Spring Dynamic Modules IN ACTION

Arnaud Cogoluègnes
Thierry Templier
Andy Piper

FOREWORD BY PETER KRIENS

SAMPLE CHAPTER

Spring Dynamic Modules in Action

by Arnaud Cogoluègnes,
Thierry Templier,
and Andy Piper

Chapter 6

Copyright 2011 Manning Publications

brief contents

PART 1 SPRING DM BASICS 1

1 ■ Modular development with Spring and OSGi 3
2 ■ Understanding OSGi technology 24
3 ■ Getting started with Spring DM 63

PART 2 CORE SPRING DM 101

4 ■ Using Spring DM extenders 103
5 ■ Working with OSGi services 133
6 ■ OSGi and Spring DM for enterprise applications 164
7 ■ Data access in OSGi with Spring DM 199
8 ■ Developing OSGi web components with Spring DM and web frameworks 236

PART 3 ADVANCED TOPICS 281

9 ■ Advanced concepts 283
10 ■ Testing with Spring DM 323
11 ■ Support for OSGi compendium services 351
12 ■ The Blueprint specification 374

6 OSGi and Spring DM for enterprise applications

This chapter covers

- Using the traditional Java EE framework in OSGi environments
- Creating OSGi bundles from existing Java artifacts
- Designing OSGi-based enterprise applications
- Handling OSGi's dynamic nature

You saw in the previous chapter that OSGi lets its modules communicate only by way of services. This helps decouple them and promotes a more modular programming model than in standard Java. Modularity is good for applications, but, as enterprise application developers, we've become negligent when developing enterprise applications in the last few years. These applications grew big and monolithic, did not have particularly strict dependency management, and sometimes used Java introspection or classloaders in fancy ways. Now that we have discovered OSGi and want to build our enterprise applications on top of this wonderful platform, we need to eliminate these bad habits. There is no place for approximation in OSGi.

Don't feel guilty or desperate: OSGi is a welcoming world, even for enterprise application developers. Throughout this chapter, we'll show you how to adapt your development to OSGi by choosing good frameworks and libraries that are OSGi-compliant, by getting existing Java artifacts ready for use in OSGi environments, and by designing your own applications to leverage the features of OSGi. This may look like a tortuous path, but you'll be surprised at how much OSGi has already become part of day-to-day development work. You may well discover that you've been using OSGi-compliant frameworks for months without knowing it.

In this chapter, we'll guide you along the OSGi path. We'll start by showing you that you can still use your favorite libraries and frameworks in OSGi: some of them are already OSGi bundles, and you'll learn how to make the others compatible. Because OSGi brings a new modularity paradigm to Java, we'll also show you how to leverage it and design applications with OSGi, backed up by Spring DM.

This may seem off topic in a Spring DM book, but it will show you that introducing OSGi in enterprise applications isn't difficult. Spring DM will be the bridge between your applications and the OSGi runtime; we'll discuss this in section 6.3, which provides guidelines about application design with OSGi. Spring DM can help you follow and implement these guidelines. You'll learn how Spring DM can assemble and communicate with your OSGi bundles easily, and how it can help you handle the powerful but tricky dynamic aspect of OSGi.

Let's start by looking at how traditional Java libraries and frameworks react within an OSGi environment.

6.1 Building an OSGi repository for enterprise applications

The deployment unit in OSGi is the bundle, which is a standard JAR file, enhanced with metadata that (among other things) informs the OSGi platform of the bundle's dependencies and what it can provide to other bundles in terms of Java packages. Having all of your libraries, frameworks, modules, and applications packaged as bundles is essential for successfully using OSGi.

In sections 6.1.1 and 6.1.2, we'll look at how to use these kinds of Java frameworks and libraries, and in section 6.1.3 we'll see how to get them easily from repositories dedicated to OSGi. Note that this section isn't specific to Spring DM; the information it covers is valid for any OSGi-based application.

6.1.1 Using Java and Java EE frameworks in OSGi environments

As a developer of enterprise applications, you'll know that you never start a new project from scratch. You know you can rely on your pet frameworks, which relieve some of the recurrent technical concerns. Indeed, that's what enterprise application development is all about—not reinventing the wheel, and reusing existing code as much as possible. But in developing OSGi applications, you'll soon notice that not all Java and Java EE libraries or frameworks are packaged as OSGi bundles. Even worse, some aren't OSGi-friendly in their use and execution.

Fortunately, some projects are aware of the growing popularity of OSGi in Java enterprise middleware and applications, so becoming OSGi-compliant, from their design to their packaging, became one of their priorities. Don't abandon OSGi because you're afraid you'll have to start your project from scratch. There are a lot of enterprise frameworks and libraries that work in OSGi environments. If you decide to develop OSGi applications, the biggest changes will be in the structure of your applications rather than in the frameworks you use.

If you're lucky, your favorite frameworks and libraries will work out of the box. If you aren't so lucky, you'll have to make their packaging OSGi-aware. We'll cover both cases, starting by describing what's known to work in an OSGi environment.

6.1.2 Choosing the right frameworks for OSGi

So you're an enterprise application veteran and you want to try out OSGi? Or you're an old hand at OSGi and are eager to exercise your skills in large-scale enterprise applications? In any case, you'll have to make both the OSGi and enterprise-application worlds work together, and you know that some Java libraries and frameworks are more suited to OSGi than others. For example, Jakarta Commons Logging (JCL)is known to be OSGi-unfriendly because of its dynamic discovery process (see the sidebar for more details). Some functions like dynamic loading of classes are sensitive within OSGi, so you should ensure that your favorite libraries and frameworks handle them in a safe and reliable way before using them in an OSGi environment.

Jakarta Commons Logging and OSGi

JCL is probably the most popular logging facade, but despite its large adoption, JCL is very OSGi-unfriendly. How can a good library become a bad egg in OSGi?

JCL is a thin wrapper around several logging implementations, Log4j being the most popular. This means you can use the JCL API in your applications and simply plug in your favorite implementation, as long as it's supported.

JCL initializes itself when the first call to the logging system is made. This initialization consists of dynamically finding which implementations are available on the classpath, choosing one, and redirecting all subsequent calls to it. In theory, this sounds simple: you drop JCL and Log4j JAR files into your applications and the latter will be used automatically in most cases. If you're unlucky, you won't get any log messages and will fight for hours trying to diagnose cryptic classloader issues.

JCL's discovery process is dynamic and relies on the use of the TCCL. Corresponding JCL implementation classes (such as `Log4jLogger`) are also loaded by the TCCL, but when it comes to instantiating one of these logging objects, JCL uses the *current* classloader, which doesn't always see the same implementation classes (because it can be different from the TCCL). This dynamic discovery process can be problematic in some servlet containers, and it makes the use of JCL in an OSGi environment very difficult, if not impossible.

(continued)

To learn more about the pitfalls of JCL's discovery process, you should read the article, "Taxonomy of class loader problems encountered when using Jakarta Commons Logging" (http://www.qos.ch/logging/classloader.jsp), by Ceki Gülcü, the founder of Log4j, SLF4J, and Logback.

You must now be wondering how Spring DM and the Spring Framework both use the Jakarta Commons Logging API. Recall that we also deploy SLF4J bundles when we use Spring DM. SLF4J is another logging facade, which strives to address JCL's pitfalls by using a static discovery process. Using SLF4J is quite similar to using JCL: you use its API, and you drop into its classpath the API's JAR, the JAR of one (and only one!) of its bindings (the bridge between SLF4J and the target logging framework), and the JAR of the logging implementations. Unlike JCL, SLF4J's discovery process is static: the SLF4J API just expects a binding class, `StaticLoggerBinder`, which is made available by the sole SLF4J binding JAR that you generally should provide on the classpath.

But still, Spring and Spring DM use Jakarta Commons Logging! Yes, they do, and that's why we also deploy a special JCL bundle, which is a library provided by SLF4J. It defines the exact same API as JCL but it's backed up by SLF4J. This library has the appearance and smell of JCL, but it's actually SLF4J. That's the trick for making Spring and OSGi happy about logging.

To help you find appropriate libraries and frameworks, table 6.1 offers a nonexhaustive list of those that are known to be OSGi-compliant.

Table 6.1 OSGi-compliant enterprise frameworks and libraries

Name	Type	Note
Spring Framework	Lightweight container and dependency-injection framework, enterprise support	The Spring Framework binaries have been packaged as OSGi bundles since version 2.5. For use in OSGi, prefer the "A" versions.
Spring Portfolio projects	Miscellaneous (security, web, batch, integration, ...)	All binaries are OSGi bundles, and most of the projects have been tested in OSGi environments.
Google Guice	Lightweight dependency-injection framework	Distributed as OSGi bundles since version 2.0.
Groovy	Java-based dynamic language	Distributed as an OSGi bundle.
Jetty	Web container	Distributed as an OSGi bundle and used as an implementation of OSGi's HTTP service.
Apache Commons	Reusable Java components	Most of the projects are OSGi-compliant thanks to the Felix Commons effort.
EclipseLink	ORM	Distributed as OSGi bundles.
OpenEJB	EJB 3.0 container	Distributed as OSGi bundles and used in several application servers.

Table 6.1 **OSGi-compliant enterprise frameworks and libraries *(continued)***

Name	Type	Note
SLF4J	Logging facade	Distributed as OSGi bundles and tested in OSGi environments.
Logback	Logging implementation	OSGi-compliant; intended to be successor to Log4j.
Wicket	Web framework	Wicket binaries have been packaged as OSGi bundles since version 1.4.
MINA	NIO framework	MINA binaries have been packaged as OSGi bundles since version 2.0. Apache server-based projects use MINA for their NIO layer.
H2	Pure Java database engine	Distributed as an OSGi bundle.

In the next section, we'll introduce you to several repositories where you can download ready-to-use OSGi bundles.

6.1.3 Getting OSGi-ready artifacts

We saw that some projects distribute their binaries as OSGi bundles. If one of your dependencies happens to not be a part of these projects, there's still a small chance you won't end up wrapping it yourself, because there are some projects targeted at making OSGi bundles available. Here is a list of some of these OSGi repositories:

- *OSGi Bundle Repository* (http://www.osgi.org/Repository/HomePage) Maintained by the OSGi Alliance, this repository hosts more OSGi-centric bundles than OSGi-ified versions of enterprise frameworks. You can search bundles by keyword or category and get precise information from the web interface. The format of the repository follows a standard described in the "OSGi RFC 112 Bundle Repository," making the repository usable remotely by any OSGi container.
- *Apache Felix Commons* (http://felix.apache.org/site/apache-felix-commons.html) This isn't exactly a repository, but a community effort to popularize the distribution of Java projects as OSGi bundles. Some volunteers OSGi-ify standard Java artifacts and make them available, hoping original developers will then include the OSGi-ification process in the build of their frameworks and libraries. Contributions include most of the Apache Commons projects, ANTLR, and cglib.
- *Eclipse Orbit* (http://www.eclipse.org/orbit/) This repository includes bundles that have been used and approved in one or more projects from the Eclipse Foundation. These bundles can contain some Equinox-specific metadata because they're meant to be used with this particular OSGi container.
- *SpringSource Enterprise Bundle Repository* (http://www.springsource.com/repository/app/) This repository hosts hundreds of open source enterprise libraries, usually OSGi-ified by SpringSource employees. It features a search engine and precise information about bundles. Artifacts are made available for use with Maven 2 and Ivy.

You should find what you need from among these repositories. But if you don't find a library, you'll have to do the dirty work yourself, and in the next section we'll describe techniques that make this relatively painless.

6.2 OSGi-ifying libraries and frameworks

Before diving into the design of enterprise applications with OSGi and Spring DM (which we'll do in section 6.3), we need to have all our dependencies be OSGi-compliant. Developing applications for an OSGi environment should not prevent you from using your favorite Java and Java EE libraries and frameworks. More and more libraries are now OSGi-friendly, because packaging them as an OSGi bundle is part of their build; but bad things happen, and perhaps one day you'll find that your best-loved Java framework is packaged as a normal JAR and is absolutely useless in your OSGi application.

This isn't a desperate situation. You're about to learn everything you need to know about transforming a non-OSGi JAR file into a 100 percent OSGi-compliant bundle, a process that we decided to qualify with the barbarism "OSGi-ification" for brevity. We'll start with a little bit of theory, and then we'll dive into the transformation. We'll first try to do it by hand and then use tools like Bnd. We'll do our experimenting on the Apache Commons DBCP library.

6.2.1 How to create OSGi-ified versions of libraries

The main issue in the OSGi-ification of an existing library is visibility. From the library's point of view, it means being able to see external dependencies but also making its own classes visible to other bundles if necessary. You may have figured out that we'll have to juggle the `Import-Package` and `Export-Package` manifest headers.

When a library is built upon other libraries, it uses their classes and imports some of their packages into its own classes. In a standard Java environment, you can add these libraries on the classpath, and any class can import their packages and use their classes. The story is different in an OSGi environment: libraries must explicitly *export* the packages they want to share, and modules that want to use them must explicitly *import* these packages. The whole export/import process is managed by the OSGi platform with metadata contained in the bundle's manifest file.

IMPORTING PACKAGES

Let's talk first about the process of importing: a library needs to use some classes defined in another library (we'll assume the other library properly exports these classes, making them visible to other bundles). As an example, consider the ORM module in the Spring Framework: this module includes support for popular ORM tools such as Hibernate, iBATIS, and OpenJPA. If we focus on Hibernate, the `Import-Package` of the ORM module might look like the following:

```
Import-Package: org.hibernate,org.hibernate.cache,org.hibernate.cfg
(...)
```

Hibernate has a lot of packages, and Spring ORM uses most of them, so we didn't include the whole list.

The previous snippet is fine regarding what Spring ORM can see (some of Hibernate's packages) but it isn't precise enough regarding *versions*. In its 2.5.6.A version, Spring ORM's Hibernate support is only tested against Hibernate 3.2, so this should appear in the manifest. The `Import-Package` header can use the `version` attribute to specify the exact version or version range the bundle needs. This attribute defaults to the range [0.0.0, ?), and because we didn't use the `version` attribute in our first manifest declaration, the ORM module would use any available version installed in the OSGi container. This could make a 3.0 Hibernate bundle eligible for use, whereas the ORM module isn't compatible with Hibernate 3.0.

As you can see, when OSGi-ifying a library, good practice consists of indicating the version of each package in the `Import-Package` header. Spring ORM declares that it works with Hibernate from version 3.2.6.ga, inclusive, to 4.0.0, exclusive:

```
Import-Package: org.hibernate;version="[3.2.6.ga,
4.0.0)",org.hibernate.cache;version="[3.2.6.ga, 4.0.0)",
org.hibernate.cfg;version="[3.2.6.ga, 4.0.0)"
(...)
```

NOTE The "ga" version qualifier stands for "General Availability" and denotes a stable, production-ready version of the software.

Spring ORM not only includes support for Hibernate, but also for iBATIS, amongst others, so the Spring ORM bundle can apply the same pattern for declaring dependencies on iBATIS:

```
Import-Package: org.hibernate;version="[3.2.6.ga,
4.0.0)",org.hibernate.cache;version="[3.2.6.ga, 4.0.0)",
org.hibernate.cfg;version="[3.2.6.ga, 4.0.0)",
(...)
com.ibatis.common.util;version="[2.3.0.677, 3.0.0)",
com.ibatis.common.xml;version="[2.3.0.677, 3.0.0)",
com.ibatis.sqlmap.client;version="[2.3.0.677, 3.0.0)"
(...)
```

Nice, but let's imagine you're working on an application that uses Hibernate and the support provided by Spring ORM. You provision your OSGi container with the corresponding bundles, but you soon notice that if you want the Spring ORM bundle to be resolved, you need all of its dependencies in your container, like iBATIS or OpenJPA, even if you only use Hibernate. That's a real pain, because you'll have to get all these dependencies as OSGi bundles and deal with their dependencies—all for nothing because you don't even use them!

Don't panic, there's a solution: these kinds of dependencies can be marked as optional in the manifest, by using the `resolution` directive. This directive defaults to `mandatory`, meaning that the bundle won't be able to resolve successfully if the imported package isn't present in the container. The `resolution` directive can also take the `optional` value, to indicate that the importing bundle can successfully resolve even if the imported package isn't present. Of course, if some code that relies on the missing import is called at runtime, it will fail.

Spring ORM declares its dependencies on ORM tools as optional, because there is little chance that all these libraries will be used at the same time in an application:

```
Import-Package: org.hibernate;version="[3.2.6.ga,
4.0.0)";resolution:=optional,
org.hibernate.cache;version="[3.2.6.ga,
4.0.0)";resolution:=optional,org.hibernate.cfg;version="[3.2.6.ga,
4.0.0)";resolution:=optional,
(...)
com.ibatis.common.util;version="[2.3.0.677, 3.0.0)";resolution:=optional,
com.ibatis.common.xml;version="[2.3.0.677, 3.0.0)";resolution:=optional,
com.ibatis.sqlmap.client;version="[2.3.0.677, 3.0.0)";resolution:=optional
(...)
```

To sum up, when OSGi-ifying libraries or frameworks, you should remember the following guidelines with respect to the `Import-Package` header:

- Import the packages that the library or framework uses, and don't import unused packages, which would tie the bundle to unnecessary dependencies.
- Specify the version of the packages, so the library or framework won't use classes that it isn't meant to use, which could lead to unexpected behavior.
- Specify the difference between mandatory and optional dependencies by using the `resolution` directive.

That's enough about importing from other bundles; let's see now how a library can make its classes visible in the OSGi platform.

EXPORTING PACKAGES

Which packages need to be exported by a library depend on its design. Some libraries clearly make the distinction between their API and their implementation classes, through some kind of special structuring of their packages. For example, interfaces (the API) may be located in one package and internal classes (implementation, utilities) in an `impl` or `internal` subpackage.

> **NOTE** Generally speaking, splitting API and implementation packages is a good design practice, not only in OSGi.

Nevertheless, the export declarations will usually end up exporting all the packages of a bundle because even if we follow the programming through interface pattern, we'll usually need an implementation that's provided by the same library as the API.

> **NOTE** We'll see more about design in section 6.3, so we'll keep things simple for now. Just remember that OSGi services are a good way to expose what other bundles need to use. This keep implementation details from leaking through the whole system.

In the `Export-Package` header, you should always specify the version of the exported package. The following snippet shows the first line of the `Export-Package` header from the Spring ORM module manifest (notice the use of the `version` attribute):

```
Export-Package: org.springframework.orm;version="2.5.6.A",
(...)
```

The `version` value can be different for each exported package, but usually all the exported packages will share the same version as the owning bundle. There are some exceptions, but this generalization covers most cases.

`Import-Package` and `Export-Package` are the most important headers to specify when OSGi-ifying libraries, but there are a few others to take note of, especially those used to identify a bundle.

GIVING AN IDENTITY TO A BUNDLE

In an OSGi environment, a library must be properly identified, because dependency resolution in OSGi builds on bundle-identity mechanisms. We've looked at many manifest headers already, especially in chapter 2, so we won't describe all of them again. We'll focus on three here.

The following snippet (part of the Spring Core 2.5.6.A bundle manifest) shows these three manifest headers:

```
Bundle-SymbolicName: org.springframework.core
Bundle-Version: 2.5.6.A
Bundle-Name: Spring Core
```

The `Bundle-SymbolicName` header specifies a unique name for a bundle, usually based on the reverse package (or domain) convention. The header value can't contain any whitespace—only alphanumeric characters, periods (`.`), underscores (_), and hyphens (`-`). The `Bundle-SymbolicName` header is compulsory, it doesn't take a default value, and it must be set carefully because it's the main component of your bundle identity.

The other aspect of a bundle identity is its version, set with the `Bundle-Version` header. Unlike the `Bundle-SymbolicName` header, the version header isn't compulsory and it defaults to `0.0.0`, but it should *always* be explicitly set. When setting the bundle version, you should follow the format and semantics of OSGi versioning (major, minor, and micro numbers, and qualifier), as explained in chapter 2. The symbolic name and version tuple comprises the identity of your bundle: there can't be two bundles with the same symbolic name and version number installed at the same time in an OSGi container.

`Bundle-Name` isn't meant to be used directly by the OSGi platform but rather by developers, because it defines a human-readable name for the bundle. Its value can contain spaces and doesn't have to be unique (even though it should be, to avoid confusion). It needs to be explanatory enough.

Now that you've seen the theory behind the OSGi-ification of libraries, let's discuss putting this into practice and see the different ways to convert a plain JAR file into an OSGi bundle.

6.2.2 Converting by hand

Because the deployment unit in OSGi is the JAR file along with some metadata, the conversion boils down to carefully editing the MANIFEST.MF file. We've already discussed the manifest headers, but we should not forget the specifics of the JAR packaging:[1]

[1] You can find more about these requirements from the JAR file specification: http://java.sun.com/j2se/1.5.0/docs/guide/jar/jar.html.

- The META-INF/MANIFEST.MF file must be the first entry in the JAR, and the `jar` command enforces this rule (so you shouldn't try to package your OSGi bundles manually).
- The manifest format has strict requirements. For instance, lines can't be longer than 72 characters and the file should end with an empty line.

Given these requirements and the sensitive needs of OSGi metadata, manually editing an OSGi manifest can end up being a nightmare. A typo or extra space can break the manifest and be difficult to track down. Take a look at the manifest of each module in the Spring Framework and imagine the daunting task of maintaining each manually. Imagine doing this for a bunch of Java EE frameworks, like Hibernate or JavaServer Faces!

Manually editing manifests, without any support from tools, isn't a realistic or desirable undertaking. In the next section, we'll discuss tools that can help you to reliably package your OSGi bundles.

6.2.3 Converting using tools

You can't deny that your life as a developer wouldn't be the same without the tools you rely on every day. You'll also probably have strong opinions on tooling: developers should not become too dependent on their tools and should know exactly what these tools do for them under the covers.

Java and Java EE have a large set of tools, both commercial and open source: IDEs (for content assistance, debugging, and so on), build tools, continuous integration servers, and many more. The good news is that OSGi tooling is getting better and better. We'll focus in this section on tools that can help you package Java libraries into OSGi-compliant JAR files. We'll adopt a progressive approach: we'll start by using a command line tool, Bnd, and we'll end up including the OSGi-ification process into a Maven 2 build. The Apache Commons DBCP, the database connection pool library, will be our candidate library.

THE BND TOOL

Bnd (http://www.aqute.biz/Code/Bnd) is a tool created by Peter Kriens to help to analyze JAR files and to diagnose and create OSGi R4 bundles. It's used internally by the OSGi Alliance to create OSGi libraries for the various OSGi reference implementations and Technology Compatibility Kits (TCKs). Bnd consists of a unique JAR file but it can be used from the command line, as an Eclipse plug-in, or from Ant (yes, a JAR can be all of this). You already know from chapter 3 that the Felix Bundle Plugin for Maven is based on Bnd.

> **Are there any other tools than Bnd?**
>
> Bnd is arguably the most popular tool for packaging JARs as OSGi bundles, but OSGi tooling is getting more and more widespread. The latest rival for Bnd is Bundlor (http://www.springsource.org/bundlor), a tool created by the SpringSource team to automate the creation of OSGi bundles.

(continued)

Like Bnd, Bundlor analyzes class files to detect dependencies, but it's also able to parse different kinds of files to detect *more* dependencies: Spring application context XML files, JPA's persistence.xml, Hibernate mapping files, and even property files. Bundlor follows a template-based approach, which consists of giving hints for the manifest generation in the guise of a property file (the same approach used by Bnd). At the time of this writing, Bundlor is still quite new, but it can already be used with Ant and Maven 2. The use of Bundlor with Maven 2 is covered in appendix B.

OSGi also gets into your development environment: there has always been the Plug-in Development Environment (PDE) in Eclipse (http://www.eclipse.org/pde/), which enables the development of Eclipse plug-ins and offers some nice support for OSGi (such as a dedicated editor for manifest files). More recent is the SpringSource Tool Suite (STS), http://www.springsource.com/products/sts) a dedicated Eclipse distribution targeting the development of Spring- and SpringSource dm Server–based applications. As SpringSource dm Server applications rely heavily on OSGi, STS offers some support for OSGi. STS was once a commercial product but has been free since mid-2009.

In this section, we'll use Bnd from the command line to OSGi-ify Apache Commons DBCP 1.2.2. So let's get down to business! Download Bnd from http://www.aqute.biz/Code/Bnd, DBCP 1.2.2 from http://commons.apache.org/dbcp/, and copy the two JARs into a working directory (both JAR files are also available in the code samples for this book).

Why Apache Commons DBCP?

Commons DBCP is a very popular database connection pool: Apache Tomcat uses it to provide its data sources, and a lot of applications embed a DBCP connection pool (often as a Spring bean). Unfortunately, DBCP isn't yet among the OSGi-ified libraries of the Apache Commons family. Converting Commons DBCP is a good exercise, and it will prove to be useful, because we'll use our brand new OSGi-ified version in chapter 7.

Note that you should stick to the version of DBCP (and of its dependency, Commons Pool) that we're using in this book, because it's likely they will be distributed as OSGi bundles one day!

You can't convert a plain JAR file into an OSGi-compliant bundle without knowing a little about it. That's why Bnd comes with the `print` command:

```
java -jar bnd-0.0.313.jar print commons-dbcp-1.2.2.jar
```

Don't be overwhelmed by the output. It's divided into sections, and we're going to analyze the most important ones.

The first section provides information taken from the manifest:

```
[MANIFEST commons-dbcp-1.2.2.jar]
Ant-Version                          Apache Ant 1.5.3
Build-Jdk                            1.4.2_10
Built-By                             psteitz
Created-By                           Apache Maven
Extension-Name                       commons-dbcp
Implementation-Title                 org.apache.commons.dbcp
Implementation-Vendor                The Apache Software Foundation
Implementation-Vendor-Id             org.apache
Implementation-Version               1.2.2
Manifest-Version                     1.0
Package                              org.apache.commons.dbcp
Specification-Title                  Commons Database Connection Pooling
Specification-Vendor                 The Apache Software Foundation
X-Compile-Source-JDK                 1.3
X-Compile-Target-JDK                 1.3
```

The more interesting section is the one starting with `[USES]`, which delivers information about the Java packages of the target JAR:

```
[USES]
org.apache.commons.dbcp                  java.sql
                                         javax.naming
                                         javax.naming.spi
                                         javax.sql
                                         org.apache.commons.jocl
                                         org.apache.commons.pool
                                         org.apache.commons.pool.impl
                                         org.xml.sax
org.apache.commons.dbcp.cpdsadapter      java.sql
                                         javax.naming
                                         javax.naming.spi
                                         javax.sql
                                         org.apache.commons.dbcp
                                         org.apache.commons.pool
                                         org.apache.commons.pool.impl
(...)
```

We now know which packages our library depends on. The output ends with an error section:

```
One error
1 : Unresolved references to [javax.naming, javax.naming.spi,
javax.sql, org.apache.commons.pool, org.apache.commons.pool.im
pl, org.xml.sax, org.xml.sax.helpers] by class(es) on the Bund
le-Classpath[Jar:commons-dbcp-1.2.2.jar]: [org/apache/commons/
dbcp/datasources/PerUserPoolDataSource.class,(...)
```

With this monolithic block of text, Bnd tells us that, with respect to the current classpath, some packages that our library needs to work are missing. We can also see that Commons DBCP depends on the org.apache.commons.pool and org.apache.commons.pool.impl packages. Indeed, Commons DBCP relies on the Commons Pool library to handle its pooling algorithm and adds a thin layer on top of it for database connections.

This dependency means that we'll need to do two things in the OSGi-ification of Commons DBCP:

- Properly import packages from Commons Pool
- Have Commons Pool packaged as an OSGi bundle

We can immediately start the OSGi-ification with Bnd's `wrap` command:

```
java -jar bnd-0.0.313.jar wrap commons-dbcp-1.2.2.jar
```

This creates a `commons-dbcp-1.2.2.bar` file in the same directory, with an OSGi-compliant manifest and all the defaults for OSGi manifest headers. Unfortunately, Bnd can't guess the proper values for some important headers, and default values aren't always appropriate. That's why Bnd uses a configuration file to supply this information: version, symbolic name, imports, and exports can be defined in a way that's similar to the manifest format but more editor-friendly and more powerful, thanks to the use of variable substitutions and pattern matching.

Where do the .class files come from?

Bnd isn't a traditional packaging tool; it doesn't need as an input a directory containing `.class` files that it will compress in a JAR file. It directly locates `.class` files in the classpath and packages them into a JAR file. You can potentially include in your OSGi bundle all the `.class` files from the classpath you specified when launching Bnd from the command line.

The following snippet shows the Bnd configuration file for converting Commons DBCP into an OSGi bundle:

```
version=1.2.2                                           ❶
Bundle-SymbolicName: org.apache.commons.dbcp            ❷
Bundle-Version: ${version}                              ❸
Bundle-Name: Commons DBCP
Bundle-Description: DBCP connection pool
Export-Package: org.apache.commons.dbcp.*;
➥     version=${version}                                ❹
Import-Package:org.apache.commons.pool.*;version=1.3.0,
➥     org.apache.commons.dbcp*;version=${version},*;
➥     resolution:=optional                              ❺
```

Bnd allows variable substitution, so we use this feature for the version ❶ because it's needed at several places in the template. We then specify the bundle's symbolic name ❷ and the version ❸, using the `version` variable, with the `${variableName}` pattern. We also specify which packages the bundle will export ❹: notice that we use a wildcard (`*`) to specify that we want to export the `org.apache.commons.dbcp` package and all its subpackages. We use the `version` variable again to specify the version of the exported packages. Finally, we specify that the bundle imports version 1.3.0 of all the Commons Pool packages it references ❺. Notice that we import the

Commons DBCP packages with the same version, to ensure a consistent class space. The last wildcard refers to all the remaining packages used by Commons DBCP, and we mark them as optional.

> **NOTE** With Bnd, you should always define configurations from the most specific to the most general. If an element is matched twice, the first match always takes precedence. That's why the instruction to mark the dependencies as optional in our `Import-Package` header comes last.

Let's issue the `wrap` command again, but now with the `properties` option, to specify the Bnd configuration file:

```
java -jar bnd-0.0.313.jar wrap -properties commons-dbcp-1.2.2.bnd
➥     commons-dbcp-1.2.2.jar
```

We can now look at the manifest file of the generated OSGi bundle. Here's an excerpt showing the `Export-Package` and `Import-Package` headers:

```
(...)
Export-Package: org.apache.commons.dbcp.cpdsadapter;uses:="javax.namin
 g,javax.sql,org.apache.commons.pool.impl,org.apache.commons.pool,java
 x.naming.spi,org.apache.commons.dbcp";version="1.2.2",org.apache.comm
 ons.dbcp;uses:="org.apache.commons.pool.impl,org.apache.commons.pool,
 javax.sql,javax.naming,javax.naming.spi,org.xml.sax";version="1.2.2",
 org.apache.commons.dbcp.datasources;uses:="org.apache.commons.dbcp,ja
 vax.sql,org.apache.commons.pool,javax.naming,javax.naming.spi,org.apa
 che.commons.pool.impl";version="1.2.2"
(...)
Import-Package: javax.naming;resolution:=optional,javax.naming.spi;res
 olution:=optional,javax.sql;resolution:=optional,org.apache.commons.d
 bcp;version="1.2.2",org.apache.commons.dbcp.cpdsadapter;version="1.2.
 2",org.apache.commons.dbcp.datasources;version="1.2.2",org.apache.com
 mons.pool;version="1.3.0",org.apache.commons.pool.impl;version="1.3.0
 ",org.xml.sax;resolution:=optional,org.xml.sax.helpers;resolution:=op
 tional
```

Now you can compare the end result with the instructions we specified in the Bnd configuration file. As you can see, Bnd is a very convenient tool. You now have an OSGi-compliant version of Commons DBCP.

> **TIP** Because Commons DBCP isn't distributed as an OSGi bundle, a good practice is to include "osgi" in the filename: commons-dbcp-osgi-1.2.2.jar. If your bundle turns out to be distributed and is used by third parties, you can also prefix it with your company name: com.manning.commons-dbcp-osgi-1.2.2.jar.

We mentioned that we also need an OSGi-compliant version of Commons Pool, because Commons DBCP is built on this library. Unfortunately, Commons Pool isn't distributed as an OSGi bundle either, so we have to again do the conversion ourselves. It turns out to be fairly simple, because we can follow the same process as for Commons DBCP. The following snippet shows the Bnd configuration file for Commons Pool:

```
version=1.3.0
Bundle-SymbolicName=org.apache.commons.pool
Bundle-Version: ${version}
Export-Package: org.apache.commons.pool*;version=${version}
Bundle-Name: Commons Pool
```

Congratulations, you can now create database connection pools with Commons DBCP in an OSGi environment! But perhaps you're fond of automation; we'll look now at how to make the OSGi-ification part of a Maven build.

THE FELIX BUNDLE PLUGIN FOR MAVEN 2

You met the Felix Bundle Plugin in chapter 3, where we used it to package our first Spring DM bundles. We relied on Maven 2 to provision the Spring DM OSGi test framework. You'll get to know it better in this section, because we'll repeat the OSGi-ification of Commons DBCP, but in a 100 percent Maven 2 style this time.

The Felix Bundle Plugin provides integration between Bnd and Maven 2: the plug-in uses Bnd under the covers, providing it with information from the POM file. By using this plug-in, you can take advantage of all of Maven 2's features (automation, dependency management, standard project structure, and so on) and still package your project as OSGi-compliant bundles. The plug-in has reasonable default behavior, making the configuration simple for simple needs.

For the OSGi-ification of Commons DBCP, we start by creating a simple pom.xml file:

```
<?xml version="1.0"?>
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.manning.sdmia</groupId>
  <artifactId>commons-dbcp.osgi</artifactId>
  <version>1.2.2-SNAPSHOT</version>
  <packaging>bundle</packaging>
  <name>commons-dbcp.osgi</name>
  <description>
    OSGified version of Commons DBCP
  </description>

  <dependencies>
    <dependency>
      <groupId>commons-dbcp</groupId>
      <artifactId>commons-dbcp</artifactId>
      <version>1.2.2</version>
      <scope>provided</scope>
    </dependency>
  </dependencies>
</project>
```

Defines project identity

Sets bundle as packaging

Describes bundle

Adds Commons DBCP dependency

Notice how we clearly state that the project is our own distribution of an OSGi bundle:

- The `groupId` refers to our company
- The `artifactId` is postfixed with `osgi`

Even if Bnd is wrapped in a Maven plug-in, it still bases its search for classes on the classpath, so we add Commons DBCP as a Maven dependency.

We now need to explicitly reference the Felix Bundle Plugin; otherwise the bundle packaging doesn't have any meaning for Maven 2. We do this inside the `build` tag (just before the `dependencies` tag), where we usually configure Maven 2 plug-ins. Listing 6.1 shows the configuration of the Felix Bundle Plugin for OSGi-ifying Commons DBCP.

Listing 6.1 Felix Bundle Plugin configuration for OSGi-ifying Commons DBCP

```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.felix</groupId>
      <artifactId>maven-bundle-plugin</artifactId>
      <version>2.0.1</version>
      <extensions>true</extensions>
      <configuration>
        <instructions>
          <Bundle-SymbolicName>
            org.apache.commons.dbcp
          </Bundle-SymbolicName>
          <Export-Package>
   org.apache.commons.dbcp*;version=${project.version}
          </Export-Package>
          <Import-Package>
            org.apache.commons.pool*;version="1.3.0",
            *;resolution:=optional
          </Import-Package>
          <Embed-Dependency>
            *;scope=provided;type=!pom;inline=true
          </Embed-Dependency>
        </instructions>
      </configuration>
    </plugin>
  </plugins>
</build>
```

❶ Declares plug-in

❷ Sets bundle's symbolic name

❸ Sets packages to export and import

❹ Instructs how to handle dependencies

We start by declaring the plug-in ❶. Never omit the version of a plug-in with Maven 2 unless you want your build to break unpredictably.

The configuration starts with the `configuration` and `instructions` elements. We use the `Bundle-SymbolicName` tag to set the manifest header ❷. We then use the `Export-Package` instruction to define the Java packages the bundle will export ❸. Notice that we can use the same syntax as in Bnd files to include subpackages. This time, we didn't define a variable for the version, because we can refer to the project version directly with the `${project.version}` variable. We define imported packages the same way as in plain Bnd ❸. Finally, we use the `Embed-Dependency` tag to tell the plug-in how to handle dependencies ❹: include all dependencies with `provided` scope (but exclude dependencies of type POM) and copy them inline in the JAR.

All set! Any Maven packaging goal (`install` or `package`) will generate a 100 percent OSGi-compliant bundle.

NOTE The Commons Pool library can also be easily OSGi-fied with the Felix Bundle Plugin.

OSGi-ifying a library and making the process part of a traditional build is fairly simple, thanks to the Felix Bundle Plugin. You just need to be careful with the generated OSGi metadata, and Bnd will handle the rest.

We've now talked a lot about converting existing libraries, but what about packaging our own modules and applications? We'll discuss this topic in the next section.

6.2.4 Packaging your own modules as OSGi bundles

If you understand how to OSGi-ify existing libraries, making your own Java applications and modules into OSGi bundles should not be a problem for you. You can apply all the OSGi-ification techniques we've covered so far to your own modules. You can stick to Bnd, choosing the mechanism that suits you best:

- Command line—Straight and simple, but difficult to automate
- Eclipse plug-in—Embedded in your development environment, but still difficult to automate
- Ant task—Included in your build, and perfect if Ant is your tool of choice for all your builds (this is covered in appendix C)
- Maven 2 plug-in—Included in your build, and fits perfectly with any Maven 2–based project

Packaging existing libraries or your own modules as OSGi bundles should not cause you any trouble now. Nevertheless, OSGi isn't only about packaging. It's also about *modularity*, and without good design, you'll have a hard time packaging your modules. That's why we'll discuss how to design OSGi enterprise applications and how to leverage Spring DM in the next section.

6.3 Designing OSGi enterprise applications

Designing OSGi enterprise applications isn't so different from developing "standard" enterprise applications: OSGi people don't pretend that the world was waiting for them in order to write modular applications. Nevertheless, anyone can learn from the strict modular approach of OSGi.

With plain Java, we can't really encapsulate our classes and interfaces; they can be used as long as they're on the classpath. The standard deployment unit in Java, the JAR file, is a convenient kind of packaging, but Java doesn't provide us with real dynamic deployment capabilities. Web (WAR) and enterprise (EAR) deployment units usually end up being monolithic, hard to split entities; they're too coarse-grained.

We, as enterprise application developers, have learned to get along with these pitfalls. But even if we managed to write well-designed, layered applications with Spring, we can still improve them and even take advantage—especially at runtime—of the way we designed them.

To see how OSGi can help to improve the design of a Java application, we'll progressively transform a standard web application packaged as a monolithic WAR file into a modular web application. Along the way, we'll also see how to introduce Spring DM into our OSGi design.

6.3.1 Organizing OSGi components

Let's start with a standard web application and reorganize it to obtain a truly modular, OSGi-compliant application.

ORGANIZING THE DEPENDENCIES

In Java, web applications are packaged as WAR files. The WAR structure is quite simple:

- It's a ZIP file
- Downloadable resources (images, JavaScript files) are located at the root of the WAR
- Application classes (servlet, web controllers) are located in WEB-INF/classes
- Libraries and frameworks (packaged as JAR files) are in WEB-INF/lib

Let's focus first on libraries and frameworks; this is the first place where OSGi can help, because the WAR packaging has some pitfalls. Web applications can embed these JAR files or let the application server provide them, as shown in figure 6.1. If the application server provides them, WAR files are smaller, and depending on the application server, the global memory footprint is also smaller.

This scenario, where the application server provides the libraries and frameworks, works well because both web applications depend on the same versions of the frameworks they use.

Figure 6.2 shows another scenario, where the production team said to the developing team: "Spring and Hibernate are provided by the application server, so don't embed them in the WAR." Unfortunately, application 2 needs different versions of Spring and Hibernate than those provided by the application server.

What would happen in figure 6.2, when the application server provides the libraries? Nobody knows. We'd have to cross our fingers and see. Application 1 has no reason not

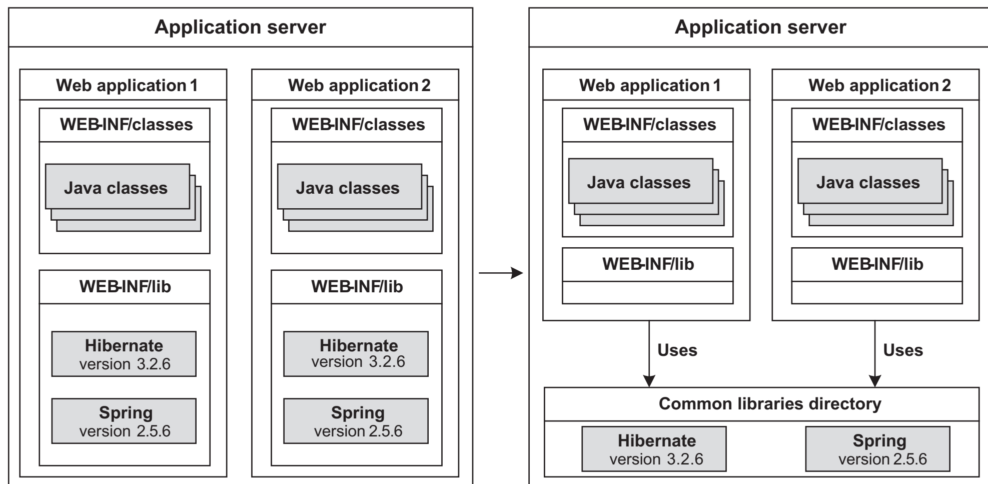

Figure 6.1 Within an application server, web applications can embed their dependencies or let the application server provide them.

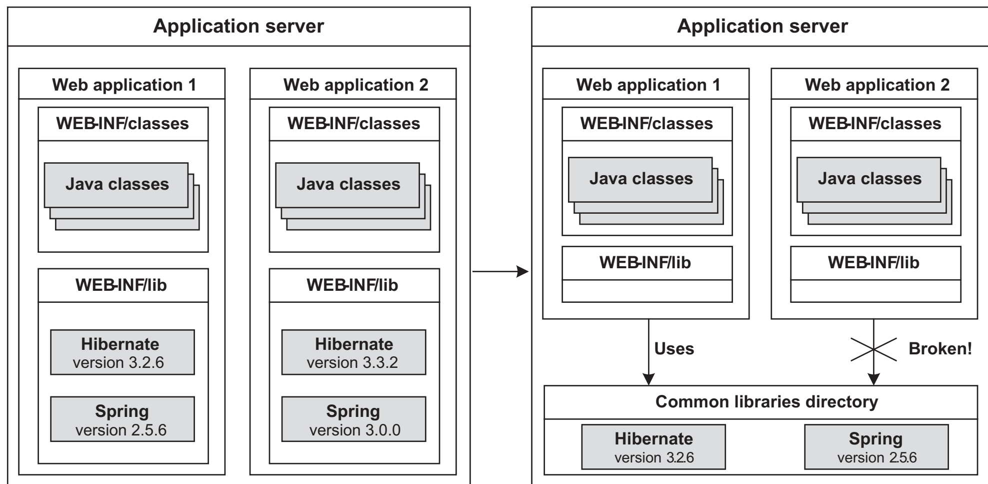

Figure 6.2 Application 2 uses different versions of Spring and Hibernate than those provided by the application server. It can't safely rely on these versions.

to work, but we can't know about application 2: perhaps it won't start, or maybe it'll fail when a user triggers an action that relies on Spring 3.0 or Hibernate 3.3.

On standard Java EE application servers, applications have no standard way to indicate that they depend on a particular version of a framework; these kinds of metadata don't exist in Java EE standards. We could try to check the version when the application starts, but this would be cumbersome, especially if we had to do the check for all of the dependencies. We could embed the libraries in the WAR (as shown in figure 6.3), but this could lead to unpredictable behavior, depending on the application server's classloading strategy and the version of the WAR-embedded and server-provided frameworks: classes could be partially loaded from different JAR files, or frameworks could try to dynamically detect some libraries (Hibernate does that with Hibernate Validator). We usually call this "JAR hell."

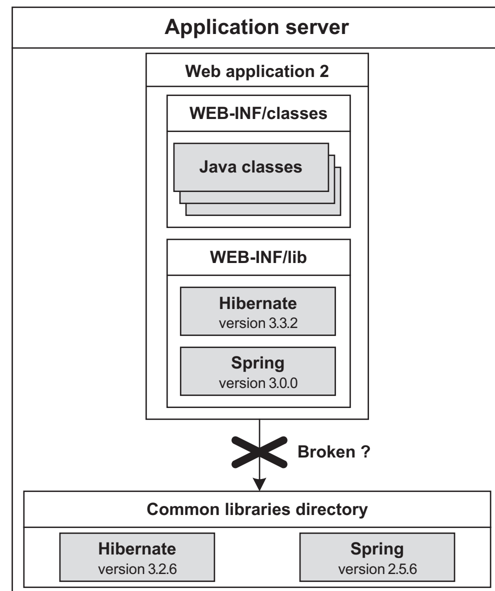

Figure 6.3 Application 2 tries to embed its dependencies. This can lead to unexpected behavior and hard-to-debug issues.

With OSGi, there is no room for approximation. Modules declare their dependencies, and the OSGi container is in charge of their resolution. We

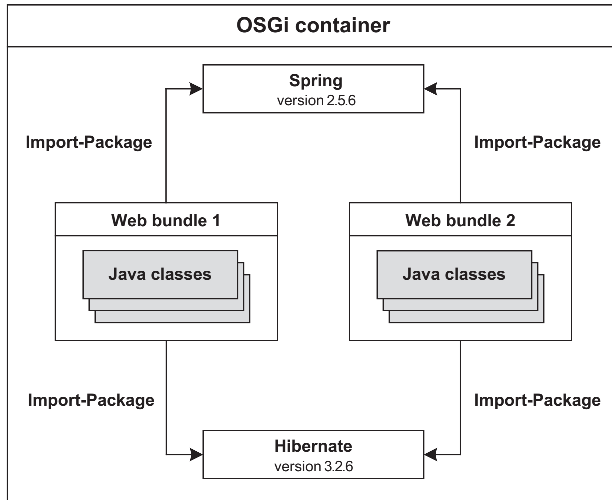

Figure 6.4 In OSGi environments, modules explicitly declare their dependencies and don't embed them.

don't need to write our own checks; the platform does it for us. If we fail to properly declare our module dependencies, shame on us.

The OSGi world is harsh, but in the end you'll benefit from this. Figure 6.4 shows how the two web applications can use the same libraries in an OSGi environment. Applications declare their dependencies, and the OSGi container does the classloader wiring to the respective dependency bundles.

Figure 6.4 demonstrates a simple scenario, so let's look at a more complex one, where applications don't rely on the same versions of frameworks, and we still want to share the dependencies among modules. This scenario is shown in figure 6.5.

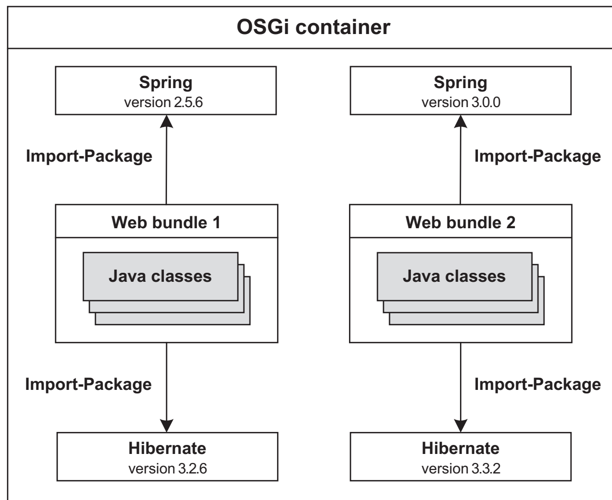

Figure 6.5 In OSGi environments, different versions of the same modules can be deployed, and dependent modules declare which version they want to use.

Thanks to its sophisticated classloading mechanisms, OSGi supports this scenario out of the box: different versions of the same library can coexist in a container.

We've looked at dependencies, so let's move on and see how we can organize our application.

ORGANIZING THE APPLICATION

Enterprise applications are usually organized as a stack of layers, where each layer has its own responsibilities and relies on the layer immediately below. This layer organization encourages best programming practices, such as separation of concerns and unit testing. The so-called domain layer is an exception: it represents business entities (customers, contracts, and so on), which mainly carry data across the layers. As such, domain entities are used in all the layers. Figure 6.6 pictures a layered application.

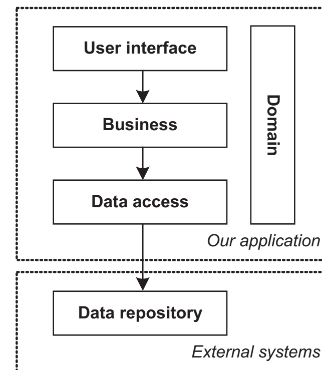

Figure 6.6 Enterprise applications are layered, for a better separation of concerns.

Domain-driven design versus the anemic domain model

Saying that domain classes carry only data is a bold statement—they can also contain behavior in the form of business-oriented methods. Layering applications is good, but, applied in a simplistic way, it can lead to poorly designed enterprise applications, whose structure becomes closer to procedural programming than real OOP. This is especially true for the domain classes, which are then limited to data-transfer tasks. Such a domain layer is commonly referred to as an *anemic domain model*. Domain-driven design promotes rich domain models, where some parts of the business logic are embedded in the domain classes.

OSGi, layered applications, and domain-driven design can cohabit, but comprehensive coverage of these topics is far beyond the scope of this book. If you want to learn more about the common pitfalls that too-strict layering can lead to, we recommend Martin Fowler's article about the anemic domain model (http://www.martinfowler.com/bliki/AnemicDomainModel.html).

How can we translate the layer concept into Java? We accomplish this by using the simplest constructs of Java—classes and interfaces—as shown in the UML diagram in figure 6.7. (This is about the design; at runtime, we'll need a little help from a lightweight container like Spring.)

The next question is how to split our OSGi bundles, now that our design tells us more about our dependencies? The answer is the one we usually don't like: it depends. It depends on various factors, such as these:

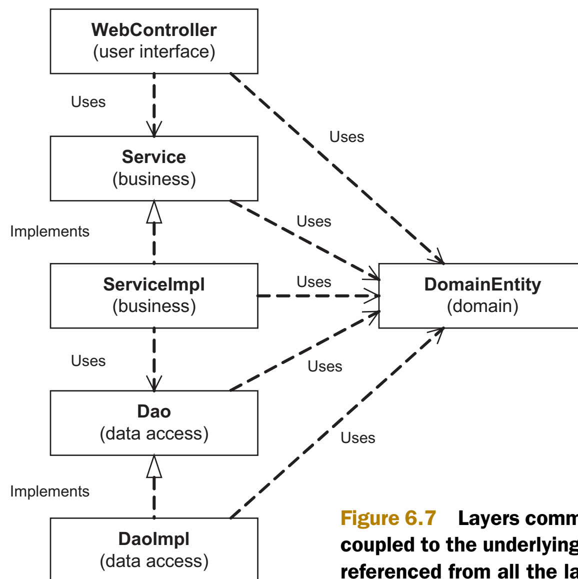

Figure 6.7 Layers communicate through interfaces to avoid being coupled to the underlying technology. Only the domain classes are referenced from all the layers.

- How we organize development and teams (this isn't related to OSGi)
- How modules are designed to evolve and be developed, updated, and refactored
- How modules are meant to be reused by other modules
- How we want to expose services (business services, data access objects, and so on), and how we want to encapsulate and hide inner mechanisms
- How many implementations of the same service are going to be deployed

OSGi static and dynamic features offer so much power that we have a choice of virtually unlimited combinations! Some combinations are good, but others should be avoided, so let's see some guidelines.

The first tip when developing OSGi applications is to separate the static components from the dynamic components.

- *Static components* are bundles that define APIs by exporting interface-based Java packages. These bundles don't contain a Spring application context (or a `BundleActivator`) and don't register or reference any OSGi services.
- *Dynamic components* are bundles that import Java packages from static bundles, provide implementations, and usually register OSGi services. As this is a Spring DM book, these bundles are Spring-powered.

Figure 6.8 illustrates the pattern of splitting static and dynamic components.

Generally, static components should be stable, changing infrequently, whereas dynamic components can be frequently updated and benefit from OSGi dynamic features such as on-the-fly service updating. Section 6.4, which deals with Spring DM and OSGi dynamic features, will explain just how using Spring DM is relevant to

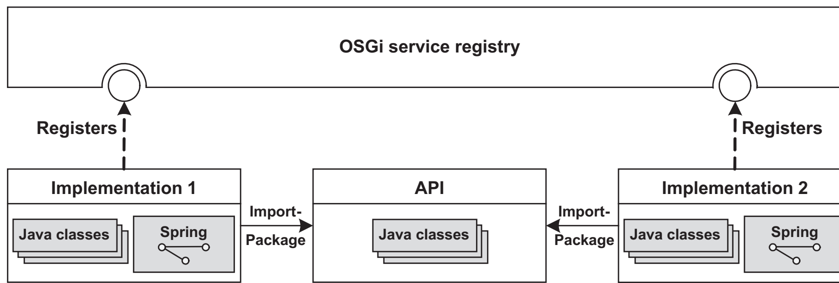

Figure 6.8 Static parts of an application (such as the API) and dynamic parts (the implementations) should always be deployed in different bundles within an OSGi environment. This helps the application to benefit from the dynamic features of OSGi.

implementing those dynamic components—the framework takes care of all the dirty work during bundle updates.

Now, let's see how we can apply this pattern to our enterprise application. The most extreme modular approach would consist in developing

- One bundle for each layer that has only classes (the domain and web layers)
- Two bundles for each layer that has interfaces and class implementations (the data access and business layers)

This approach could be labeled the "So you want modularity" approach, and it's shown in figure 6.9.

The approach shown in figure 6.9 is the most flexible: you can update any part of your application, and other modules can be built on top of any of yours by importing your packages and defining other implementations. Also note that by using Spring

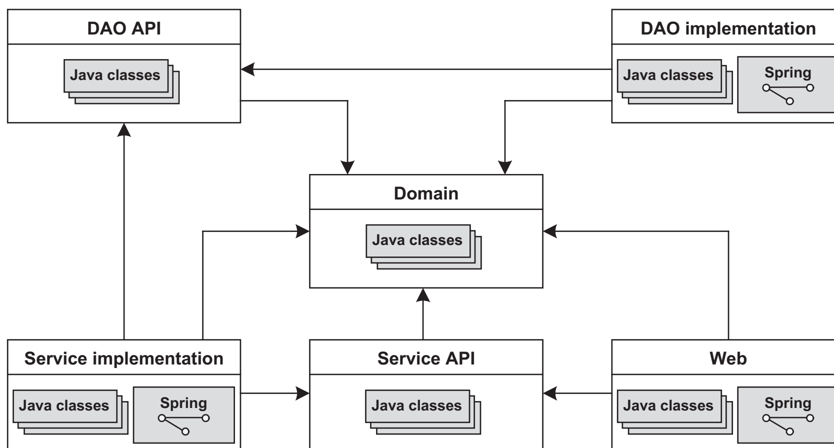

Figure 6.9 Organizing bundles in the "So you want modularity" way. There's at least one bundle for each layer, and two bundles if the layer has an API and implementation classes. (Arrows represent the `Import-Package` manifest header.)

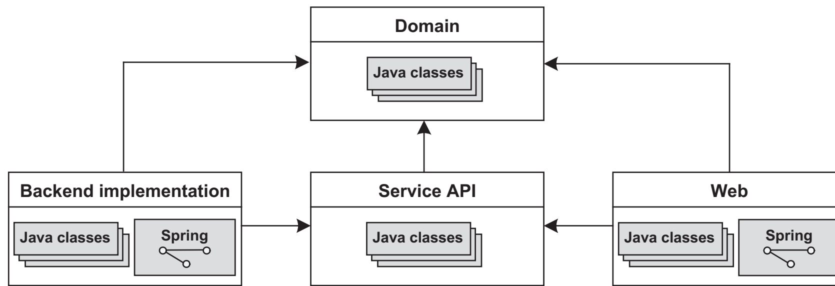

Figure 6.10 Organizing the bundles in a SOA way, the backend bundle hides its internal functioning. (Arrows represent the `Import-Package` manifest header.)

DM for implementation bundles, you benefit from dependency injection and all the enterprise support of the Spring Framework (data access, transaction management, and so on). One of the drawbacks of this approach is that you have to maintain of lot of bundles, usually as separate projects.

That approach gives us flexibility, but what if we don't want or don't need it? Perhaps exporting the packages of data access object (DAO) interfaces is useless, because what we're really interested in is the business services. Remember, OSGi is sometimes referred to as a service-oriented architecture (SOA) in a JVM. We can still define the service API, but the implementation can hide its inner workings. We can then reorganize our bundles in a simpler way and gather the service implementation and data access layer in the same bundle—the backend bundle.

Taking this approach doesn't change the logical organization of our application, as it's still a layered application; we just changed its physical organization. Figure 6.10 illustrates this new organization.

The SOA approach is no less flexible than the extremely modular approach. If DAOs and business services follow the same development and deployment cycles, there is no point in splitting them into different bundles.

We know now that the way we organize bundles is a question of balance. We've mainly discussed how bundles are *statically* linked by their dependencies, but we haven't considered how bundles communicate with each other using OSGi's service layer. That's the topic of the next section.

6.3.2 Defining interactions between application bundles

In our enterprise application, Java packages can be shared between bundles because of the `Export-Package` and `Import-Package` headers in their manifests. Nevertheless, Java packages aren't enough; an application needs real Java objects to run, and these objects must be registered as services in the OSGi registry. That's where Spring DM comes in.

Spring DM will instantiate and wire beans in our bundles and register them in the registry based on declarations in the context file. We'll end up not writing a single line

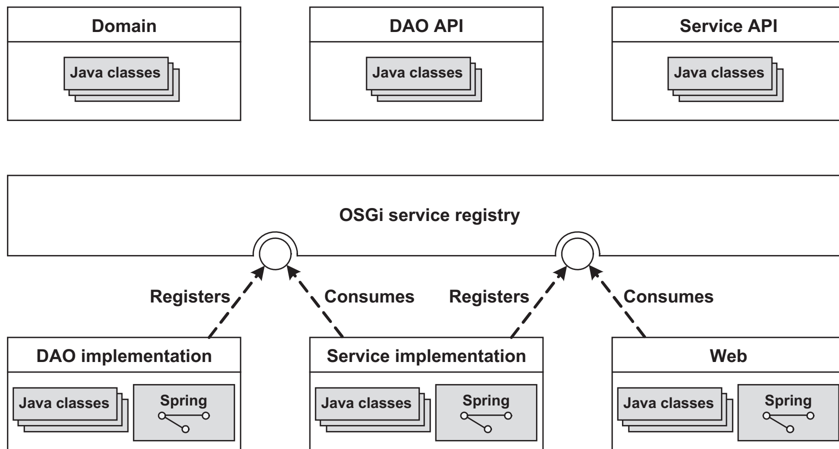

Figure 6.11 Spring DM helps implementation bundles to share OSGi services.

of code related to OSGi. (We'll see later that Spring DM will even handle OSGi's dynamic behavior for us.)

Let's again take our extreme modular approach from the previous section and focus on the service dependencies—we'll ignore dependencies related to Java packages for now. The implementation bundles are backed by Spring DM and can easily register or consume services (see figure 6.11).

In this scenario, if other bundles need to use our DAOs, they can easily consume them, regardless of whether or not they're Spring-powered bundles. Remember that our OSGi services are created by the Spring lightweight container, and as such, they can benefit from dependency injection or AOP. They can become transactional or get automatic database-connection handling with a few lines of XML or by inserting a couple of Java annotations. These are some of the benefits of using Spring DM.

Now, let's fall back to our simpler SOA approach. It works in much the same way, but the backend bundle has a bigger Spring application context because it hosts DAOs and business services. In this scenario, DAOs can't be consumed by other bundles, because the only entry point is embodied by the business services. The SOA scenario from the OSGi service layer's point of view is shown in figure 6.12.

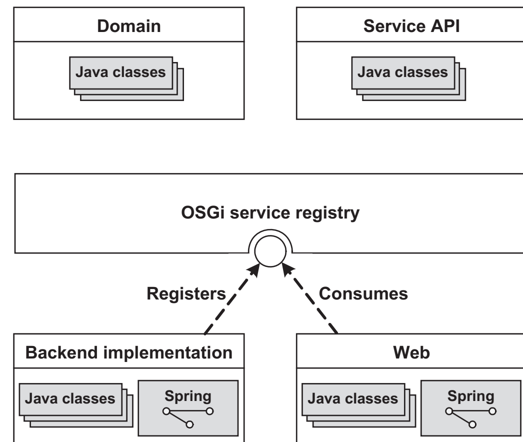

Figure 6.12 The SOA approach implies fewer registered services and offers better encapsulation.

What should we remember about the way bundles communicate? Mainly that there is still no simple answer and that we need to find a compromise between what we want to offer and what we want to hide. Generally speaking, we should only expose what is useful and is *prepared* to be used as a service: you should not let your system be compromised because a poorly written service isn't used the way it was meant to be.

In any event, Spring DM will be your friend when it comes to registering or consuming services. You'll like Spring DM even more when you see in the next section how it helps you handle dynamic behavior in OSGi.

6.4 How Spring DM handles OSGi applications' dynamic behavior

Within OSGi, services can appear and disappear at any time. This dynamic behavior is specific to OSGi; it's sophisticated and powerful but more complicated to deal with than static services.

Tracking services using plain OSGi is quite painful and error prone. The best tool OSGi offers for this task is the `ServiceTracker`, which accomplishes a lot, but we want more! When using the `ServiceTracker`, we still need to write code, and we're tied to the OSGi API. Moreover, we have to write the same kind of tracking code over and over.

That's where Spring DM comes in. You saw in the previous chapter that with Spring DM you can register and consume services declaratively. This looks static at first sight, but Spring DM handles all the dynamics for you, adopting a reasonable default behavior in most cases—default behavior that you can override.

In this section, we'll discuss typical cases related to OSGi's dynamic behavior and how Spring DM can help you deal properly and reliably with them. These cases range from the appearance and disappearance of a service or collection of services to the dynamic update of your modules.

6.4.1 Dealing with the appearance or disappearance of services

Spring DM's support for referencing services comes in two flavors: *individual*, when you need only one service matching a given description, and *collection*, when you want to have all the services that match some criteria. In enterprise applications, the individual case is the most common: a business service needs only one OSGi service implementing a given DAO interface. Spring DM is able to transparently handle service appearance and disappearance for both individual and collections of service references. We'll cover the mechanics of both flavors here.

When Spring DM's transparent management isn't enough, because you need to track services more carefully, Spring DM offers a simple POJO-oriented solution to react to the binding and unbinding of services. We'll also cover this topic, using a Swing application to illustrate it.

DEALING WITH AN INDIVIDUAL SERVICE REFERENCE

Let's go back to our enterprise application, using the extreme modular approach. Imagine it involves retrieving users from the database, so we'll have a user DAO,

created and wired in a Spring-powered bundle, and registered as an OSGi service by Spring DM:

```
<bean id="contactDao"
      class="com.manning.sdmia.directory.dao.jdbc.
➥    ContactDaoJdbc">
  <property name="dataSource" ref="dataSource" />
</bean>

<osgi:service
  id="contactDaoOsgi"
  interface="com.manning.sdmia.directory.dao.
➥    ContactDao"
  ref="contactDao" />
```

Creates and injects DAO

Registers DAO as OSGi service

This contact DAO is meant to be imported and used by business services, such as the contact business service, defined in another Spring-powered bundle:

```
<osgi:reference
  id="contactDao"
  interface="com.manning.sdmia.directory.dao.
➥    ContactDao">

<bean id="contactService"
      class="com.manning.sdmia.directory.service.impl.
➥                 ContactServiceImpl">
  <property name="contactDao" ref="contactDao" />
</bean>
```

Imports DAO from OSGi registry

Injects DAO into business service

In this scenario, there is only one OSGi service implementing the `ContactDao` interface, and it will be imported by the service. If there is more than one, Spring DM will pick one by following a predetermined strategy. If the choice doesn't suit the business service, that's too bad. It should have given Spring DM enough information to pick the right service.

But what happens if there's no OSGi service implementing the `ContactDao` interface? This could happen if the business service bundle is deployed in the OSGi container and the data access bundle isn't. In this scenario, Spring DM will figure out that the Spring application context of the business service bundle has a missing dependency, and it will defer the application context startup until the dependency is satisfied, which means when an OSGi service implementing the `ContactDao` interface is registered. If this condition isn't met after 5 minutes, Spring DM will throw an exception. This is the Spring DM default behavior: references are mandatory, and all mandatory references must be resolved before an application context can start. We covered how to change this default behavior in chapter 4, by using the `timeout` directive of the `Spring-Context` header.

This is reasonable default behavior, but what if the business service bundle doesn't contain only the user business service but critical business services that need to be available as soon as possible? They would be unavailable because the user business service doesn't have this unique dependency.

You can resolve this issue by making the reference to the contact DAO optional, by using the `availability` attribute of the `reference` tag:

```
<osgi:reference
  id="contactDao"
  interface="com.manning.sdmia.directory.dao.ContactDao"
  availability="optional" />
```

Now the business service application context will start up even if there is no contact DAO available in the OSGi service registry.

The user business service delegates data access operations to the `contactDao`, so what happens if the user business service handles an incoming request and calls the `contactDao`? Well, nothing. The call will block until a contact DAO service is registered. This behavior makes sense: the overall system isn't in a nominal state, and the missing dependency should not be missing for long, so we can wait until it appears.

All of this is handled by Spring DM, which injected a proxy into the user business service in place of the user DAO. This proxy blocks when someone tries to call it, but as soon as the target OSGi service (the contact DAO, in this example) appears on the service registry, the proxy delegates all the calls to it.

So far we've been talking about startup, but services can appear and disappear after the OSGi container has been started. Let's imagine the container reached its steady state a long time ago and that the DAO service then disappears. Any reference to it can be replaced on the fly if Spring DM finds a replacement for it. Finding this replacement will depend on the filter the importing bundle declared when it imported the DAO and on the availability of a matching service.

As you can see, in the case of an individual service reference, Spring DM handles most of the dynamic behavior. It does provide some opportunity for tuning, but the defaults should fit in most cases.

Let's now discuss the case of a collection of service references.

DEALING WITH A COLLECTION OF SERVICE REFERENCES

We learned in chapter 5 that with Spring DM we can declaratively reference collections of OSGi services, thanks to the `list` and `set` tags of the `osgi` namespace. Collections of services are interesting for the parts of an application that can be extended with additional services or for implementing observer-based patterns like the whiteboard pattern, where a central component (the whiteboard) periodically needs a snapshot of all the available services that meet some requirements (the listeners). What can Spring DM do about the content of these collections when services appear or disappear?

In fact, Spring DM populates the collections as needed. It adds matching services to the collection, and when one of the services referenced in the collection is unregistered from the OSGi registry, Spring DM automatically removes the reference from the collection. Note, though, that this is only valid for collections (java.util.List and java.util.Set) that are managed by Spring DM. Indeed, Spring DM can't track service appearances and disappearances and update collections that it doesn't totally control.

This is good news: we can use our collections of service references as any other collections. We usually use collections by iterating over them, using `Iterators`, but there's one important thing to know when iterating over a collection of service references managed by Spring DM: even the `Iterator` is dynamic. Imagine you start iterating over a collection of service references, and its content changes during the iteration because some services were unregistered and some matching services were registered. With Spring DM, you'll be aware of this immediately, because the `Iterator` will reflect these changes dynamically.

Thanks to Spring DM's transparent dynamic management, we're now well prepared to deal with the appearance or disappearance of services. Nevertheless, the support for the dynamic side of OSGi would be incomplete if we could not easily *track* services and react accordingly.

REACTING TO THE APPEARANCE AND DISAPPEARANCE OF SERVICES

In plain OSGi, the `ServiceTracker` is the Holy Grail for the developer who wants to track services. But despite its unquestionable usefulness, the `ServiceTracker` implies the use of the OSGi API and needs to be registered programmatically, which means writing a `BundleActivator`. This quickly becomes cumbersome, especially if we need to track services in many bundles.

We saw in chapter 5 that, when referencing a service (either with the `reference`, `list`, or `set` tags), we can attach a listener that will be warned when a matching service is registered or unregistered. This can be done with the `listener` tag of the `osgi` namespace. This is powerful, because an importing bundle can easily react to the appearance or disappearance of one or more matching services.

The next question is how do we deal with the generated events? Let's take as an example a Swing program, the Paint application.

> **NOTE** The Paint application is the "official" Apache Felix demonstration application, used to illustrate how OSGi helps to create dynamic and extensible applications. It was written by Richard S. Hall, the founder of the Apache Felix project and co-author of *OSGi in Action*.

The Paint application is a Swing application that allows you to choose shapes from a toolbar and lay them on a painting area. The different kinds of shapes are represented by the `SimpleShape` interface, which has several implementations: `CircleShape`, `SquareShape`, `TriangleShape`, and so on. Figure 6.13 shows the UI of the Paint application.

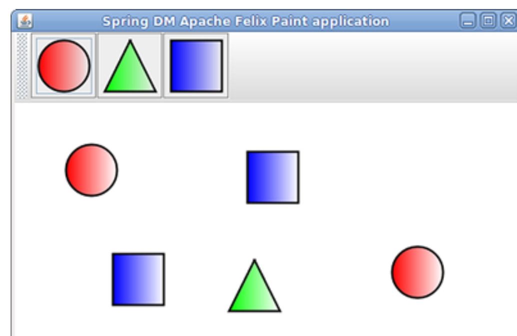

Figure 6.13 The Paint application

The design of the Paint application is simple: a `DrawingFrame` handles the

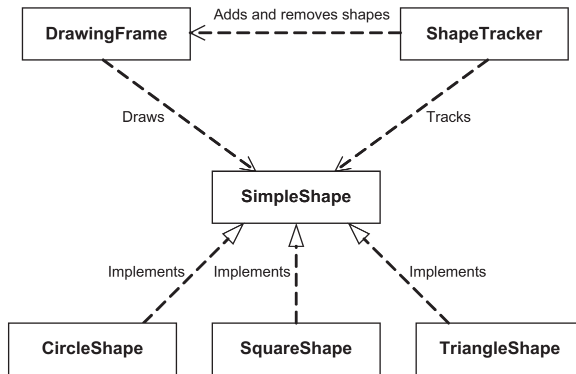

Figure 6.14 The UML design of the Paint application. The `SimpleShape` interface is an extension point, and implementations are then good candidates for being OSGi services. The `ShapeTracker` manages the appearance and disappearance of shapes.

user interaction and the drawing of the shapes, and a `ShapeTracker` tracks the different kinds of shapes available and notifies the `DrawingFrame` of their appearance or disappearance. Figure 6.14 illustrates the design of the Paint application with a UML diagram.

The dynamic part of the Paint application rests in the availability or unavailability of shapes. If a new shape implementation appears in the system, it should be automatically added to the toolbar. Conversely, if a shape disappears from the system, it should be automatically removed from the toolbar. Shapes can be seen as contributions to an extension point and are therefore good candidates for OSGi services. We can easily infer the organization of our application as OSGi bundles, as shown in figure 6.15.

If a bundle wants to contribute to the Paint application, it has to define an implementation of the `SimpleShape` interface and publish the instance in the OSGi registry. Here is the `SimpleShape` interface:

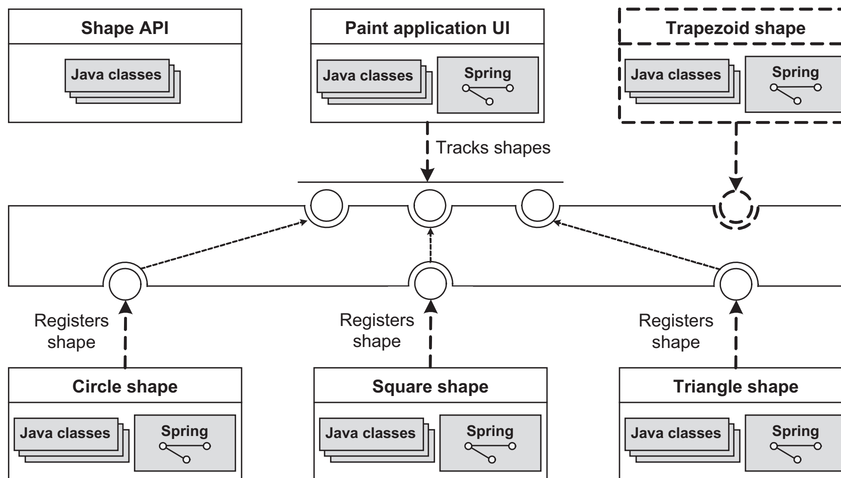

Figure 6.15 The Paint application as OSGi bundles. The `Shape` API bundle exports essential packages; the Paint application UI and `Shape` implementation bundles are Spring-powered.

```
package com.manning.sdmia.paint.shape;

import java.awt.Graphics2D;
import java.awt.Point;

public interface SimpleShape {

  public static final String NAME_PROPERTY =
➥       "simple.shape.name";
  public static final String ICON_PROPERTY =
➥       "simple.shape.icon";

  public void draw(Graphics2D g2, Point p);
}
```

Service property key for the shape name

Service property key for the shape icon

Method to implement to draw shape

The bundle of a SimpleShape implementation leverages the Spring lightweight container to declare the shape as a bean and Spring DM to export the bean to the service registry. Here is an excerpt from the book's code samples, which shows how to export the square shape implementation as an OSGi service:

```
<osgi:service
    ref="squareShape"
    interface="com.manning.sdmia.paint.shape.
➥     SimpleShape">
  <osgi:service-properties>
    <entry key-ref="nameProperty" value="square"/>
    <entry key-ref="iconProperty"
           value-ref="squareIcon"/>
  </osgi:service-properties>
</osgi:service>
```

Exports squareShape bean to OSGi service registry

Defines name and icon property

The bundle of the Paint application is interested in SimpleShape services and wants to know when some are registered or unregistered. It then uses the list tag to import SimpleShape services and the inner listener tag with the callback methods plugged into its ShapeTracker:

```
<osgi:list
    id="shapes"
    availability="optional"
    interface="com.manning.sdmia.paint.shape.
➥     SimpleShape" >
  <osgi:listener ref="shapeTracker"
                 bind-method="addingShape"
                 unbind-method="removedShape" />
</osgi:list>

<bean id="shapeTracker"
      class="com.manning.sdmia.paint.ShapeTracker">
  <property name="drawingFrame" ref="drawingFrame" />
</bean>
```

Imports SimpleShape services

Registers shape tracker as listener

With this configuration, Spring DM calls the addingShape or removedShape method of the ShapeTracker when a shape service is registered or unregistered respectively.

The ShapeTracker will have to do all the dirty work, but OSGi dynamics are no longer part of its concern; it can focus on the update of the UI. Here are the two callback methods, free from any reference to the OSGi API:

```
public class ShapeTracker {
  (...)
  public Object addingShape(
        SimpleShape shape, Map properties) {
    processShapeOnEventThread(ADDED, properties, shape);
    return shape;
  }

  public void removedShape(
        SimpleShape shape, Map properties) {
    processShapeOnEventThread(REMOVED, properties,
      shape);
  }
  (...)
}
```

Bind method (annotates addingShape)

Unbind method (annotates removedShape)

This means that the application will work properly and can be tested outside of an OSGi environment, which is very convenient. Moreover, the application class (the ShapeTracker) is relieved of the burden of OSGi dynamics, because Spring DM handles most of the complex plumbing. Shape services can appear or disappear, and the UI is updated on the fly, as shown in figure 6.16, where the square shape service has been removed from the registry.

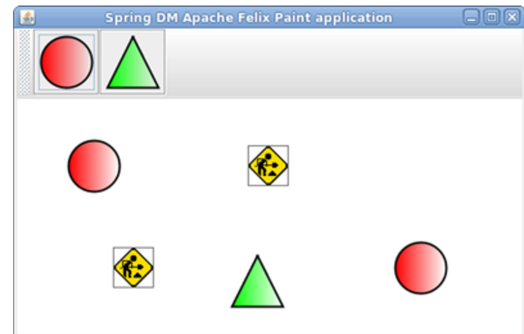

Figure 6.16 The square shape service has been unregistered. It's removed from the toolbar and drawn squares are replaced by "under construction" icons, all of this on the fly.

The Paint application is the perfect example of using OSGi services as an extension mechanism. With the dynamic features of OSGi and a little help from Spring DM, tracking services becomes easy, without any references to the OSGi API.

In the next section, we'll continue with dynamics and see how Spring DM handles the updating of bundles.

6.4.2 Providing a new version of a component

One of the major features of OSGi is its capacity to dynamically update components, without stopping the container. If we take our application in its extreme modular form, we can stop the DAO implementation bundle, install a new version, update the bundle, and restart it without redeploying or updating dependant bundles.

As you saw previously, if calls are made on the DAO service reference, Spring DM lets them block until it reappears, and as soon as Spring DM registers the new version

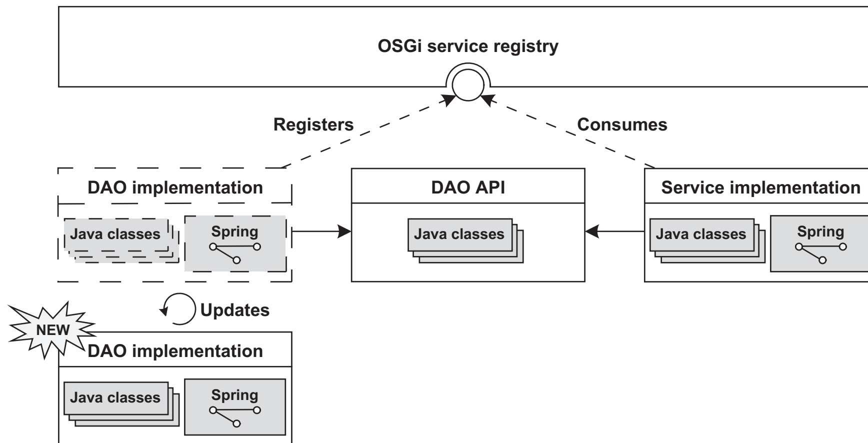

Figure 6.17 **Updating an implementation bundle. With Spring DM, the service update happens transparently, and dependant bundles don't need to be restarted.**

of the DAO, dependant Spring-powered bundles will import the new reference. This is a common update operation, and the dynamic, on-the-fly service replacement works out of the box with Spring DM. This is one benefit of following the pattern of splitting static and dynamic parts of an application. Figure 6.17 illustrates this kind of update.

How can you update a service like this? Let's do it with Equinox. You start by issuing the ss command to find the DAO implementation bundle:

```
osgi> ss

Framework is launched.

id  State       Bundle
0   ACTIVE      org.eclipse.osgi_3.5.0.v20090520
1   ACTIVE      com.manning.sdmia.ch06.directory-datasource_1.0.0
2   ACTIVE      com.manning.sdmia.ch06.directory-domain_1.0.0
3   ACTIVE      com.manning.sdmia.ch06.directory-modular-dao_1.0.0
4   ACTIVE      com.manning.sdmia.ch06.directory-modular-dao-jdbc_1.0.0
5   ACTIVE      com.manning.sdmia.ch06.directory-service_1.0.0
6   ACTIVE      com.manning.sdmia.ch06.directory-modular-service-impl_1.0.0
7   ACTIVE      com.manning.sdmia.ch06.directory-web_1.0.0
(...)
```

As you can see, the DAO implementation bundle is bundle number 4 (shown in bold). Let's stop it and uninstall it:

```
osgi> stop 4

osgi> uninstall 4
```

This is the point at which Spring DM blocks calls on OSGi services that were registered by the bundle and imported by Spring-powered bundles (the business service implementation bundle, for example). You can then install a new version of the DAO implementation bundle and start it:

```
osgi> install
➥     file:./com.manning.sdmia.ch06.directory-modular-dao-jdbc_1.0.1.jar
Bundle id is 44

osgi> start 44
```

As soon as the new version of the bundle registers a `contactDao` OSGi service, Spring DM will send it the blocked calls, so that the waiting incoming requests can be processed.

A less common but still relevant situation is what happens when the DAO disappears but can be replaced. This means that there are one or more OSGi services that can suit the importing bundles. In this case, Spring DM will automatically switch to the next best replacement, observing the filters of the dependant bundles.

This kind of update works well for black-box components, meaning components that don't export any packages and just provide services. These are what we previously called the *dynamic components* of our OSGi applications. But what about the static components of an application? In our DAO-based example, the static part is the DAO API, which the DAO implementation and service implementation bundles depend on. Usually, if we ship a new version of the DAO API bundle, it will come with a new version of the DAO implementation and the service implementation bundles, which benefit from the API updates. Simple update operations won't be enough in this case, because implementation bundles need to be wired to the new version of the API bundle classes. For this wiring to happen, you need to *refresh* the DAO API bundle, and the OSGi framework will handle the refresh of dependant bundles.

To summarize, when updating the DAO API bundle, we'll have to go through these steps:

1. Stop the DAO API, DAO implementation, and service implementation bundles.
2. Install the new versions.
3. Update the bundles.
4. Refresh the DAO API bundle (the OSGi framework will then automatically refresh all the bundles that import Java packages exported by the DAO API bundle).
5. Start the DAO API bundle.

Figure 6.18 illustrates this procedure.

When updating an API bundle, the precise procedure is the hardest part, because Spring DM handles the Spring application context startup order, service registration, and importing. If you carefully design and organize your OSGi application components, Spring DM will manage the complex technical jobs.

That's it for dealing with OSGi dynamics. You now know all the techniques and tricks to benefit from this unique OSGi feature!

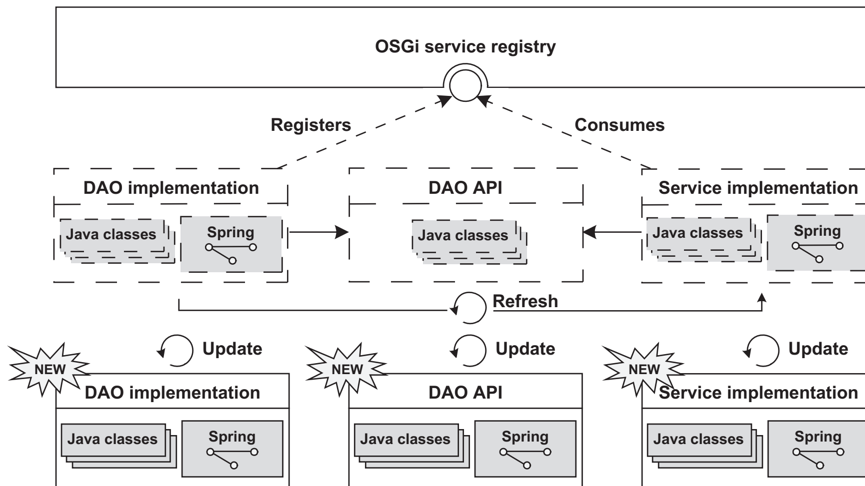

Figure 6.18 Updating an API bundle. It implies updating, but also refreshing, dependant bundles to wire them with the new classes.

6.5 Summary

You should now have a good understanding of the right way to write OSGi-based enterprise applications.

The first step is to have OSGi-compliant JAR files. More and more libraries and frameworks are packaged as OSGi bundles, but alternatively you can rely on OSGi repositories or handle the OSGi-ification yourself, building your own repository or even making your OSGi bundles available to the global OSGi-ification effort.

The second step is to properly design your OSGi applications and understand how to benefit from OSGi specificities. If you're on your own for the design, Spring DM is a great help for filling in the gap between OSGi and traditional application development. It's able to handle transparently most of the dynamics-related issues, and it allows you to retain a POJO-based development style.

We'll continue with enterprise application development, as the next chapter is dedicated to a topic without which enterprise applications wouldn't be the same: access to relational databases with Spring DM.

JAVA/SPRING

Spring Dynamic Modules IN ACTION

Cogoluègnes • Templier • Piper

Spring Dynamic Modules is a flexible OSGi-based framework that makes component building a snap. With Spring DM, you can easily create highly modular applications and you can dynamically add, remove, and update your modules.

Spring Dynamic Modules in Action is a comprehensive tutorial that presents OSGi concepts and maps them to the familiar ideas of the Spring framework. In it, you'll learn to effectively use Spring DM. You will master powerful techniques like embedding a Spring container inside an OSGi bundle, and see how Spring's dependency injection compliments OSGi. Along the way, you'll learn to handle data access and web-based components, and explore topics like unit testing and configuration in OSGi.

This book assumes a background in Spring but requires no prior exposure to OSGi or Spring Dynamic Modules.

What's Inside

- An introduction to OSGi for Spring developers
- How to use Spring with Spring DM
- How to develop enterprise OSGi applications

A Java EE architect, **Arnaud Cogoluègnes** specializes in middleware. **Thierry Templier** is a Java EE and rich web architect. He contributed the JCA and Lucene to Spring. **Andy Piper** is a software architect with Oracle and a committer on the Spring DM project.

For online access to the authors and a free ebook for owners of this book, go to manning.com/SpringDynamicModulesinAction

"A crucial book."
—From the Foreword
by Peter Kriens
OSGi Technical Director

"A uniquely informative book and a vital reference."
—John Guthrie, Sybase, Inc.

"Dynamic modules sans voodoo: the best resource out there!"
—David Dossot
Co-author of *Mule in Action*

"Incredibly useful and accessible...will save you days or weeks of effort!"
—Peter Pavlovich
Kronos Incorporated

"Right book, right time."
—Denys Kurylenko
LinkedIn Corp.

ISBN 13: 978-1-935182-30-6
ISBN 10: 1-935182-30-7

MANNING $59.99 / Can $74.99 [INCLUDING eBOOK]