still a `Log` service available, namely *B*. SCR needs to update our component to bind to *B*, but under the static policy the only way to rebind to *B* is to recreate the component, whereas the dynamic policy allows SCR to switch to *B* on the fly.

An interesting "gotcha" here is the order in which the bind and unbind methods of a dynamic reference will be called. During service replacement, the bind method will be called *first* with the new service object, and *then* unbind will be called with the old service. The reason SCR calls the methods this way around is to give our component an opportunity to perform some clean-up operation with both the old and new service visible at the same time, albeit briefly. Unfortunately it means that the "natural" implementations of `unsetLog` shown in Listing 11.20 are incorrect and broken.

Listing 11.20 Incorrect Implementation of Unbind Methods

```
1    // Synchronized version -- wrong!
2    public synchronized void unsetLog(Log log) {
3        this.log = null;
4    }

6    // AtomicReference version -- wrong!
7    public void unsetLog(Log log) {
8        logRef.set(null);
9    }
```

This is why `unsetLog` was omitted from the code for `DynLogMailboxListener` in Listing 11.19. A correct implementation of an unbind method needs to handle both of the following cases:

- Unbind is called with the service that is currently bound to the component. This means that the service is being unbound, so we `null` out the reference.
- Unbind is called with a service other than that currently bound to the component. This means that the service is being replaced, and the bind

method was already called with the value of the new service. In this case there is nothing for us to do.

The correct implementation requires us to check that the provided parameter is the bound service before we set our field to `null`[8]. When using `sychronized` we can do a simple equality check as shown in Listing 11.21. If we are using an `AtomicReference` then we can take advantage of one of the special atomic operations offered by that class: the `compareAndSet` method takes two parameters, and it will set the content of the reference to the right-hand value if, and only if, the current content of the reference is equal to the left-hand value. This is performed as a single non-blocking atomic operation.

Listing 11.21 Correct Implementation of Unbind Methods

```
1   // Synchronized version
2   public synchronized void unsetLog(Log log) {
3      if(this.log == log)
4         this.log = null;
5   }

7   // AtomicReference version
8   public void unsetLog(Log log) {
9      logRef.compareAndSet(log, null);
10  }
```

This completes the implementation of `DynLogMailboxListener` from Listing 11.19, so we can now go ahead and test that it works.

11.6.6. Running the Example

To try out the dynamic policy variant of the mailbox listener we should first stop the static-policy variant, `optlogmailboxlistener`, otherwise we could get confused about the messages printed to the console from both listeners. Next we install and start the `dynlogmailboxlistener` bundle (the `bnd` descriptor is again omitted because it is trivially derived from previous examples). At this point the `ConsoleLog` service is currently available, so we should see the following output:

```
osgi> stop 13

osgi> install file:dynlogmailboxlistener.jar
Bundle id is 14

osgi> start 14
DynLogMailboxListener created

osgi> LOG: 1 message(s) arrived in mailbox growing
LOG: 1 message(s) arrived in mailbox growing
...
```

[8] This explains why `volatile` cannot be used: there is an unavoidable check-then-act sequence in the unbind method.

Now remove the log service, wait a little, and reinstate it again:

```
osgi> stop 7

No log available!
No log available!
osgi> start 7
ConsoleLog created
LOG: 1 message(s) arrived in mailbox growing
LOG: 1 message(s) arrived in mailbox growing
...
```

We should be able to see our component being dynamically bound and unbound each time the log service is stopped and started, without any need for it to be restarted.

11.6.7. Minimising Churn with Dynamic References

In Section 11.6.3 we discussed the concept of churn, and the reasons why SCR does not rebind a static, optional reference when a service becomes available to satisfy the reference.

Now consider another scenario: our component is currently bound to a service instance, let's call it A. Then a new service of the same type becomes available, B, which has a much higher ranking than A. Let's say that the value of A's `service.ranking` property is zero while B's is 10. Should SCR unbind the reference to A and rebind it to B?

In the case of a static policy reference, the answer should be obvious. No, SCR will not discard and recreate a component merely to supply it with a "better" reference. The same rule applies here as it did before – components are only recreated when SCR has no choice, but in this scenario SCR has the choice to do nothing. Therefore, it does nothing.

Less obviously, the same is true for a dynamic policy reference. Though the churn created by rebinding a dynamic reference is less than that for a static reference, it is still not zero since our component may need to do some internal work when its service references are re-bound. SCR will always favour doing nothing, i.e. a bound service reference will never be re-bound unless the bound service has become unavailable.

So when is the ranking of a service actually used by SCR? Only in the following cases:

- A component is starting up, and there are multiple possible instances that could be chosen to satisfy its reference. SCR will chose the one with the highest ranking.
- Many (i.e. more than two) instances of a service are available and a component is bound to one of them; but then the bound service goes

away. SCR will use ranking to decide which of the remaining services should be chosen to rebind the reference.

11.6.8. Recap of Dynamic Reference Implementation

Listing 11.22 shows a template summarising the correct coding pattern for dynamic service references with DS, given a service reference of type `Foo`.

Listing 11.22 Template for Dynamic Service Usage with DS

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- "dynamic_template.xml" -->
<scr:component xmlns:scr="http://www.osgi.org/xmlns/scr/v1.1.0">
   <implementation
      class="org.osgi.book.ds.DynamicReferenceTemplate"/>

   <!-- possible <service> element here -->

   <reference interface="org.osgi.book.ds.Foo" bind="setFoo"
     unbind="unsetFoo" policy="dynamic"
     cardinality="0..1"/>
</scr:component>
```

```
package org.osgi.book.ds;

import java.util.concurrent.atomic.AtomicReference;

public class DynamicReferenceTemplate {

   private final AtomicReference<Foo> fooRef =
      new AtomicReference<Foo>();

   public void setFoo(Foo foo) {
      fooRef.set(foo);
   }

   public void unsetFoo(Foo foo) {
      fooRef.compareAndSet(foo, null);
   }

   public void doSomething() {
      Foo foo = fooRef.get();
      if(foo != null) {
         // do something with foo
      }
   }
}
```

11.7. Component Lifecycle

Alongside the ability to offer and consume services, many components need *lifecycle*. Just like bundles, components can be active or inactive, and in their active state they can do work that is not related to offering services. Examples

include opening server sockets, running threads, monitoring system devices, and so on. Therefore DS supports *activation* and *deactivation* of components.

Thanks to this lifecycle support, DS components can largely replace the use of `BundleActivators`. In fact they have a great advantage over bundle activators because they can very easily access services to do their work. This is why we said that in Section 11.5.1 that DS bundles do not really need activators.

In the simplest case, we can take advantage of lifecycle in our components simply by adding methods named `activate` and `deactivate`, without parameters. Listing 11.23 shows an example in which we reimplement the `HeartbeatActivator` example from Chapter 2 (see Listing 2.7 on page 42) but with a small difference: instead of printing messages directly to the console every five seconds, we send messages to the log service, as long as it is available.

Listing 11.23 The Heartbeat Component

```
1  package org.osgi.book.ds.lifecycle;
2
3  import org.osgi.book.logapi.Log;
4
5  public class HeartbeatComponent {
6
7      private Thread thread;
8      private Log log;
9
10     public HeartbeatComponent() {
11         System.out.println("HeartbeatComponent created");
12     }
13
14     public void activate() {
15         System.out.println("HeartbeatComponent activated");
16         thread = new Thread(new Runnable() {
17             public void run() {
18                 try {
19                     while (!Thread.currentThread().isInterrupted()) {
20                         Thread.sleep(5000);
21                         if(log != null)
22                             log.log(Log.INFO, "I'm still here!", null);
23                         else
24                             System.err.println("No log available!");
25                     }
26                 } catch (InterruptedException e) {
27                     System.out.println("I'm going now.");
28                 }
29             }
30         });
31         thread.start();
32     }
33     public void deactivate() {
34         System.out.println("HeartbeatComponent deactivated");
35         thread.interrupt();
36     }
37     public void setLog(Log log) {
38         System.out.println("HeartbeatComponent.setLog() invoked");
39         this.log = log;
40     }
41 }
```

Listing 11.24 shows the XML declaration. SCR automatically detects the `activate` and `deactivate` methods on our component class so we don't need to declare them in the XML. We can however choose to call them something else, for example `begin` and `end`, in which case we need to add two attributes `activate="begin"` and `deactivate="end"` to the top-level `component` element of the XML.

Listing 11.24 The Heartbeat Component, XML Descriptor

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <scr:component xmlns:scr="http://www.osgi.org/xmlns/scr/v1.1.0">
3    <implementation
4       class="org.osgi.book.ds.lifecycle.HeartbeatComponent"/>
5
6    <reference interface="org.osgi.book.logapi.Log" bind="setLog"
7       cardinality="0..1"/>
8 </scr:component>
```

Let's take this example for a spin, ensuring that the `ConsoleLog` service is still available.

```
osgi> start 7

osgi> install file:heartbeat_component.jar
Bundle id is 15

osgi> start 15
HeartbeatComponent created
ConsoleLog created
HeartbeatComponent.setLog() invoked
HeartbeatComponent activated

osgi> LOG: I'm still here!
LOG: I'm still here!
...
```

Since the component does not offer a service, and its reference to the log service is satisfiable, it is created immediately, passed a reference to the log and activated. Then it starts to send log messages every five seconds.

Now let's shutdown the log service:

```
osgi> stop 7
HeartbeatComponent deactivated
I'm going now.
HeartbeatComponent created
HeartbeatComponent activated
No log available!
No log available!
...
```

We are using the static policy so we expect to see our component discarded and recreated when the log service goes away. In addition we see that the `deactivate` method is called before the component is discarded, and the `activate` method is called on the new instance. This is an important feature, as it means SCR does not simply "forget" about objects it discards, but ensures they can be cleaned up in an orderly fashion.

11.7.1. Lifecycle and Service Binding/Unbinding

From the debugging output above we saw that the bind method for the service reference was called before the activation method. For a mandatory reference this is important since the component should never be in an active state when the service reference is unsatisfied. However with a dynamic, optional reference we should be able to bind and unbind services without activating or deactivating the component.

Let's take a look at how the lifecycle methods interact with the bind and unbind methods:

Static References

With a static reference, the bind method *must* be called before the `activate` method. However in the case of an optional reference the bind method may not be called at all. If bind has still not been called by the time `activate` is called then our component can assume that the reference will remain unbound throughout its lifetime.

When a bound service goes away, the `deactive` method *must* be called before the unbind. However we usually do not bother to implement unbind on a static reference.

Dynamic References

Dynamic, mandatory references work largely the same as static ones: the bind must be called before `activate`, and `deactivate` must be called before unbind.

However there is a difference when service replacement occurs, i.e. the bound service goes away but an alternative service is immediately available. In this case, the bind and unbind methods are called without deactivating and activating the component, i.e. while the component remains active. This can even happen concurrently while our activate or deactivate method is running in another thread.

With a dynamic, optional reference, the bind and unbind methods can really be called at *any* time, including concurrently with activate/deactivate.

11.7.2. Handling Errors in Component Lifecycle Methods

It is possible that during the activation of our component, we might detect an error condition that means the component cannot start. For example we

might find that some required resource is not available, or an unexpected error might occur. The proper way to deal with this is to throw an exception from the activate method, and we are free to throw whatever exception types we like, either checked or unchecked.

When this happens, SCR catches the exception[9] and deals with it as follows. First, it writes a message to the OSGi log, if there is an instance of the `LogService` available. Then it stops the activation of the component from going ahead. If the component includes a provided service, then either the service will not be published or the existing service registration will be withdrawn.

Activation errors may create a small problem for consumers of our services. As we have seen, components that offer services are usually not created or activated until the service is actually used for the first time; technically, our activate method will be executed when a consumer first calls `BundleContext.getService` on the published service. Only at that point does our activate method have a chance to detect an error condition and throw an exception, but it is already too late to unpublish the service because the consumer bundle has already seen it. Therefore the call to `BundleContext.getService` will return `null`, which the consumer must be able to handle.

In most cases this will be fine, since well-behaved service consumers should always be aware that `null` may be returned from `getService`. And if the consumer is another DS component using a service reference element, then we do not have to worry at all because SCR will handle the error. For example, if the other DS component has a mandatory reference to our service but our activate method throws an error, then that component will remain in the "unsatisfied" state.

Incidentally, the deactivate method may also throw an exception, but in this case SCR simply logs the error and continues.

11.8. Unary vs Multiple References

As we mentioned when the `cardinality` attribute of a reference was introduced, we can also request SCR to bind our component to multiple instances of the target service, rather than just one. This is done by setting the `cardinality` attribute to either "0..n" or "1..n". The first of these is "optional and multiple", meaning that the component can continue to exist even when there are no services available for the reference. The latter is "mandatory and multiple" meaning that at least one service must exist, otherwise the

[9] Note that SCR will not catch instances of `java.lang.Error` or its subclasses, since these indicate error conditions that cannot be reasonably handled by an application, e.g., "out of memory".

component will be deactivated[10]. Multiple references are commonly used to implement the whiteboard pattern (see Chaper 7) using DS.

At this point there should be nothing difficult or surprising about the way multiple references work: SCR will simply call our bind method once for each service that is available, and call unbind for each bound service that goes away.

11.8.1. Static Policy with Multiple References

This begs the question of what happens when the static policy is used for a multiple reference. Suppose when our component starts there are nine services available for it to bind to: SCR will call bind nine times – once for each service instance – before activating and publishing the component.

Then one of those nine services goes away. Now SCR must deactivate the component, call unbind nine times (if we have provided an unbind method), create a new component instance and call bind eight times before activating it. But what if a new service arrives? SCR will simply ignore it, for the same reason it refuses to re-bind an optional reference, as described in Section 11.6.3. Thus there is continuous downwards pressure on the number of services bound to a static multiple reference, so it will inevitably trend towards zero.

Because of this effect, it is the author's opinion that static policy is nearly always inappropriate for use with a multiple reference. Therefore in the examples to follow we will use dynamic policy exclusively.

11.8.2. Implementing Multiple References

If using dynamic policy to implement a multiple reference then we must also be prepared to deal with the bind/unbind methods being called concurrency while we are trying to use the services in another thread. This is the same problem we had in Section 11.6.4 but now instead of managing just a single object we must manage a collection of objects safely.

The key challenge is that, when we use the services, we typically have to iterate over the entire collection. For example a component that implements the whiteboard pattern would iterate over every service reference in order to invoke the listener methods. While this happens the collection contents must not change, so we would have to perform the entire iteration inside a `synchronized` block, but as we learned in Chapter 6 it is dangerous to hold such a lock while invoking services. Also the bind/unbind methods would be held up for as long as the iteration is running.

[10] Unfortunately DS does not support any other cardinalities besides these, so we cannot ask for a cardinality of, say, "2..n" or "0..5".

As before, we could choose to take a copy of the collection into a local variable, but this would be inefficient if executed many times. It is better to make the common case fast, perhaps at the expense of the less common case. We are likely to *access* the collection far more often than we *update* the collection, because updating only happens during bind or unbind. There is a collection class in the Java 5 concurrency APIs that makes exactly the desired trade-off: `CopyOnWriteArrayList`. It allows us direct access to the collection without copying for read operations, and internally copies the contents when a write operation occurs. Crucially an `Iterator` over this collection type will not be affected by changes to the collection contents that occur after the `iterator` method is called, so we can write a straightforward iteration block without worrying at all about concurrent changes. Listing 11.25 shows a component which does exactly this – note the lack of any explicit synchronization or defensive copying.

Listing 11.25 Implementing Multiple References

```
1  package org.osgi.book.ds.multiple;
2
3  import java.util.Collection;
4  import java.util.concurrent.CopyOnWriteArrayList;
5
6  import org.osgi.book.reader.api.MailboxListener;
7
8  public class WhiteboardComponent {
9
10    private final Collection<MailboxListener> listeners
11       = new CopyOnWriteArrayList<MailboxListener>();
12
13    public void fireMessagesArrived() {
14       for (MailboxListener listener : listeners) {
15          listener.messagesArrived("mailbox", null, new long[0]);
16       }
17    }
18    public void addMailboxListener(MailboxListener listener) {
19       listeners.add(listener);
20    }
21    public void removeMailboxListener(MailboxListener listener) {
22       listeners.remove(listener);
23    }
24 }
```

11.9. Discussion: Are These True POJOs?

In the introduction to this chapter, on page 221, it was claimed that DS components are POJOs. However we have seen now some programming idioms that are probably quite unfamiliar to developers who have worked with only with traditional component containers, such as static dependency injection frameworks. The differences range from the presence of additional "unset" methods, to the thread-safe approach we must take when updating references

under the dynamic policy. It *may* seem that the necessity to use such idioms violates the POJO principle.

But what does POJO – "Plain Old Java Object" – really mean? The term came about as a reaction to heavyweight frameworks, in particular the Enterprise JavaBeans (EJB) specification versions 1 and 2, in which "beans" had to implement special interfaces and perform all their interaction with other beans and resources via look-ups executed against the EJB container, using its API. This made those beans almost entirely reliant on their container, and they were very hard to unit test because they could not be isolated from the container or from each other. Dependency injection frameworks allowed us to write POJOs, with no reliance on any container.

Likewise, DS components have no reliance on their container at all. It should be easy to see that *any* of the component classes discussed so far in this chapter can be instantiated and used outside of DS and outside of the OSGi framework, whether for the purposes of testing or for deployment to an alternative runtime. We do not even need OSGi libraries on our compilation classpath. Therefore these components are clearly still POJOs, despite their slightly more complicated internals.

In fact, DS components are arguable *more* widely deployable than any other kind of POJO since they make fewer assumptions about the container or environment in which they will be executed. Static dependency injection frameworks are great for removing the API dependencies and look-ups from our component code, but they make no provisions for components to be dynamically unwired or re-wired, nor do they typically support the concept of optional wiring. Therefore a POJO written to such a container usually assumes that the dependencies given to it will never become invalid, and it assumes that its setters and lifecycle methods will not be called concurrently from different threads. In effect, it assumes it will always be deployed to such a non-threatening, static environment, and it will fail if deployed to a more dynamic environment in which multi-threading is a factor. Thus it still has a dependency on one very important aspect of its container! By contrast, well-written DS components assume that concurrency may happen, and they cope with it; yet they behave perfectly correctly in a static environment too.

The POJO movement exhorts us to keep our components plain, but there is no requirement for them to be dumb. The components we have developed in this chapter have been *smart* (or *safe*) POJOs, which run correctly under both a static dependency injection framework and under a dynamic, multi-threaded environment.

Unfortunately, "Smart" POJOs are (currently) slightly harder to write than dumb ones. It is the author's hope that better tools or languages will help to close this gap. There also remains the problem of legacy component code that may need to be ported from a static DI framework, but for those components

we use DS's static policy with mandatory service reference.

11.10. Using Bnd to Generate XML Descriptors

Now that we have seen many examples of DS XML descriptors, you should have a firm understanding of what they contain and how they are structured. You may also be quite tired of typing them in, as XML is a verbose and fussy format. Fortunately `bnd` offers us two ways to avoid writing them entirely.

11.10.1. Bnd Headers for XML Generation

The simplest approach to generating DS component XML descriptors from `bnd` is by writing instructions directly in the `bnd` descriptor file. This offers a much more accessible and less verbose approach to specifying the required metadata.

Generation from `bnd` headers is be done by extending the semantics of the `Service-Component` header. Rather than specifying the path to a resource inside the bundle, we can instead specify the name of a class, and then further annotate it with attributes. For example Listing 11.26 shows an alternative version of the `bnd` descriptor for `LogMailboxListener`, originally seen in Listing 11.12.

Listing 11.26 XML Generation Header for the Logging Mailbox Listener

```
# logmailboxlistener.bnd
Private-Package: org.osgi.book.ds.mboxlistener
Service-Component: org.osgi.book.ds.mboxlistener.LogMailboxListener;\
   provide:=org.osgi.book.reader.api.MailboxListener;\
   log = org.osgi.book.logapi.Log
```

The declaration starts with the implementation class name of the component. Further attributes are separated by semicolons, and the first attribute `provide` specifies the service interface. The second attribute `log` uses slightly different syntax (a solitary equals rather than colon and equals) to indicate that it is a service reference. The name of the reference is `log` and the service interface is `org.osgi.book.logapi.Log`. The bind and unbind methods are inferred fromthe simple name of the service interface to be `setLog` and `unsetLog`.

When we run `bnd` on the descriptor in Listing 11.26, it will generate an XML file in the resulting bundle at the path `OSGI-INF/org.osgi.book.ds.mboxlistener.LogMailboxListener.xml`. The contents of this generated XML file are shown in Listing 11.27[11].

[11] One interesting point to note with this XML file is it does not include the namespace that

Listing 11.27 Generated XML for the Logging Mailbox Listener

```
<?xml version='1.0' encoding='utf-8'?>
<component name='org.osgi.book.ds.mboxlistener.LogMailboxListener'>
  <implementation
      class='org.osgi.book.ds.mboxlistener.LogMailboxListener'/>
  <service>
     <provide interface='org.osgi.book.reader.api.MailboxListener'/>
  </service>
  <reference name='log' interface='org.osgi.book.logapi.Log'
       bind='setLog' unbind='unsetLog'/>
</component>
```

The `Log` reference in this example uses the default static policy, and the default cardinality which is `1..1` i.e. unary/mandatory. We can change this by appending a single character annotation to the end of the service interface name. Listing 11.28 is a variant which makes the `Log` reference optional; it also has the effect of switching to the dynamic policy, which is typically preferred with optional references.

Listing 11.28 XML Generation Header for the Optional Logging Mailbox Listener

```
# logmailboxlistener.bnd
Private-Package: org.osgi.book.ds.mboxlistener
Service-Component: org.osgi.book.ds.mboxlistener.OptLogMailboxListener;
   provide:=org.osgi.book.reader.api.MailboxListener;\
   log = org.osgi.book.logapi.Log?
```

The full list of indicator characters is shown in Table 11.1. These indicators do not cover all possible combinations of cardinality and policy – for example, there is no indicator for multiple/static – but they do cover the most sensible combinations. If we need access to more unusual options then we can use long-form attributes, as shown in Listing 11.29. The `multiple`, `optional` and `dynamic` attributes each take a list of service reference names to which they apply.

Listing 11.29 Long Form Cardinality and Policy Attributes in Bnd

```
Service-Component: org.osgi.book.ds.web.MyWebComponent;
   log = org.osgi.book.logapi.Log;\
   http = org.osgi.service.http.HttpService;\
   multiple := log;\
   optional := log, http;\
   dynamic := http
```

we said was required for SCR to treat the declaration according to DS 1.1 rules. In fact `bnd` has detected we are not using any of the 1.1 features, so it omits the namespace, which will allow the component to be used under DS 1.0 as well as 1.1.

Character	Cardinality	Policy
?	0..1	dynamic
*	0..*n*	dynamic
+	1..*n*	dynamic
~	0..1	static
	1..1	static

Table 11.1.: Cardinality and Policy Indicators in Bnd

11.10.2. XML Generation from Java Source Annotations

While the generation approach based on `bnd` headers is undoubtedly more succinct than directly entering XML, the syntax for them is still somewhat fiddly, especially with the need for two different kinds of assignment operators (i.e., `=` and `:=`).

A more natural solution is offered by `bnd` based on generation of XML descriptors from Java 5 Annotations inserted into the Java source code. This has the great advantage that the declarations are located physically alongside the component code that they refer to; and also the positions of the annotations can be used to communicate information – such as bind method names – that would otherwise have to be spelled out explicitly in an XML or `bnd` descriptor. To get a feel for how the annotations work, see Listing 11.30 which reimplements the `LogMailboxListener` class from earlier in the chapter, but with added annotations for `bnd` to process.

The `@Component` annotation indicates that this class defines a component, and it has an attribute to specify the service (or services) provided by the component and a service property. Other attributes exist, for example we could add `immediate="true"` in order to turn off the lazy instantiation feature described in Section 11.5.1.

The second annotation we see here code is `@Reference`, which defines a service reference. In all cases the `service` attribute is compulsory with this annotation, as it defines the service interface to which the reference should bind. We also have some attributes to control the static/dynamic policy, the cardinality and so on. Note that instead of a single cardinality attribute with values "0..1", "1..*n*" etc., we have a pair of boolean attributes named `optional` and `multiple`.

The `@Reference` is always attached to the bind method for a reference, and there are some special rules used by `bnd` to determine which method provides the unbind. By default, the name of the unbind method is presumed to be the name of the bind method prefixed by "un", so in this example the unbind method corresponding to `setLog` is `unsetLog`. There is a special case when the

name of the bind method begins with "add", which is often used with multiple references: the "add" segment of the name is replaced with "remove" to form the unbind method name. So, if we have a bind method named `addListener` then the unbind method name will be `removeListener`. However if neither of these rules suits our needs then we can explicitly indicate the name of the unbind method by adding an `unbind` attribute to the `@Reference` annotation, for example `unbind="deleteFoo"`.

Listing 11.30 LogMailboxListener with Bnd DS Annotations

```
1  package org.osgi.book.ds.annotated;

3  import java.util.concurrent.atomic.AtomicReference;
4  import org.osgi.book.logapi.Log;
5  import org.osgi.book.reader.api.*;

7  import aQute.bnd.annotation.component.*;

9  @Component(provide = MailboxListener.class,
10            properties = { "service.ranking=10" })
11 public class LogMailboxListener implements MailboxListener {

13    private final AtomicReference<Log> logRef
14          = new AtomicReference<Log>();

16    public void messagesArrived(String mboxName, Mailbox mbox,
17                                long[] ids) {
18       Log log = logRef.get();
19       if(log != null) log.log(Log.INFO, ids.length +
20             " message(s) arrived in mailbox " + mboxName, null);
21       else System.err.println("No log available!");
22    }
23    @Reference(service = Log.class, dynamic = true, optional = true)
24    public void setLog(Log log) {
25       logRef.set(log);
26    }
27    public void unsetLog(Log log) {
28       logRef.compareAndSet(log, null);
29    }
30 }
```

To compile this source code we will need visibility of the annotations defined by `bnd`. This can be done by putting the `bnd` JAR itself on the compilation classpath; however in case we do not wish to put the whole of `bnd` on the classpath, the annotation interfaces are available in a separate JAR named `annotation-`*`version`*`.jar`, downloadable from the same location as `bnd`. Just be careful to use matching versions of the annotations and `bnd`. Note that the annotations are not retained at runtime, so it is *not* necessary for your bundles to import the `aQute.bnd.annotation.component` package with `Import-Package`.

Listing 11.31 shows an annotation-based version of the `HeartbeatComponent` seen earlier. This time the `@Component` annotation has no attributes – in particular no `provide`, since the component is not a service. Instead it has `@Activate` and `@Deactivate` annotations to indicate its lifecycle methods.

Listing 11.31 Heartbeat Component with Bnd DS Annotations

```
1  package org.osgi.book.ds.annotated;
2
3  import java.util.concurrent.atomic.AtomicReference;
4  import org.osgi.book.logapi.Log;
5  import aQute.bnd.annotation.component.*;
6
7  @Component
8  public class HeartbeatComponent {
9
10    private Thread thread;
11    private final AtomicReference<Log> logRef
12       = new AtomicReference<Log>();
13
14    @Activate
15    public void start() {
16       thread = new Thread(new Runnable() {
17          public void run() {
18             try {
19                while (!Thread.currentThread().isInterrupted()) {
20                   Thread.sleep(5000);
21                   Log log = logRef.get();
22                   if(log != null)
23                      log.log(Log.INFO, "I'm still here!", null);
24                   else System.err.println("No log available!");
25                }
26             } catch (InterruptedException e) {
27                System.out.println("I'm going now.");
28             }
29          }
30       });
31       thread.start();
32    }
33    @Deactivate
34    public void stop() {
35       thread.interrupt();
36    }
37    @Reference(service=Log.class, dynamic=true, optional=true)
38    public void setLog(Log log) {
39       logRef.set(log);
40    }
41    public void unsetLog(Log log) {
42       logRef.compareAndSet(log, null);
43    }
44 }
```

The rest of the DS examples in this chapter will use these `bnd` source code annotations.

11.10.3. Automatic Service Publication

In Listing 11.30 the `@Component` annotation included a `provide` attribute to indicate that the component should be published under a particular service interface. In fact this was not necessary: when generating the XML descriptor, `bnd` automatically includes a service publication element for any interfaces implemented by the component class. This happens both when generating using the special `bnd` headers, and when generating from Java source annotations.

In this case, `bnd` would have detected that the component class implements the `MailboxListener` interface, and therefore would automatically include the XML necessary to publish under that interface. Therefore it was not necessary to include the `provide` attribute explicitly.

In some cases we don't want to provide a service even when the component class implements an interface. In these cases we need to specify an empty list explicitly, as follows:

```
1 @Component(provide = {})
2 public class NonServiceComponent implements MySecretInterface {
3     // ...
4 }
```

11.11. Configured Components

In the introduction to this chapter, on page 221, it was mentioned that DS components can be configured in a number of ways. Listing 11.32 shows a component that is able to receive configuration data.

The *only* change needed to make a DS component configurable is to add a parameter of type `Map<String,Object>`[12], then simply access the data and use it however we like.

Notice that this code goes to quite some lengths to accept data in a variety of different formats. In particular we accept any of the sub-classes of `java.lang.Number` (i.e. `Integer`, `Long`, `Double` etc.) and in addition any string that is parseable as a Long. Being flexible in the data we accept is a good way to make robust components that are widely reusable. Unfortunately it can make our code somewhat longer than otherwise necessary – especially

[12] In fact the type parameters are erased at runtime, so any form of `Map` can be used, including the raw type. However we know from the DS specification that the map keys will always be of type `String` so it is convenient to add this to our method signature.

Listing 11.32 Configured Component

```
1  package org.osgi.book.ds.annotated;
2
3
4  import java.util.Map;
5
6  import aQute.bnd.annotation.component.*;
7
8  @Component(properties = {"interval=5000"})
9  public class ConfiguredComponent {
10
11    private static final String PROP_INTERVAL = "interval";
12    private static final String MSG_MANDATORY =
13       "The property named '%s' must be present.";
14    private static final String MSG_INVALID =
15       "The property names '%s' is of invalid or unrecognised type.";
16
17    private Thread thread;
18
19    @Activate
20    public void start(Map<String,Object> config) {
21       final long interval = getMandatoryLong(config, PROP_INTERVAL);
22       thread = new Thread(new Runnable() {
23          public void run() {
24             try {
25                while (!Thread.currentThread().isInterrupted()) {
26                   Thread.sleep(interval);
27                   System.out.println("I'm still here!");
28                }
29             } catch (InterruptedException e) {
30                System.out.println("I'm going now.");
31             }
32          }
33       });
34       thread.start();
35    }
36
37    @Deactivate
38    public void stop() {
39       thread.interrupt();
40    }
41
42    static long getMandatoryLong(Map<String,Object> cfg, String key) {
43       Object o = cfg.get(key);
44       if(o == null)
45          throw new IllegalArgumentException(
46                String.format(MSG_MANDATORY, key));
47       else if(o instanceof Number)
48          return ((Number) o).longValue();
49       else if(o instanceof String)
50          return Long.parseLong((String) o);
51       else
52          throw new IllegalArgumentException(
53                String.format(MSG_INVALID, key));
54    }
55 }
```

since, in this example, the `getMandatoryLong` method is used only once! In reality we may wish to pull `getMandatoryLong`, along with other similar methods for different data types, into a library of utility functions.

11.11.1. Sources of Configuration Data

But where does the configuration data actually come from? In fact there are three sources of data, in ascending order of priority.

Component Descriptor Properties

The first source of configuration data is the set of `property` and `properties` elements found in the XML descriptor of the component declaration (or in the `properties` attribute of the `@Component` annotation). We saw the use of the component properties back in Section 11.5.3 to provide service properties; the same set of properties are also considered part of the configuration of the component.

The dual use of properties, i.e. both as part of the published properties of a service and as the internal configuration of the component, is an important design point of DS. Publishing all of the properties that form the configuration of the component enables consumers to filter or select over any of those properties. It's difficult to predict which properties of a component may be useful for a consumer to have visibility of, so DS makes them all visible.

However, specifying configuration properties in the XML descriptor is very limiting, since they must all be fixed at the time that the bundle is built – there is no way to dynamically change them at runtime. Therefore the XML descriptor properties are really only suitable for the *default* configuration of our component.

Configuration Admin

The second source of configuration data is the Configuration Admin (CM) service, discussed at length in Chapter 9.

When using DS we can receive configuration data from Configuration Admin without ever having to implement the `ManagedService` or `ManagedServiceFactory` interfaces. SCR quietly listens for configurations having a Persistent ID (PID) matching the name of our component. If the Configuration Admin service is available, and if a configuration record exists with a matching PID, then our component will receive the data inside that configuration record through its activation method.

When configuration data is available from both Configuration Admin and from the XML descriptor, the Configuration Admin data overrides the XML descriptor properties. The overriding happens on individual properties, not the whole set of properties. This enables us to use the XML descriptor to supply default values for many of the required properties, with Configuration Admin overriding a subset of them at runtime.

Factory Components

The third source of configuration data, overriding both of the previous two sources, is the content of a `Dictionary` that can be passed to a call to `ComponentFactory.newInstance` method of a so-called "factory component".

We have not yet discussed factory components, however they are rarely used, and this source of configuration data is simply not relevant to the standard components we have looked at so far. Therefore we can safely ignore it until we come to factory components later, in Section **??**.

11.11.2. Testing with FileInstall

We will now test the example configured component by feeding it some data using Apache FileInstall, which we first saw in Chapter 9, Section 9.2.2. Recall that FileInstall looks for a file named `<pid>.cfg` in the `load` directory, where `<pid>` is the PID of the configuration record that will be created. As stated above, the PID should match the name of the DS component.

We have so far not explicitly set the names of our DS components, and so they have defaulted to the names of their implementation classes. Therefore the PID we need to create is `org.osgi.book.ds.annotated.ConfiguredComponent`. We can override this default and set the name to whatever we choose by adding a `name` attribute at the top level element of our XML descriptor, or to the `@Component` annotation.

To test configuring the component from Listing 11.32, we need to create a file in the `load` directory named `org.osgi.book.ds.annotated.ConfiguredComponent.cfg` with the following content:

```
interval=10000
```

When this file is created we should notice that the polling interval of the component slows down.

11.11.3. Dealing with Bad Configuration Data

Sometimes the configuration data that a component receives – whether from the XML descriptor or through Configuration Admin – will be "bad". The definition of "bad" data depends on our component, but typically it would include:

- Missing parameters which are required by the component.
- Data values out of range, e.g. `interval=-100`.
- Corrupt or nonsensical data values, e.g. `interval=fish`.

Unfortunately neither Configuration Admin nor Declarative Services provide any validation of the data they supply to the components, therefore it is the responsibility of each component to check the data it receives to see whether it is valid before using it. What should our component do if it finds invalid data? It has two choices; either it can proceed using default values as substitutes for the supplied values, or it can throw an exception.

If we proceed with defaults then it's a good idea to at least write a warning message into a log, so that an administrator might find the warning and correct the invalid data. For this reason it may be useful to include an optional reference to the OSGi Log Service in our component declaration. The other alternative – throwing an exception – will prevent the activation of the component as described in Section 11.7.2.

11.11.4. Dynamically Changing Configuration

As we know from Chapter 9, configuration data can be changed at runtime, even while our application is running and therefore while the components using it are active. Components therefore need to have a way to receive changes to their configuration.

The components we have seen so far had only `activate` and `deactivate` methods. When their configuration data changes, the only way SCR can update them is to deactivate and recreate them, then call `activate` with the new data. This is rather reminiscent of the static policy for service references, and much of the time it is perfectly adequate.

However, sometimes it is not. If a component is expensive to create then it may be wasteful to destroy and recreate it merely to change its configuration. Therefore DS includes the ability to dynamically reconfigure components without recycling them. They key to this is the *Modified Method* that, if it exists, is called by SCR when the configuration of the component changes while the component is active. Just like dynamic service replacement, a component that implements a modified method must be aware that it might be called from an

arbitrary thread even while the configuration data is being used, so thread-safe programming patterns must be used.

Unlike `activate` and `deactivate`, there is no default method name for the modified method; we *must* specify it using the `modified` attribute at the top level of the XML Declaration as follows:

```
<scr:component xmlns:scr="http://www.osgi.org/xmlns/scr/v1.1.0"
               activate="start" modified="modified" deactivate="stop">
   ...
</scr:component>
```

Or if we are using the `bnd` annotations we can simply attach the `@Modified` annotation to the appropriate method. The method itself can take a parameter of type `Map<String,Object>` just like the activate method, and in fact it is perfectly possible to use the same method for both activation and modification if we write it carefully. Listing 11.33 shows a very simple component that allows dynamic reconfiguration using a modified method, which also happens to be the same as the activate method.

Listing 11.33 A Component with Modifiable Configuration

```
package org.osgi.book.ds.annotated;

import java.util.Map;
import java.util.concurrent.atomic.AtomicReference;

import org.eclipse.osgi.framework.console.*;
import org.osgi.service.cm.ConfigurationException;
import aQute.bnd.annotation.component.*;

@Component(provide = CommandProvider.class)
public class ModifiableConfigComponent implements CommandProvider {

   static final String PROP_MSG = "message";
   static final String DEFAULT_MSG = "This is the default message";

   private final AtomicReference<String> messageRef
      = new AtomicReference<String>(DEFAULT_MSG);

   @Activate
   @Modified
   public void start(Map<String, Object> config)
   throws ConfigurationException {
      String message = (String) config.get(PROP_MSG);
      if (message == null)
         messageRef.set(DEFAULT_MSG);
      else
         messageRef.set(message);
   }
   public String getHelp() {
      return "test - print a test message\n";
   }
   public void _test(CommandInterpreter ci) {
      ci.println(messageRef.get());
   }
}
```

An important point to note is the way exceptions are handled. Whereas an ex-

ception thrown from an activate method prevents activation of the component, an exception thrown from a modified method does *not* cause the component to deactivate – though of course it does cause a log entry to be written. Therefore a modified method cannot trigger deactivation when it detects that the new configuration is invalid. We have to either continue using the old configuration or fall back to defaults for the fields that are invalid. If that is not sufficient, i.e. we really need the component to be deactivated, then it is best not to supply a modified method.

11.11.5. Configuration Policies

A problem that was discussed in the Configuration Admin chapter (Section 9.2.3) was what to do when no configuration is available. In many cases we can just use the default values we get from the XML descriptor, but other times there is just no sensible default. As before, consider a mailbox component backed by a database – it would be very dubious to put the database connection parameters into the XML descriptor, they really need to be supplied at runtime via Configuration Admin. If no configuration record exists for our component, it would be best not to activate our component nor register it as a service.

DS provides a much neater way to achieve this goal. Using hand-written `ManagedService` instances it was necessary to perform "split" registration, producing additional complexity. With DS we can simply declare a "policy" for our component with respect to its configuration. This is done with the `configuration-policy` attribute on the top-level element of the XML descriptor, which may be set to the value "`require`":

```
<scr:component xmlns:scr="http://www.osgi.org/xmlns/scr/v1.1.0"
               configuration-policy="require">
   ...
</scr:component>
```

Using the `bnd` annotations this becomes an attribute named `configurationPolicy` on the `@Component` annotation, which takes values defined by the enumeration `ConfigurationPolicy`. For example:

```
import static aQute.bnd.annotation.component.ConfigurationPolicy.*;

@Component(properties = {"interval=5000"},
           configurationPolicy = require)
public class ConfiguredComponent {
   // ...
}
```

Here is the meaning of each of these policies:

Optional (`optional`)

This is the default if no policy is specified.. The component will always be created, and if configuration data with a PID matching the component's name is available, it will be supplied to the component. Otherwise, the XML descriptor data will be supplied.

Require (`require`)

Under this policy, the component will not be "satisfied" unless there exists a configuration record with a PID matching its name. If such a record is missing, or has not been created yet, then the component will not be created or activated, nor will it be registered as a service. If a matching configuration becomes available *later*, then the component will be activated and registered. Lifewise if the configuration is deleted then the component will be unregistered, deactivated and destroyed.

Note that the required configuration policy does not enforce anything about the *content* of the configuration record. The configuration may have missing fields, nonsensical values and so on, but SCR will still use it to activate the component so long as it has a matching PID. Again it is the component's responsibility to deal with any problems in the data itself, and that may include throwing an exception to abort activation.

It may appear that the lack of validation makes the required configuration policy less useful than it could be, or even useless. For example, with the default optional policy, we can detect a missing configuration record in our activate method and throw an exception, as described in Section 11.11.3. However there is an important a difference if the component provides a service. Under the optional configuration policy, the service will always be published, but due to lazy activation the component will not discover until later whether its configuration data is missing (see Section 11.7.2). With the required configuration policy, the service will only be registered when a configuration record exists. Admittedly the configuration may be invalid, meaning the component still needs to throw an exception during activation, but this policy at least handles the very common case of a missing configuration.

Ignore (`ignore`)

This policy means that data from Configuration Admin should simply be ignored. The component will always be created and it will never be supplied with data from Configuration Admin, even if there is a record with a PID matching the component's name. The component will *only* receive configuration data from the XML descriptor.

11.11.6. Example Usage of Required Configuration

Listing 11.34 shows a slightly altered version of the last component that uses the required component configuration. This time a default message is not used, and a modified method is not provided, because we want the component to become inactive when its configuration is either missing or invalid. Upon building this component into a bundle and installing/starting it, we should *not* immediately see a new instance of the `MailboxListener` service published. Only when we create a configuration record with a PID of `org.osgi.book.ds.annotated.RequiredConfigComponent` will we see the service published.

Listing 11.34 A Component with Required Configuration

```
1  package org.osgi.book.ds.annotated;
2
3  import java.util.Map;
4  import java.util.concurrent.atomic.AtomicReference;
5
6  import org.eclipse.osgi.framework.console.*;
7  import org.osgi.service.cm.ConfigurationException;
8  import aQute.bnd.annotation.component.*;
9  import static aQute.bnd.annotation.component.ConfigurationPolicy.*;
10
11 @Component(configurationPolicy = require,
12       provide = CommandProvider.class)
13 public class RequiredConfigComponent implements CommandProvider {
14
15    static final String PROP_MSG = "message";
16
17    private final AtomicReference<String> messageRef
18       = new AtomicReference<String>(null);
19
20    @Activate
21    public void start(Map<String, Object> config)
22          throws ConfigurationException {
23       String message = (String) config.get(PROP_MSG);
24       if(message == null) throw new ConfigurationException(
25             PROP_MSG, "Property is mandatory.");
26       messageRef.set(message);
27    }
28    public String getHelp() {
29       return "test - print a test message\n";
30    }
31    public void _test(CommandInterpreter ci) {
32       ci.println(messageRef.get());
33    }
34 }
```

11.12. Singletons, Factories and Adapters

Figure 11.2 summarises the two forms of cardinality we have seen so far in DS.

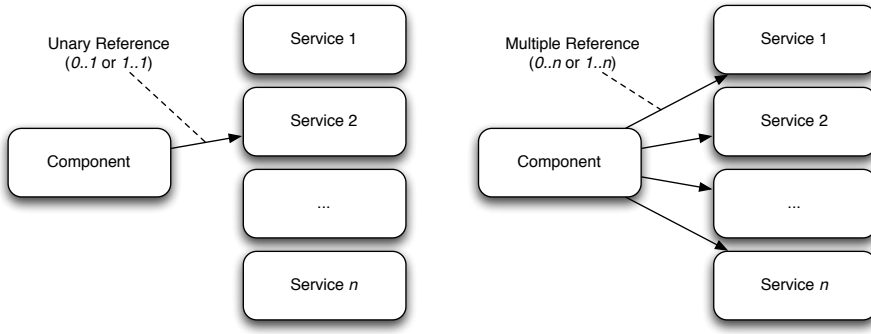


Figure 11.2.: Unary and Multiple Cardinalities in DS

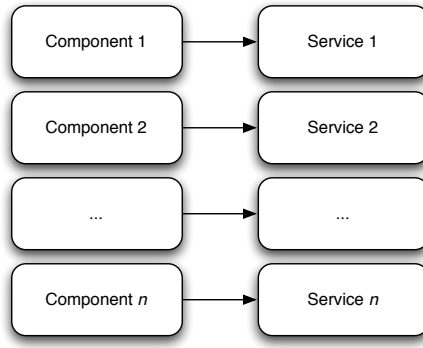


Figure 11.3.: One Component Per Service – Not Supported by DS

In both of these cases, the component on the left of the diagram – the component actually constructed by SCR – is a singleton. The cardinality of the reference controls whether the component binds to all instances of a service or selects one of them, but it does not control the number of instances of the component.

Thus the cardinality shown in Figure 11.3, in which we create an instance of the component for each instance of the service, is not supported by DS. However it is quite easy to support this kind of reference using `ServiceTracker` – Section 4.8 included a classic example in Listing 4.12 on page 93.

Part III.

Practical OSGi

12. Using Third-Party Libraries

Inevitably when developing a Java application, we rely on third party libraries to implement some part of the application or to make our own code simpler or shorter. Typical Java projects tends to have a large collection of JAR files that they build and deploy with, and for the most part this is a good thing: the reliance on external code is the result of Java's huge ecosystem of developers producing useful libraries, both open source and commercial. It's also a result of Java's platform independence, meaning we can simply drop a JAR onto our classpath and use it without worrying about CPU architectures and so on.

However, the uncontrolled use of third party libraries can land us in "JAR Hell", as we have already seen. OSGi offers a solution, but only when it is given properly constructed bundles to manage. In a perfect world, all Java libraries would already be packaged as OSGi bundles with explicit dependencies and exports, and we could simply install them into our framework and start using them. Sadly in the real world only a small fraction of libraries are offered as OSGi bundles, although the number is increasing all the time. To use the remainder, we have to do some extra work.

The two main approaches to using arbitrary are *embedding* the library into a bundle and *wrapping* the library as a bundle. We will shortly see the benefits of each approach and the steps involved. However, the very first step is the same in both cases, and is very important but easily overlooked.

12.1. Step Zero: Don't Use That Library!

The first thing we should do when considering using a third party library is to try *not* to use it. This may seem obtuse, but the serious point is that using a library carries costs as well as benefits, and we should be sure that the benefits outweigh the costs. Unfortunately, whereas the benefits are felt immediately, the costs tend to be spread over time, or may even be paid by somebody else: end users, maintainers, administrators, other developers. It is human nature to be tempted by an immediate gain and to ignore the future cost, even if it is large.

The benefits of using a library are clear enough. By reusing code that's available elsewhere we can get our job done quicker, and end up with more maintainable code. These benefits are approximately proportional to the amount

of code that can avoid writing ourselves by making use of the library: it's a bigger win to eliminate a thousand lines of unnecessary code than ten lines. Also some libraries offer features that are just so hard to implement correctly (e.g., concurrency or encryption utilities) that the majority of programmers should leave it to experts to write the code.

But what are the costs? As soon as we use a library — for example one delivered as a JAR file — we lose the ability to run in environments where that JAR file is not available. We can solve this problem in one of two ways: either we include the library with our application, or we must have a system for finding pre-installed copies of the library on any platform that we deploy to.

The first solution can lead to bloated downloads and a waste of resources, as we may have reduced the amount of source code but we increase the size of our application, sometimes dramatically. Unless the library is very small and focussed, inevitably there will be parts of it that we don't use: those parts add to the size of our application without contributing anything. This option also ignores the fact that there may be a perfectly usable copy of the library already available on the target computer. Users may find that they have many identical copies of common libraries like Apache Log4J as each Java application they use installs its own independent copy[1].

The second solution is clearly more desirable but it is a very difficult technical challenge for which nobody has yet found a good solution. Historically this has been hindered by the non-modular nature of JAR files: if our application finds a copy of a library on a user's computer, how does it know whether it is the correct version? Indeed, how does it even identify it as the correct library, given only an arbitrary file name? And does the library have further dependencies that are not yet present? Obviously OSGi can help this situation a great deal, but it is still not an easy problem to solve. For example, if our application installs a bundle because it was not previously present on the target computer, how does it make sure that bundle is available to other applications that might wish to use it? Solving this calls for the use of standard repositories and deployment practices, but no such standard has achieved widespread acceptance yet.

This situation should improve with time, but however easy OSGi makes the management of dependencies it will always be easy to manage *no* dependency. Therefore it better not to introduce a dependency unless doing it would be a significant benefit.

[1] For example, a quick scan of the author's computer reveals 26 independent copies of Log4J.

12.2. Augmenting the Bundle Classpath

So we have determined that, yes, we really do want to use that third-party library, `foo.jar` from XYZ Corporation from our bundle. The quickest and easiest way to use do this (though not necessarily the best way) is to augment the bundle's internal classpath by using the `Bundle-Classpath` manifest header.

12.2.1. Embedding JARs inside a Bundle

In the introduction chapter we discussed the classpath mechanism used in standard Java to make libraries available to an application. OSGi abolishes the single, global classpath and replaces it with bundles, but each bundle has an "internal" classpath which works just like the global Java one. That is, each bundle has a list of JARs which it reads sequentially when it tries to load a class[2]. The `Bundle-Classpath` header allows us to specify the bundle's internal classpath much like we use the `-classpath` command line parameter to specify the global one.

Note that the location of the JARs referenced by `Bundle-Classpath` cannot be an arbitrary filesystem location, it must be a path to a JAR file embedded inside the bundle JAR. If the value of the header is "`libs/mylib.jar`" then there must exist a JAR file nested inside the bundle JAR at the location `lib/mylib.jar`. It's very important that `Bundle-Classpath` should not be used to refer to JARs at arbitrary external locations because this would expose us to many of the same deployment problems that standard Java suffers from. Bundles must be coherent, self-contained, installable units, having dependencies only on other bundles as expressed through the manifest.

The `Bundle-Classpath` header is specified as a comma-separated list. In a bundle where the value of this header is "`libs/mylibA.jar,libs/mylibB,jar`", a class load request will first try to find the requested class in `mylibA`, then in `mylibB`. Each path is always specified relative to the bundle root, and there is a special path we can add to the list: "." (i.e., dot or period), which simply means the bundle JAR itself. When we don't include "." in the `Bundle-Classpath`, then classes will not be loaded directly from the bundle JAR, which is just a container for the nested JARs.

At runtime, the classes in included JARs are fully-fledged and integral parts of our bundle, there is no difference between classes loaded from nested JARs and classes loaded directly from the bundle JAR. We can even export packages from the nested JARs using the `Export-Package` header.

[2] Assuming that the class could not be imported via `Import-Package` or `Require-Bundle`. Recall that imported packages take precedence over local ones (see Section 3.8).

If we don't include the `Bundle-Classpath` header in our manifest, it takes the default value of just ".". So by default the classpath is equal to the bundle JAR, which is exactly the behaviour we have seen until now.

12.2.2. Problems with Augmenting the Bundle Classpath

Just as the global, flat classpath in standard Java leads to myriad problems — which we characterise as "JAR Hell", overuse of embedded third-party libraries in a bundle can lead to a loss of modularity and "mini-Hells".

Embedding a JAR in our bundle makes the library offered by the JAR available to that bundle... but it does not normally make it available to any *other* bundle. If another bundle wants to use the library as well it may follow the same embedding approach, and now we have two copies of the library. Following this reasoning to its logical conclusion, we might find every bundle in our application embedding its own copies of the same collection of libraries!

The problem is worse than just bloat due to duplication, though that is bad enough. The identity of a Java class is defined by its implementation code and *the classloader that loaded it*, therefore a class that is loaded separately by two different bundles will have two separate identities, even if the embedded JARs are byte-for-byte identical. This means that instances of any such class cannot be communicated between the two bundles, because they are incompatible.

For example, suppose bundles A and B both embed a copy of "Joda Time" library, a popular library for date and time handling. If A exposes an API that accepts an `org.joda.time.LocalDate` object as a method parameter, then bundle B may reasonably expect that it can create an instance of `LocalDate` and pass it to A. However this would cause a `ClassCastException` since the `LocalDate` class has been loaded in separate classloaders.

Notwithstanding the above problems, embedding JARs in our bundles is not *always* the wrong approach. It can be appropriate for small or specialised libraries that are used only in the internal implementation of a bundle. However, if a library is likely to be used by multiple bundles, or if it forms part of the public API of a bundle, then it should be *wrapped*, i.e. turned into a full, standalone bundle.

12.3. Finding OSGi Bundles for Common Libraries

In the above scenario it would have been preferable if Joda Time was available as an OSGi bundle: then bundles A and B could simply have imported the `org.joda.time` package, and would have been able to pass instances of `LocalDate` freely since there would be just one definition of it.

Unfortunately Joda Time is not (yet!) available as an OSGi bundle if we download it from its official web site[3]. But, we can download an OSGi bundle of Joda Time from several alternative sites. In general, if a library is useful to other developers, it's increasingly likely that somebody else has already done the conversion and we can simply reuse the result. Also as projects such as Eclipse and Spring are now using OSGi, they have started to offer large repositories containing many common open source libraries. These repositories should be our first port of call when seeking a "bundleized" copy of a library.

SpringSource Enterprise Bundle Repository [4] is currently the most comprehensive repository. SpringSource, the company behind the popular Spring Framework, offers this repository to assist developers using their OSGi-based application server product. It contains a very large (and growing) number of open source libraries, offered through an advanced web interface.

Eclipse Orbit [5] was created for the use of Eclipse. It is not as comprehensive as SpringSource's repository because only libraries needed by official Eclipse projects are included, and also the licence for each library must be deemed compatible with the Eclipse Public Licence (EPL). Nonetheless Orbit may sometimes contain a library not found elsewhere, so it is useful to know about.

TODO

12.4. Transforming JARs into Bundles, Part I

The repositories listed above do not always help. Maybe we need to use a commercial library which cannot be freely redistributed, or a specialised library that is not popular enough to make it into a general-purpose repository. Or we may need a more recent version of a library than is available from the repositories, which inevitably lag somewhat behind the latest releases from the official web sites. In these cases, we need to perform the transformation to a bundle ourselves.

Fortunately, bnd makes this a very easy process, though it is not — and as we will see, cannot be — fully automatic. We will first look at the process for "OSGifying" a straightforward library, then we will look at some less straightforward examples. Our first example is Joda Time

[3] http://joda-time.sourceforge.net/
[4] http://www.springsource.com/repository
[5] http://www.eclipse.org/orbit/

12.4.1. Step 1: Obtain and Analyse Library

First we need to get hold of the library and work out a few basic pieces of information about it. The latest version of Joda Time is 1.5.2 and it can be downloaded from:

http://downloads.sourceforge.net/joda-time/joda-time-1.5.2.zip

After downloading and unzipping we need to find the main JAR file, which in this case is `joda-time-1.5.2.jar`, and decide on the symbolic name and version we will give to the bundle. It's common practice to use a symbolic name that looks like a Java package name, and usually we choose the highest-level unique package name. This is `org.joda.time`. The version number is clear from the original ZIP file: 1.5.2.

12.4.2. Step 2: Generate and Check

Having decided these values we can write them down in a bnd descriptor named `org.joda.time_1.5.2.bnd` as shown in Listing 12.1

Listing 12.1 Bnd Descriptor for Joda Time

```
# org.joda.time_1.5.2.bnd
-classpath: joda-time-1.5.2.jar
version: 1.5.2

Bundle-SymbolicName: org.joda.time
Bundle-Version: ${version}
Export-Package: *;version=${version}
```

The first line after the comment is simply a convenience which allows us to omit the `-classpath` switch each time we run bnd from the command line. The second line sets an internal identifier for the version of the library: we do this so that if we need to change the version string later, we only need to change it in one place. Note that this identifier will not be included in the final bundle manifest, since it does not begin with an upper case letter.

A blank line serves to separate the preamble from the actual manifest headers where we set the symbolic name, bundle version and a list of exported packages. It's reasonable to assume that all of the packages in the JAR should be exported: the asterisk achieves this, and we just need to add the version attribute to the exports.

Now let's run bnd:

```
$ java -jar /path/to/bnd.jar org.joda.time_1.5.2.bnd
org.joda.time_1.5.2.jar 647 542631
```

Bnd reports success rather tersely by stating the name of the bundle JAR it has generated (`org.joda.time_1.5.2.jar`), the number of entries in the JAR (647, which includes entries for directories as well as files) and the size of the JAR in bytes.

At this stage we should check for certain problems that can crop up in the set of imported and exported packages for the bundle. We could do this by directly examining the bundle manifest, but it can be hard to read. Bnd offers a utility for printing information about bundles, so we can look at the imports and exports by running bnd with the `print -impexp` command:

```
$ java -jar /path/to/bnd.jar print -impexp org.joda.time_1.5.2.jar
[IMPEXP]
Export-Package
  org.joda.time                 {version=1.5.2}
  org.joda.time.base            {version=1.5.2}
  org.joda.time.chrono          {version=1.5.2}
  org.joda.time.convert         {version=1.5.2}
  org.joda.time.field           {version=1.5.2}
  org.joda.time.format          {version=1.5.2}
  org.joda.time.tz              {version=1.5.2}
  org.joda.time.tz.data         {version=1.5.2}
  org.joda.time.tz.data.Africa{version=1.5.2}
  org.joda.time.tz.data.America{version=1.5.2}
  ...
```

In this case bnd has not generated any imports, because Joda Time does not have any dependencies on libraries outside the standard JRE. The exports also look fine, with no unexpected entries. Therefore this bundle is complete: we have successfully "OSGified" Joda Time.

12.5. Transforming JARS into Bundles, Part II

Now we will look at a slightly trickier example: HSQLDB [**?**] is an embeddable SQL relational database engine written in pure Java. It can be used for file-based storage in a standalone application, or as a traditional database server. In either mode it offers JDBC support. The download link is as follows:

http://downloads.sourceforge.net/hsqldb/hsqldb_1_8_0_9.zip

After unzippping, the main JAR is at `lib/hsqldb.jar` and we choose the symbolic name `org.hsqldb`. But what about the version? Here we hit the first small problem: HSQLDB uses four numeric parts in its version number, whereas OSGi allows only three numeric plus an alphanumeric part. We need to map the final segment to an alphanumeric string, bearing in mind the warnings of Section **??**. Therefore we add a leading zero to the final segment to get a version number of `1.8.0.09`. The resulting bnd descriptor is shown in Listing 12.2. Notice that this follows exactly the same pattern as the Joda Time descriptor; in fact our first pass bnd descriptor always looks like this.

Listing 12.2 Bnd Descriptor for HSQLDB, First Pass

```
# org.hsqldb_1.8.0.09.bnd (1st Pass)
-classpath: lib/hsqldb.jar
version: 1.8.0.09

Bundle-SymbolicName: org.hsqldb
Bundle-Version: ${version}
Export-Package: *;version=${version}
```

Unfortunately when we run bnd against this descriptor we hit another problem: it prints an error and fails to generate the bundle JAR[6].

```
$ java -jar /path/to/bnd.jar org.hsqldb_1.8.0.09.bnd
One error
1 : Unresolved references to [.] by class(es) on the Bundle-
Classpath[Jar:dot]: [hsqlServlet.class]
```

The error message is somewhat cryptic but observing the reported class name of `hsqlServlet.class` and taking a look in the original JAR reveals the source of the problem: this class is in the default package (i.e., it does not have a package name), which means it cannot be exported. This is probably a mistake by the authors of HSQLDB, or at least a questionable decision, since putting classes in the default package is very bad practice for a library. Fortunately it is almost certainly safe to simply omit this class, which we can do by refining our export statement as follows:

```
Export-Package: org.hsqldb*;version=${version}
```

If we run `bnd` again it will now generate a bundle JAR:

```
$ java -jar /path/to/bnd.jar org.hsqldb_1.8.0.09.bnd
org.hsqldb_1.8.0.09.jar 356 706011
```

As before we should check the generated imports and exports. The result of calling bnd with the `print -impexp` command is shown in Listing 12.3.

12.5.1. Step 3: Correcting Imports

The set of imports detected by bnd tells a story about the composition of HSQLDB. Notice the presence of dependencies on both Swing and the HTTP Servlet APIs, indicating that this JAR contains a mish-mash of both GUI and server side functionality. Sadly this is quite typical of many Java libraries.

We know that HSQLDB can be used in several contexts. Clearly there is some GUI code in this JAR, along with some usage of the Servlet API, but we also know that HSQLDB can be embedded in standalone processes, including ones

[6] Earlier versions of bnd may treat this problem as merely a warning and still build the bundle JAR, but we should deal with the problem anyway.

Listing 12.3 Generated Imports and Exports for HSQLDB, First Pass

```
$ java -jar /path/to/bnd print -impexp org.hsqldb_1.8.0.09.jar
[IMPEXP]
Import-Package
  javax.naming
  javax.naming.spi
  javax.net
  javax.net.ssl
  javax.security.cert
  javax.servlet
  javax.servlet.http
  javax.sql
  javax.swing
  javax.swing.border
  javax.swing.event
  javax.swing.table
  javax.swing.tree
  sun.security.action
Export-Package
  org.hsqldb                    {version=1.8.0.09}
  org.hsqldb.index              {version=1.8.0.09}
  ...
```

that are "headless", i.e. without a GUI. So neither the Swing nor the Servlet dependencies seem to be core to the functionality of HSQLDB.

The Servlet API dependency is the biggest problem, because it means our HSQLDB bundle will not resolve unless it can be wired to an exporter of the `javax.servlet` and `javax.servlet.http` packages. These are not part of the standard JRE libraries, so they would have to be offered by another bundle. It would be a shame to prevent access to HSQLDB entirely just because one part of it may not be available.

On the other hand, we can't simply remove the dependencies, because then the bundle would not be able to load classes from the `javax.servlet` and `javax.servlet.http` packages at all, even when they are available! Therefore the best thing to do is to make the imports optional, so that HSQLDB can use the Servlet API when an exporter can be found, but will not be prevented from resolving if no exporter is present. We do this by adding an explicit `Import-Package` instruction to the bnd descriptor:

```
Import-Package: javax.servlet*;resolution:=optional, *
```

Notice the "*" at the end of the instruction, which acts as a catch-all. Without it, we would import *only* the `javax.servlet` and `javax.servlet.http` packages.

There is some risk inherent in marking an import as optional. By doing so we are asserting that the bundle will still work in the absence of the import... but bnd has detected that *something* in the bundle uses it, so is our assertion really safe? It depends on what we mean by the bundle "still working". To answer in full we would have to examine all possible use-cases of HSQLDB,

following the code paths through the bundle to see if they touch the optional imports. If a use-case touches one of the optional imports, it will fail with a `NoClassDefFoundError` when the package it not present, but as long as the majority of use-cases (and in particular the ones we consider to be "core" for the library) do not fail then the bundle is still useful[7].

Unfortunately a full examination of all the use-cases tends to be far too much work, or even impossible if the library is closed-source. So we usually take a judgement call instead: HTTP servlets do not seem to be core to the functioning of HSQLDB, so we make those imports optional.

How about the Swing package dependencies? These are less of a problem since Java Standard Edition always includes Swing in the base JRE library. Still, there are Java editions and Java-like platforms that do not include Swing, such as Java Micro Edition or Google Android, so it is worthwhile marking these imports as optional also[8].

Most of the remaining listed packages are parts of the JRE base library, and do not need to be marked optional. But there is one package on the list that is quite troubling: `sun.security.action`. This package is part of the Sun JRE implementation, but is *not* part of the publicly specified JRE libraries. Sun strongly discourages developers from using these packages as they are not present in other JRE implementations such as IBM's J9 or BEA/Oracle's JRockit, and they are not even guaranteed to stay the same across different versions of Sun's own JRE. If we leave this dependency in our bundle, it be restricted to running on the Sun JRE.

We need to do some detective work to see whether the dependency can be safely made optional. It seems likely that it can, because nowhere on the HSQLDB web site or documentation is it stated that HSQLDB only runs on the Sun JRE! The first step is to look at which packages of HSQLDB make use of the import. The `print -uses` command will tell us. Listing 12.4 shows the (abridged) output, from which we can see that the offending package is only used by the `org.hsqldb.util` package.

To narrow it down further, we need to look at the source code, which fortunately we have available since HSQLDB is open source. Searching inside the `org.hsqldb.util` package reveals that the class `GetPropertyAction` from `sun.security.action` is used by `ConnectionDialogCommon` and `DatabaseManagerSwing`. Clearly these are GUI classes, and we have already made the Swing dependency optional, so it makes no difference to mark the `sun.security.action` dependency as optional also. Furthermore if we read

[7] In some cases this analysis can reveal that the imports are not used at all! This tends to happen when some part of a library has become obsolete and is no longer reachable from the public API, but it has not yet been removed.

[8] It would take a more thorough analysis to see whether HSQLDB actually does run on Java ME or Android.

the code for these classes, `GetPropertyAction` is actually loaded using reflection, and there is proper error handling in place for when the class cannot be loaded. These utility classes have been carefully coded to use the Sun JRE class when possible, but not to fail when running on other JREs. Our conclusion is that is *is* safe to mark the import as optional. Listing 12.5 shows the final version of the bnd descriptor for HSQLDB.

Listing 12.4 Package Uses for HSQLDB, First Pass (Abridged)

```
$ java -jar /path/to/bnd.jar print -uses org.hsqldb_1.8.0.10.jar
[USES]
org.hsqldb                                  java.sql
                                            javax.net
                                            ...
org.hsqldb.store                            java.sql
                                            org.hsqldb.lib
...
org.hsqldb.types                            org.hsqldb
                                            org.hsqldb.lib
org.hsqldb.util                             java.sql
                                            javax.swing
                                            ...
                                            sun.security.action
```

Listing 12.5 Bnd Descriptor for HSQLDB, Final Version

```
# org.hsqldb_1.8.0.09.bnd (Final Version)
-classpath: lib/hsqldb.jar
version: 1.8.0.09

Bundle-SymbolicName: org.hsqldb
Bundle-Version: ${version}
Export-Package: org.hsqldb*;version=${version}
Import-Package: javax.servlet*;resolution:=optional,\
    javax.swing*;resolution:=optional,\
    sun.security.action;resolution:=optional,\
    *
```

12.5.2. Step 4: Submit to Repository

Sadly all of the work we have just done to "bundleize" HSQLDB was unnecessary, since HSQLDB can already be found in the SpringSource repository, amongst others. The preceding section was intended merely as an illustration of the process.

Still, if we were really the first to bundleize HSQLDB then ideally we should also be the last! We can now publish the bundle to a repository so that other developers need not repeat our efforts. All of the public repositories have a process for submitting new bundles.

It is also a good idea to maintain your own repository. Companies and organisation working with OSGi should consider creating a company-wide bundle

repository into which developers across the organisation can contribute bundles. This is particularly useful for commercially licensed libraries that can never appear in the public repositories but can be legally shared within a company, assuming the licence permits doing so. This can also be done at a project level if OSGi is not yet used in other projects, and even individual developers can benefit from keeping bundleized libraries in a safe place so they can be reused later.

12.6. Runtime Issues

You may have noticed that in our efforts so far to bundleize two libraries, we have not actually *run* either of them! Bnd is a powerful static analysis tool, and in many cases the metadata produced is accurate. Nevertheless we should always check the resulting bundles by running them, as this can reveal certain kinds of problem that are simply not accessible to any static analysis tool. These problems fall into two main categories: reflection-based dependencies; and classloader shenanigans.

12.6.1. Reflection-Based Dependencies

Bnd searches for dependencies by analysing the compiled bytecode of every Java class that will appear in a bundle. In this way it can find all of the static or "compiled-in" dependencies. For example when looking at the class A it may find that one of the methods of A invokes a method on class B: therefore A depends on B. Or, perhaps a method of A takes an instance of B as a method parameter, or returns a B. Or, A may be a subclass of B, and so on. All of these kinds of dependency are easily discovered by bnd and used to calculate the `Import-Package` header.

However, sometimes a dependency exists that is not visible to bnd, and this usually happens when the Reflection API is used to load classes by name at runtime.

In fact, bnd is still able to discover some of these dependencies too. Recall that it found a dependency on `sun.security.action.GetPropertyAction` in HSQLDB, which appears in the Java source as follows:

```
1 Class c = Class.forName("sun.security.action.GetPropertyAction");
2 Constructor constructor = c.getConstructor(...);
```

This worked because the parameter passed to `Class.forName` was a hard-coded constant. But it may have been a variable, and the value of that variable may have been initialised from a configuration file or even text entry by the user, so it's impossible to know until runtime what class should be loaded. Therefore bnd cannot generate an import.

This presents a problem for OSGi. Bundle classloaders can only load classes that are explicitly imported with `Import-Package` or `Require-Bundle`, or are present on the bundle's internal classpath. If a library tries to load a class `org.foo.Bar` and the package `org.foo` is not available then `Class.forName` will throw a `ClassNotFoundException`. This is the case even if some bundle exports `org.foo`.

Worse of all, we can only find such errors through testing the code at runtime. This sounds almost as bad as normal Java, where `ClassNotFoundException`s and `NoClassDefFoundError`s can crop up at any time. But of course it is not really that bad at all — reflection-based dependencies are very much rarer than normal static ones. Let's look at how to deal with them.

12.6.2. Hidden Static Dependencies

Some code uses reflection simply to "hide" an otherwise normal static dependency. Why would anybody wish to hide a dependency? HSQLDB has already given us a perfect example: by loading the `GetPropertyAction` with reflection, the code still works on non-Sun JVMs that do not have the class, yet it can take advantage of the class when it is available.

Another use-case is hiding a dependency from the compiler. Using reflection, a we can access features of a library at runtime even if we do not have access to that library at compile time.

Once we have found such a hidden static dependencies they are easy enough to deal with. We simply add the relevant package to our `Import-Package` header. Whether we make it optional or not depends on whether we think that it is core to the functionality of the library.

12.6.3. Dynamic Dependencies

TODO

12.7. ClassLoader Shenanigans

Some libraries go well beyond simple reflection and dive deep into Java's flexible classloading architecture. Although there is nothing wrong with this as such, it needs to be done with great care. As we saw with dynamic dependencies, there are some common assumptions made by libraries that are fine in standard Java but do not hold true in more complex environments such as OSGi or J2EE.

An example of a widespread library that gets classloading dramatically wrong is Apache Commons Logging. It fails quite disastrously when run under OSGi. The reason for this is ironic: Commons Logging attempts to compensate for Java's weak modularity and extensibility support by searching the "classpath" for a concrete logging implementation it can use.

The idea behind Commons Logging is reasonable: many libraries need to generate log messages, but they don't know which logging framework will be used by the applications they are embedded in. If a library links with any concrete logging framework (e.g., Log4J), then it can be difficult to use the library in an application that uses any other framework. So, Commons Logging offers an abstraction for library code to link against, and it binds dynamically to a concrete logging framework at runtime. It is this dynamic search that fails under OSGi, since it assumes global visibility of logging classes on the "classpath". It would be possible to use a straightforward static binding, if Commons Logging were available as multiple JARs, each bound to a single concrete logging implementation. But Commons Logging opts for a single JAR as it is safer and more manageable under standard Java.

In general, libraries that use "classloader shenanigans" should be avoided by OSGi developers. Sadly it is not always so easy, since they may be used by other libraries required by our application. It is best in these cases to find alternative libraries. In the case of Commons Logging, which is a particular problem since it is so widespread, the SLF4J (Simple Logging Framework for Java) [**?**] project offers an API-compatible replacement, and is already available as an OSGi bundle.

13. Testing OSGi Bundles

TODO

14. Building Web Applications

TODO

Part IV.

Appendices

A. ANT Build System for Bnd

The following is suggested project structure for building OSGi projects based on ANT and bnd. The project structure is assumed to be as in Figure A.1. The purpose of these directories is as follows:

src contains the Java source of our bundles, laid out in the normal Java way with subdirectories for each package.

test contains JUnit-based tests for our Java source, also laid out in package subdirectories.

bundles contains binary or pre-built bundles as JARs that form the dependencies of our code. We will need these at compile time as well as runtime. For example we may include `osgi.cmpn.jar`, which is a bundle that contains the API (but *not* implementation!) of all the OSGi Compendium services.

All of the bnd descriptor files are placed at the top level of the project[1]. This is also where we place `build.xml` and a supplementary properties file called `build.properties`. The latter file is shown in Listing A.1; the settings shown will certainly need to be changed to match your own computer.

Listing A.1 `build.properties`

```
1 # Path to the Felix installation directory
2 felix.home=/path/to/felix-1.0.3

4 # Location of the JUnit JAR
5 junit.path=/path/to/junit/junit-4.4.jar

7 #Location of bnd.jar
8 bnd.path=/path/to/bnd/bnd.jar
```

[1] This could be changed to a subdirectory by editing the `bundle` target of the ANT build.

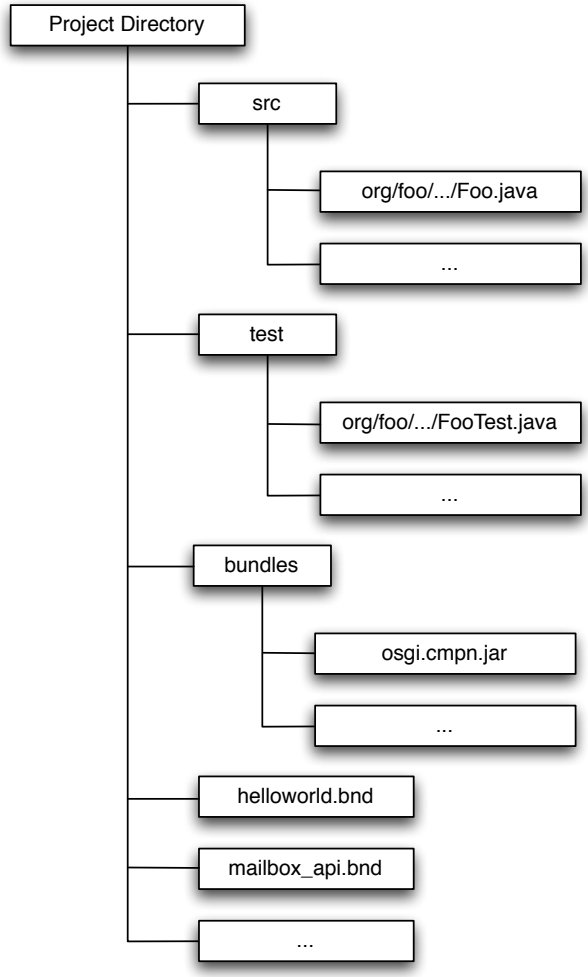


Figure A.1.: OSGi Project Structure

Listing A.2 `build.xml`

```
<?xml version="1.0" encoding="UTF-8"?>
<project name="osgibook" default="bundle">

    <!-- Import machine-specific settings -->
    <property file="build.properties"/>

    <!-- Setup build paths -->
    <property name="build_dir" value="build"/>
    <property name="build_classes_dir" value="${build_dir}/classes"/>
    <property name="build_bundles_dir" value="${build_dir}/bundles"/>
    <property name="build_test_dir" value="${build_dir}/tests"/>

    <!-- Set a classpath for the OSGi libraries -->
    <path id="osgilibs">
        <pathelement location="${felix.home}/bin/felix.jar"/>
        <fileset dir="bundles" includes="*.jar"/>
    </path>

    <!-- Set a classpath for JUnit tests -->
    <path id="test_classpath">
        <path refid="osgilibs"/>
        <pathelement location="${junit.path}"/>
    </path>

    <!-- Load the bnd custom task -->
    <taskdef resource="aQute/bnd/ant/taskdef.properties"
             classpath="${bnd.path}"/>

    <!-- TARGET: clean; cleans all build outputs -->
    <target name="clean" description="Clean all build outputs">
        <delete dir="${build_dir}"/>
    </target>

    <!-- TARGET: compile; compiles Java sources -->
    <target name="compile" description="Compile Java sources">
        <mkdir dir="${build_classes_dir}"/>
        <javac srcdir="src" destdir="${build_classes_dir}"
               debug="true" classpathref="osgilibs"/>

        <mkdir dir="${build_test_dir}"/>
        <javac srcdir="test" destdir="${build_test_dir}"
               debug="true" classpathref="test_classpath"/>
    </target>

    <!-- TARGET: bundle; generates bundle JARs using bnd -->
    <target name="bundle" depends="compile"
            description="Build bundles">
        <mkdir dir="${build_bundles_dir}"/>
        <!-- Convert an ANT fileset to a flat list of files -->
        <pathconvert property="bnd.files" pathsep=", ">
            <fileset dir="${basedir}">
                <include name="*.bnd"/>
            </fileset>
        </pathconvert>
        <bnd classpath="${build_classes_dir}" failok="false"
            output="${build_bundles_dir}" files="${bnd.files}"/>
    </target>
</project>
```

Bibliography

[1] Boeing 747 Fun Facts. http://www.boeing.com/commercial/747family/pf/pf_facts.html.

[2] Java JAR File Specification. http://java.sun.com/j2se/1.4.2/docs/guide/jar/jar.html.

[3] Apache Jakarta Commons HttpClient. http://jakarta.apache.org/httpcomponents/httpclient-3.x/.

[4] Eclipse Equinox. http://www.eclipse.org/equinox.

[5] Eclipse. Eclipse Public License 1.0. http://opensource.org/licenses/eclipse-1.0.php.

[6] Knopflerfish OSGi. http://www.knopflerfish.org/.

[7] Apache Felix. http://felix.apache.org/site/index.html.

[8] Concierge OSGi. http://concierge.sourceforge.net/.

[9] JSR 277: Java Module System. http://www.jcp.org/en/jsr/detail?id=277.

[10] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Boston, MA, USA, 1995.