

Spring Dynamic Modules

IN ACTION

Arnaud Cogoluègnes
Andy Piper
Thierry Templier

 MANNING





MEAP Edition
Manning Early Access Program

Copyright 2010 Manning Publications

For more information on this and other Manning titles go to
www.manning.com

Table of Contents: Spring Dynamic Modules in Action

Part One: Spring DM Basics

Chapter One: Modular Development with Spring and OSGi

Chapter Two: Understanding OSGi Technology

Chapter Three: Getting Started with Spring DM

Part Two: Core Spring Dynamic Modules

Chapter Four: Using Spring DM Extenders

Chapter Five: Working with Services

Chapter Six: OSGi and Spring DM for Enterprise Applications

Chapter Seven: Data Access in OSGi with Spring DM

Chapter Eight: Using Spring DM and Common Web Frameworks to Develop OSGi Web Components

Part Three: Advanced Topics

Chapter Nine: Advanced Concepts

Chapter Ten: Testing with Spring DM Test Support

Chapter Eleven: Support of Compendium OSGi Services

Chapter Twelve: The Blueprint Specification

Appendix A: Developing Spring DM Applications with Eclipse

Appendix B: Developing Spring DM Applications with Maven 2

Appendix C: Using Ant for OSGi Development

Appendix D: Developing Spring DM Applications with the Pax Tools

1

Modular Development with Spring and OSGi

Spring DM – or Spring Dynamic Modules for OSGi™ Service Platforms as it's more properly called – is all about modularity. If you are a Java programmer you have probably heard of, or used, Spring – the dependency injection framework for Java. But what about OSGi? This dynamic module system for Java may be less familiar; but no matter, Spring DM is also about OSGi for the masses, providing OSGi's modularity features in a neat Spring-shaped package that means you don't need to get too involved in the nitty-gritty of OSGi to benefit from its features.

In this book we will describe what Spring DM is, how to use it, and more importantly how to *benefit* from it. Because it's not only about using modular Java systems to get things to work, but it's about getting them to work well. We will also look at some of the implementation challenges with Spring DM, challenges that boil down to OSGi's strict classloading model. For instance using object-relational mapping tools or creating web applications can appear daunting in an OSGi environment, but never fear – we are here to help!

In this chapter, after having reminded you of the concepts of modularity, the specifics of the Spring framework and the features of OSGi, we will show you where Spring DM fits and how its approach and its features simplify the development of standard Java applications in an OSGi environment.

1.1 Java Modularity

We all fall in love with abstraction sooner or later. Abstract data types, polymorphism, and encapsulation – these are all ideas that appeal to the engineers in us and mesh neatly with the old adage of keeping it simple, stupid. No man is an island, however, and code is no different. No matter how beautiful your code – and let's face it we all like to think we write beautiful code – it eventually has to interact with other code; and so begins the slow, usually inevitable, decline into meandering dependencies between code you know is perfect and code you feel should be unnecessary; if it weren't for the fact that you don't quite understand what it does, your boss will shoot you if you break it and anyway who wants to fix somebody else's mistakes.

In this book you will learn how Spring DM and its OSGi substrate can be used to address the problems caused by unmanageable spaghetti code, but before we get to the cool technology, it's worth reviewing what we mean by modularity and the kind of problems modular software is designed to solve.

1.1.1 What is modularity and what is it good for?

Have you encountered any of these issues?

THE BIG, BAD APPLICATION PROBLEM

Your application is 600mb in size and takes 10 minutes just to repackage one java class and 20 minutes to deploy to your favorite application server. When you do finally get it to deploy - and the champagne is all gone - you find you made a mistake and you have to undeploy, fix and redeploy, all outside of office hours – missing the soccer and poker night with your mates.

Probably you have encountered a less extreme version of this – it's amazing how big applications get given time and a team of any significant size. What you want to do, of course, is replace only some subset of the application without having to fully repackage, fully test or fully redeploy, but often this is easier said than done.

THE BRITTLE CHANGE PROBLEM

A high-profile customer of your application reports a problem, you go in and quickly fix the code and try to rebuild the application only to find that the application won't build because some other component is using the internal function you just changed. When you finally get the application built and deployed you start getting reports of other parts of the application now malfunctioning. Working into the night - again missing the soccer - you discover that the change you made had unforeseen consequences that could only be discovered by running the full QA suite – a process that takes 48 hours.

Ok, so most people do a better job of testing before changing a production application, but in today's world of "business at the speed of thought", lengthy testing and hardening cycles often mean lost revenue, lost customers – or both – and what you really want is to be able to make changes to an application knowing that the change will only impact the functionality you want.

THE BUILD THE WORLD PROBLEM

Your development team is distributed in Beijing, Mumbai and San Ramon, each sub-team works on different parts of the application in different timezones and on different development schedules. As the team and application grows you find that it becomes increasingly hard to keep the different parts of the product separate – any time you make a change you find it impossible to tell who is using the functionality you changed and whether they will be affected by your changes. The only solution is to keep all the teams on exactly the same version of the application and to rebuild and retest the entire product anytime any change is made. You eventually spend all of your time building and testing the product and none actually developing it. The company goes bankrupt, you lose your job and you have plenty of time to watch the soccer – if only your widescreen TV hadn't been removed by debt-collectors.

This is similar to the brittle change problem but is a development problem rather than a production / runtime problem.

These problems are all symptoms of un-modular applications. A modular application is in essence one that is divided into several pieces with the connections between the pieces being controlled and well-defined. It is this limit on connections that limits the impact of change and markedly improves things like testability. But what is a "connection", what creates connections and how do you reduce them? The answers to these questions are clearly contingent upon technology, which in our case is Java.

1.1.2 Java – the end and the beginning

Java is great – we would argue it is the best general-purpose programming language ever developed; for it addresses many of the deficiencies of languages that went before it. Java is also a very dynamic language without being slow - a huge leap forward from C++ or Smalltalk - but this flexibility also comes at a price. Writing dynamic programs that load classes on the fly, swapping out code at runtime and trying to keep the whole system functioning is a demanding task that defeats many an able coder. "Connections" between Java components are in the form of class references, either created statically when the program was compiled or dynamically as it runs. Dynamic binding of class references in a Java program also has an additional dimension, due to the fact that classes can be created and loaded

dynamically as well. One can easily see how linkages between classes at runtime - and hence the overall modularity of the system - can quickly become unmanageable.

Usually developers rely on some kind of runtime framework, in the form of an application server, to address these issues for them. But application servers are not interested in helping you write better programs, they are more interested in preventing user code from interfering with their own operation and supporting operational and development requirements for applications that can be installed, uninstalled and reinstalled at runtime. Furthermore the way that application servers do this is usually proprietary, relying on proprietary knobs and descriptors to facilitate the required features.

Java EE acknowledged the need for composite applications by introducing the Enterprise Archive file, but ear files assume a world of EJBs and servlets – a world that most Spring users would reject to a greater or lesser extent. EAR files also only allow you to split your application up into coarse-grained components; they do nothing to enforce the program architecture that you know is required.

1.1.3 Key concepts

In order to explore Java modularity we need to introduce some key concepts as well as revisit some technology concepts that should be familiar. Figure 1.1 illustrates the constituent parts of a modular system.

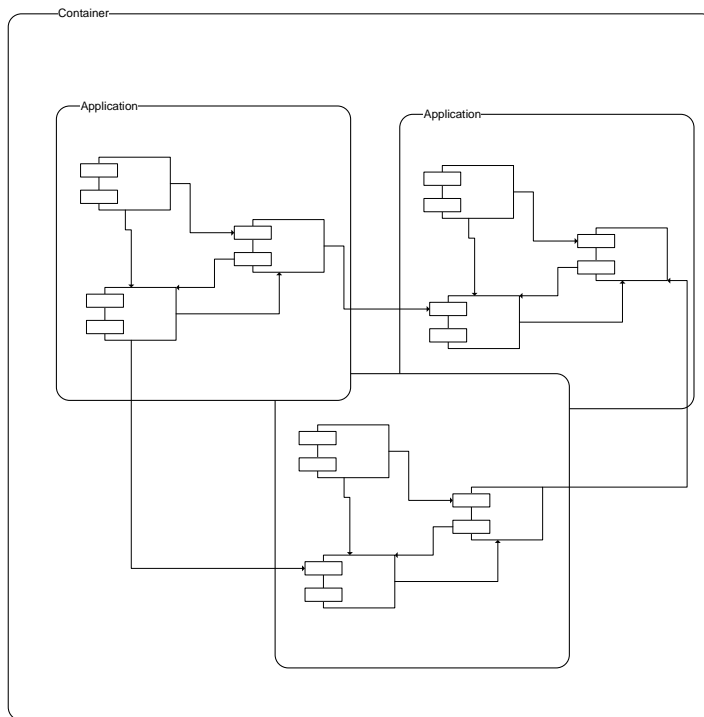


Figure 1.1 Modular Application Structure

- An application is the collection of software components that make up the functional essence of your program including java class files, configuration files, web content and the like.
- An application requires some kind of container in which to run, this could be a web server – such as Tomcat - , application server – such as WebLogic - or something similar.
- An application is composed of a number of coarse-grained modules. Physically these could be individual jar files, war files, ejb jars or similar or some collection thereof. Conceptually modules embody a logical unit of functionality with (hopefully) well-defined APIs and purpose. Modules can be further sub-divided into other modules, software components or similar, but their key quality is that they can be treated as black boxes of software. You don't need to know what's inside – what's

important is their defined perimeter in terms of APIs, inputs and outputs.

- Build-time modularity refers to the relative isolation – or lack thereof – in which a module can be compiled and assembled. The modules required to be present (e.g. because of the classes they provide) before a module can be built are referred to as its build-time dependencies and ideally consist only of APIs. Links between modules that do not consist of APIs can lead to circle of dependencies - where, for instance, A depends on B depends on C depends on A – referred to as a dependency cycle or circular dependencies.
- Run-time modularity refers to the relative isolation of modules at runtime. Run-time modularity is generally more onerous than build-time modularity since modules that only require APIs at build-time generally require implementations of those interfaces at runtime. Implementations generally have their own run-time dependencies, which in turn have their own dependencies and so on. The transitive closure (the set of all required modules) of a module's runtime dependencies can therefore get quite large and might include all modules of the application

You can imagine a modular system a little bit like a fractal diagram or particle physics – software is split into its components parts and those components into others and so forth. There is not really a Higg's boson of Java – unless you count the class; systems can be almost infinitely subdivided as long as it makes sense to do so and as long as the linkages between the parts are minimised. In the next section we discuss a little more why this matters.

1.1.4 Are your applications really modular?

So far, so simple! Why should we care – what's the value proposition of modularity in any shape or form? At stake is building robust, maintainable systems – anyone can write and maintain helloworld, but no system is as simple as helloworld, and many are at the opposite extreme in terms of complexity. The drive towards ever more complex systems is inevitable in the digitally connected world that we now live in, but that complexity is now not something any individual can handle. Just as there are really no “renaissance men” today – the world of science is simply too broad and deep – complete understanding of today's systems is beyond even the most talented. To deal with this problem we use abstractions – from the outside we rationalize about systems using black box components whose implementation we do not need to understand; from the inside we focus simply on the implementation and the required inputs and outputs. Abstractions help us cognitively, but physically they do not guarantee that we can build and maintain the various parts of a system; for that we need modules – building blocks that physically separate the components of a system – allowing us not only to reason about the system without recourse to insider knowledge of various parts but build and run it in the same manner.

So to the question “are your applications really modular”, you should already know the answer – it will be defined by the degree of pain you feel when trying to make changes – or the degree of slippage you experience when trying to develop new functionality! If you don't know then the answer is almost certainly “no”, and that may not matter now if you are a lone developer or part of a small team, but beware – small programs have a funny way of getting bigger quickly, and it's much easier to keep things in order than it is to untangle them in the first place.

So by now you should understand what we mean by modularity and appreciate the need for modularity in Java systems. But, practically speaking, how do we make Java systems more modular? To answer this question we need to first look at how many modular Java systems are being built today, namely using the Spring framework, what modularity features it provides and what features we need to look further afield for.

1.2 The Spring framework

Spring is a layered application framework and lightweight container, the foundations of which are described in the book *Expert One-on-One J2EE Design and Development* by Rod Johnson. The Spring project itself started in 2003. The lightweight container and the *aspect-oriented programming (AOP)* system are the main building blocks of Spring. Besides these, Spring provides a common abstraction layer for transaction management, integration with various persistence solutions (plain JDBC, Hibernate, JPA) as

well as with Java enterprise technologies (JMS, JMX). Spring even provides its own *Model View Controller (MVC)* Web framework; Spring MVC. The Spring framework is not an “all-or-nothing” solution: one can choose the modules according to their needs, the lightweight container being the glue for the application and the Spring classes.

The Spring Framework is now widely used in Java enterprise applications and well documented in books like *Spring in Action* by Craig Walls. We'll see in this section the building blocks of the Spring Framework: its lightweight container that makes dependency injection a breeze, its support for aspect-oriented programming and for the development of enterprise applications. We'll start immediately by the POJO based model the Spring Framework promotes and how it changes the development of Java enterprise applications by allowing writing more loosely coupled components.

1.2.1 POJO based development

The acronym POJO stands for *Plain Old Java Object*. A POJO is a regular Java object, without any dependencies on framework interfaces or classes. POJOs are often cast as a lightweight alternative to EJBs - at least prior to EJB 3.x - since they do not require developers to implement special interfaces. Even though EJBs in 1.x/2.x tried to promote modular and reusable code, their APIs inevitably leaked into business code, making the use of an EJB container mandatory.

Using POJOs has many advantages - no technical background is needed, ease of testing and better reusability – to name just a few. Given these obvious advantages what drove the undoubtedly bright inventors of EJBs? Well, enterprise applications have recurring technical, or *platform*, requirements (for instance database access and transactions, asynchronous programming and so on) that can be fulfilled by a *container*. The container can relieve the programmer from this implementation burden by managing the components they write: creation, destruction, transactions between method calls... Some would say that EJB 1.x/2.x promoted the right principle, but just did not find the right balance; the technology was too intrusive and too heavyweight in many cases.

Frameworks made POJOs popular by giving them some of the technical services an EJB container could traditionally provide. Object-relational mapping tools like Hibernate gave them high performance persistence. Lightweight containers like Spring gave them powerful configuration features, nearly unlimited power with aspect-oriented programming. All of this in a standard Web container or even in a standalone Java application.

The POJO approach was adopted by Java EE in the last few years with the emergence of standards such as EJB 3.x or JPA. Enterprise components can now be plain Java classes, with annotations typically used to describe their behavior or to give configuration instructions. With POJOs you don't only get rid of technical dependencies, but you also get the possibility to design your application exactly as you need, making the decoupling of your classes easier. That's what we'll see in the next section.

1.2.2 Loose coupling of classes

Loose coupling is the first step on the road that leads to true modular programming. Any object oriented system is made up of components, interacting with each other. They must be as independent as possible, otherwise one change in the system can trigger cascading changes. Imagine one of your business services needs to notify other components that a user has been created. The following snippet illustrates a tightly coupled solution (you should not do this!):

```
public class BusinessServiceImpl implements BusinessService {
    private JmsNotifier jmsNotifier;                                #A

    public void createUser(User user) {                            #B
        ...                                                         #C
        jmsNotifier.notify("User created");
    }
}

#A Declare class-based property
#B Create user (not shown)
#C Notify
```

Why can the previous snippet be considered tightly coupled? The reason is that the notification technology used is concretely defined using the `JmsNotifier` property. This tells us that the `BusinessService` cannot be used or even tested without a JMS container. Re-use of the `BusinessService` with another type of notification technology or even without notification at all will not be an easy task.

The `BusinessService` can be easily decoupled from the notification by introducing a `Notifier` interface instead of using an implementation. The following snippet shows this solution.

```
public class BusinessServiceImpl implements BusinessService {
    private Notifier notifier;                                #A

    public void createUser(User user) {
        ...                                                  #B
        notifier.notify("User created");                    #C
    }
}
```

#A Declare interface-based property
#B Create user (not shown)
#C Notify

In this snippet the notification concept is embodied by the `Notifier` interface, without any reference to the underlying technology. Thanks to this abstraction, the `BusinessService` can now be easily reused with any kind of notification (email, JMS...), the only requirement being to implement the `Notifier` interface. Figure 1.2 shows how we evolved from a tightly coupled solution to a loosely one.

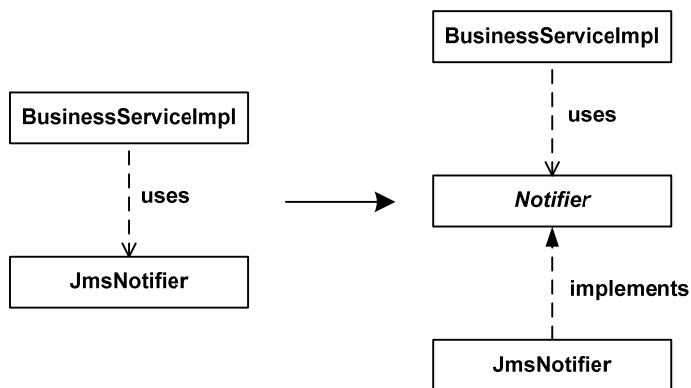


Figure 1.2 From a tightly to loosely coupled solution. But now, how to get the right implementation?

Now in `BusinessService`, the notification concept became a simple dependency. The next problem is to assign the right `Notifier` implementation. With a straight instantiation in `BusinessService`, we lose all the benefits of using an interface:

```
public class BusinessServiceImpl implements BusinessService {
    private Notifier notifier = new JmsNotifier();
    ...
}
```

A common pattern is to delegate the dependency creation and initialization to another object:

```
public class BusinessServiceImpl implements BusinessService {
    private Notifier notifier = NotifierFactory.create();
    ...
}
```

With this solution the `BusinessService` is no longer tied to any `Notifier` implementation and must actively *retrieve* its dependencies from a third object. We usually say that the object has to *lookup* its dependencies – typically in Java EE this is accomplished using a naming technology such as JNDI. Figure 1.3 is a UML representation of our new solution.

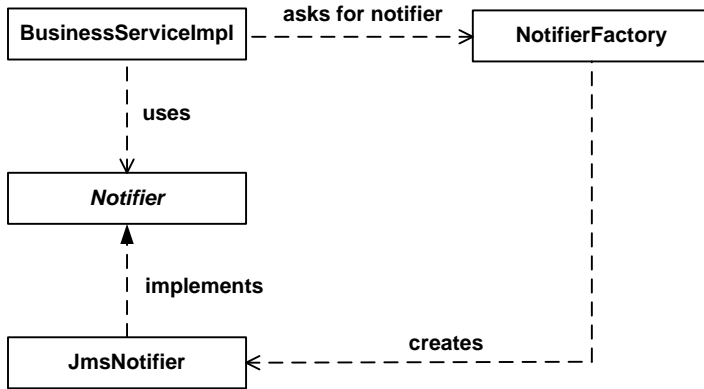


Figure 1.3 The lookup solution – implies a dependency toward a factory

Although we can achieve a certain degree of loose coupling with the lookup solution, it does not scale well. In large applications, dependencies can represent hundreds of object and wiring them all with factories quickly becomes a nightmare (just imagine figure 1.3 with hundreds of business services, even more dependencies like the notifier and the corresponding factories!). Moreover, the `BusinessService` directly depends on a third object to retrieve its dependency. This factory (or service locator or object registry or whatever you want to call it) now becomes itself a dependency and the service cannot work properly without it. POJO zealots would even say that the `BusinessService` is no longer a POJO as the `NotifierFactory` is just an awkward artifact to handle the `Notifier` assignment.

One way to get rid of this dependency wiring problem is to let a third system assemble all of the application components. This leads us to dependency injection.

1.2.3 Dependency injection

Dependency injection is all about creating, configuring and wiring components. The system in charge of assembling the components is called a lightweight container. With a container managing their dependencies for them, components do not need to concern themselves with lookups. Rod Johnson calls it the *Hollywood Principle* – “don’t call us, we’ll call you”! Figure 1.4 illustrates how this container can manage and wire together our components. Notice that the container API does not leak to our application classes.

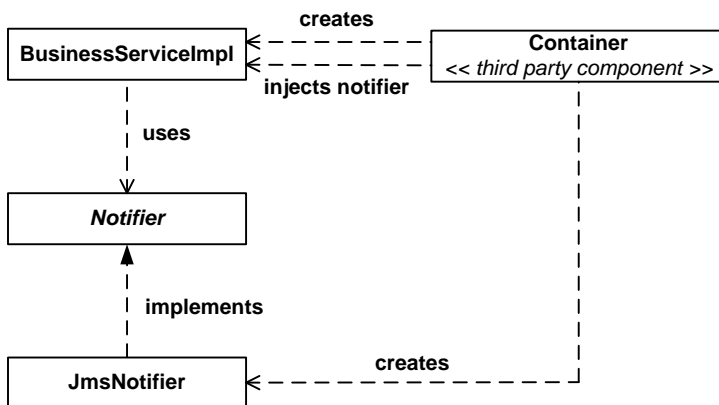


Figure 1.4 Dependency injection with a lightweight container – applications classes do not depend any more on a building component like a factory

Typically, a number of ways exist for configuring the component wiring; ranging from XML files to Java annotations, or even plain text. The Spring lightweight container offers a rich set of features to do the

wiring, the following snippet illustrates the XML-based configuration for our `BusinessService` and its `Notifier` dependency.

```
<beans (...)>
  <bean id="notifier" class="foo.bar.notify.JmsNotifier" /> #A

  <bean id="businessService" #B
    class="for.bar.service.impl.BusinessServiceImpl"> #B
    <property name="notifier" ref="notifier" /> #C
  </bean>
</beans>
```

#A Declare Notifier

#B Declare business service

#C Inject Notifier into business service

The container takes care of each step of a component's creation (instantiation, wiring), leaving it with no dependencies on any callable class or interface such as a factory. In Spring terminology, the object repository is called the *context*.

Annotations can alleviate the XML verbosity, but using them ties the annotated class to the Spring API, as shown in the following snippet.

```
import org.springframework.beans.factory.annotation.Autowired; #1

public class BusinessServiceImpl implements BusinessService {
  @Autowired #2
  private Notifier notifier; #2
  ...
}

<beans (...)>
  <bean id="notifier" class="foo.bar.notify.JmsNotifier" />

  <bean id="businessService" #3
    class="for.bar.service.impl.BusinessServiceImpl" /> #3
</beans>
```

Cueballs in code and text

The `BusinessService` now needs to import the Spring `Autowired` annotation (#1). The `notifier` property is annotated with it (#2). The Spring lightweight container uses this information to automatically inject into the `BusinessService` a `Notifier` object declared in the Spring context. Spring uses a default strategy to automatically resolve dependencies: it tries to autowire by type. If several `Notifiers` are declared in the context, Spring cannot know which one to choose and the context startup will fail. Fortunately we have only one `Notifier` object in the XML configuration so Spring will reliably resolve the `BusinessService` dependency. The `BusinessService` declaration at #3 is shorter than in the previous listing as the explicit dependency injection is not needed anymore, thanks to the autowiring.

Dependency injection is a simple yet rather powerful pattern. Combined with interface-based programming, it allows for more testable and less tightly coupled code. Component management (creation, wiring) can then be delegated to lightweight containers such as Spring. The Spring lightweight container can be a great help in your OSGi development, as it help in building the inner elements of your modules and Spring DM will take over to make them interact easily with the OSGi platform.

With full control over the component lifecycle, lightweight containers can do much more than just wiring them up together. We will see this in a second with aspect-oriented programming.

1.2.4 Aspect-oriented programming (AOP)

In traditional object-oriented programming (OOP), components are loosely coupled and each of them has its own responsibilities. Nevertheless components often require additional features beyond the basic ones they are meant to provide. These features are usually related to system services, such as security, logging or resource management, and are called *cross-cutting concerns* because they tend to cut across components or application layers. You can deal modularly with these cross-cutting concerns by using aspect-oriented programming and transparently relieve your OOP components from dealing with them

directly. In AOP, the unit of modularity is called an *aspect* and deals with one cross-cutting concern (the unit of modularity in OOP is the class). Figure 1.5 illustrates the transition from a pure object-oriented solution to a solution using both object- and aspect-oriented programming. Notice how the business code is tangled with cross-cutting concerns such as logging and security in the 100% object-oriented solution. By using AOP, cross-cutting concerns are deported to dedicated aspects and application code does not have to deal with them anymore.

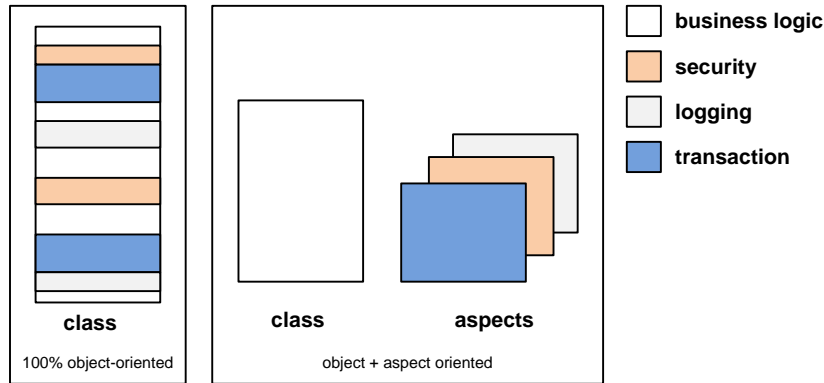


Figure 1.5 AOP is complementary to object-oriented programming

With AOP components one can focus on their core tasks and then the aspects can be easily reused in any situation. In this way AOP is not meant to replace OOP but rather complements it.

By managing components, Spring is able to apply aspects to them. The action of applying aspects is called *weaving*. Different ways of weaving exist, the simplest being using a proxy in place of the target component. Proxy weaving can be achieved with standard Java features, as long as you stick to interface programming. As each call to the target is intercepted by the proxy, any aspect can be called before or after the call is delegated to the target.

To demonstrate AOP with Spring, let's modularize the notification when a user is created in our business service. This notification is a cross-cutting concern and therefore should be taken out of the service and included in a dedicated aspect. The code of this aspect is illustrated in listing 1.1.

Listing 1.1 The notification aspect

```
package foo.bar.aop;

import foo.bar.notify.Notifier;

public class NotifierAspect {
    private Notifier notifier;

    public void userCreated() {
        notifier.notify("User created");
    }

    public void setNotifier(Notifier notifier) {
        this.notifier = notifier;
    }
}

#A Delegates notification to the Notifier
```

The aspect takes care of the notification, as long as an appropriate `Notifier` is injected. The code corresponding to the notification in the `BusinessService` is removed: not more `notifier` property and no more explicit call to the `Notifier`. Listing 1.2 shows the corresponding XML configuration, which leverages the Spring AOP configuration facilities.

Listing 1.2 AOP configuration with Spring

```

<beans (...)>

  <bean id="businessService"                                #1
        class="foo.bar.service.impl.BusinessServiceImpl" /> #1

  <bean id="notifier" class="foo.bar.notify.JmsNotifier" /> #2

  <bean id="notifierAspect" class="foo.bar.aop.NotifierAspect"> #3
    <property name="notifier" ref="notifier" /> #3
  </bean> #3

  <aop:config>
    <aop:aspect ref="notifierAspect"> #4
      <aop:after-returning #5
        pointcut="
execution(* foo.bar.service.impl.BusinessServiceImpl.createUser(..))" #6
        method="userCreated"/> #7
      </aop:aspect>
    </aop:config>

</beans>

```

Cueballs in code and text

The business service declaration at #1 no longer has a dependency as the notification has been totally modularized. A JMS-based `Notifier` is declared at #2 and injected into the aspect (#3). These are traditional component declarations in Spring. The aspect weaving starts at #4 with the aspect element, referencing the aspect by its name, `notifierAspect`. The notification should be launched *after* the `createUser` method call on the business service, hence the use of the `after-returning` element (#5). Spring needs to know on which components to apply the aspect. This is the goal of the black magic formula at #6 which defines the `createUser` method using AspectJ pointcut syntax. The `method` attribute at #7 refers to the aspect method to call when the aspect is triggered.

Some would say “Wow! What a waste of energy for a tiny, little notification!” Well, the example is rather simplistic but you need to realize that the notification has been totally externalized from the business service. With a little extra-work we could build a more adaptable notification aspect and reuse it as-is in other situations.

AOP pushes modularization a step further, but in a different way than traditional OOP does. AOP appears tricky at first glance but, in a Spring-managed environment, it is just a matter of XML (or annotation) configuration. The Spring Framework brings AOP out-of-the-box, and as Spring DM builds a bridge between the Spring Framework and OSGi, you’ll get AOP automatically in your OSGi developments. Now, as you saw the main benefits of the Spring lightweight container, we’ll study what the framework can bring to your enterprise developments.

1.2.5 Enterprise support

Besides its lightweight container and AOP features, the Spring framework provides support for a number of Java/Java EE standards and frameworks. Why does it provide this support, are these libraries not self-sufficient? Configuring and combining them in an application can turn out to be tricky, largely due to their configuration subtleties. Spring is able to handle configuration for the most common cases, leaving you with very simple configuration steps to perform. If you are more comfortable with the target framework or need some particular tuning, all the regular settings are still available through Spring configuration.

Many enterprise frameworks deal with connection or resource handling. In the best case, a managed environment provided by a full featured application server can handle this typical boilerplate and error-prone code but often one must handle it by hand. This results in dedicated “plumbing” code, with all the flaws that come with it: hard to maintain and integrate, never thoroughly tested and barely reusable. Application components end up being littered with this boilerplate code which has nothing to do with their core task.

Spring support, then, is particularly useful in the data access area, with its common abstraction layer for transactions and its template-based approach to persistence. Spring allows for declarative transaction management in various situations ranging from managed environment with JTA to native database transaction with plain JDBC. Declarative transaction management is made possible through Spring AOP support and can be applied to any kind of component. It has no impact on application code and switching from one environment to another is just a matter of configuration. Application components can then focus on their primary task and do not even know they are fully transactional.

Transaction management in Spring is tightly integrated with data access support. Spring provides a common template-based approach for popular data access solution (JDBC, Hibernate, JPA...). These templates are in charge of the resource management (connections for JDBC, sessions for object-relational mapping tools etc) and expose a handy API to the developer. Nevertheless, Spring templates are not compulsory and you can stick to the tool's native API, whilst still gaining benefits of a managed environment.

Transaction management and data access are only a few of the common use cases; Spring support extends to number of other technologies, like illustrated in figure 1.6.

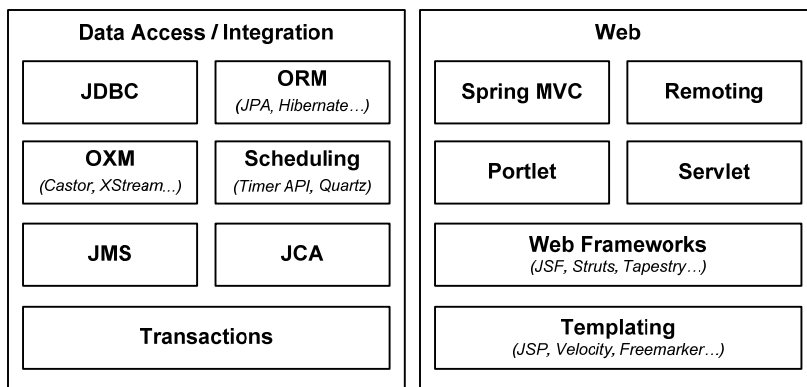


Figure 1.6 Spring enterprise support

Spring and its philosophy should no longer be a stranger to you, and now you should surely understand that Spring is the first step towards modularity. We'll see in the next section how the OSGi technology pushes the Java platform to its boundary to offer *true* modularity.

1.3 A new approach to modular development with OSGi

As described at the beginning of the chapter, we assert that modularity and reusability are key issues in the development of any reasonably sized application but the Java platform does not provide complete support at this level. OSGi is a technology that makes it possible to tackle these problems head on for Java applications. OSGi relies heavily on Java's different features and operating mechanisms, and in particular the class loader feature.

1.3.1 Aims of OSGi

Designed from the ground up to be lightweight and dynamic, OSGi is supported and standardized through a body called the OSGi Alliance. The technology is relatively old and mature having been created in 1999, targeting embedded Java and networked devices then extended to support mobile devices. This was key to its subsequent development since, from the beginning, it had to take into account the constraints of the host environment in terms of memory and resources and had to provide a platform lightweight enough to work on these systems.

Since 2004 the technology has gained significant adoption from the open source community thanks mostly to its adoption by the integrated development environment Eclipse in order to handle its plug-in architecture. In the same period, the technology has also been integrated into middleware stacks such as

the application servers WebSphere and Jonas in order that they might benefit from dynamic management of OSGi components.

Subsequently the use of the OSGi technology has moved into server applications themselves. This aspect enables a better organization of the different parts of these applications, to combine several different versions and improve their availability, since runtime updates can now be possible.

So we have seen where OSGi technology comes from and that it has become more and more popular through recent years even for Java developers working on Web applications; but questions remain; What actually *is* OSGi and how can this technology help us in our application development?

- The first part of the answer can simply be found in the short description of the technology provided by the OSGi Alliance, “OSGi - The Dynamic Module System for Java”. The two keywords here are dynamic and module. The key concept of OSGi is how it addresses the modularity issue in a dynamic way for the Java platform.
- It also provides strong module foundations and also offers a new approach to modularize developments based on the component concept and addressing these limitations. Let’s now dive more into the concepts, features and mechanisms of the technology.

There is a real lack of solutions to address these aspects within Java and regular Java applications offer an incomplete or insufficient level of modularity. OSGi addresses this deficiency in four ways:

- Modularity with an extensibility mechanism;
- Visibility and isolation between modules;
- Module and dependency versioning;
- Dynamic handling of modules.

1.3.2 OSGi layers

In order to address these concerns OSGi provides a standardized environment which is divided into different layers, as shown in the figure 1.7. Each layer is responsible for addressing a specific feature of the OSGi container corresponding to the entity which manages components. Every OSGi container is based on the Java virtual machine and specifically implements three layers that enable support for dynamic modularity.

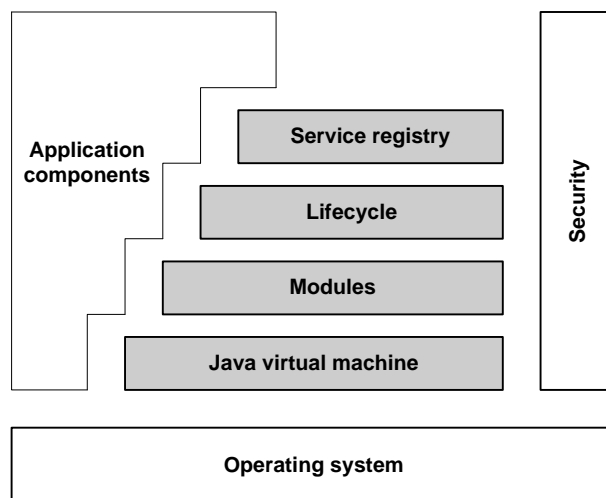


Figure 1.7 OSGi layered architecture

MODULE LAYER

The first container layer, the modules layer, is responsible for handling components. Its main goal is to provide modularity based on the Java platform. The key building block for this layer is the module or

component which in OSGi is called a *bundle*. To ensure robust modularity, OSGi provides certain features and characteristics which are described in the remainder of this section.

The first characteristic of the module layer is strict classloader isolation. As a matter of fact, this layer allows every artifact in a bundle to be hidden. By default, a component is really a black box with OSGi. If you want to make things visible, you need to specify them explicitly and, conversely, when using classes, the required packages must be specified as being visible before you can reference them. This ensures that the provider has strict control over what is seen by other components. Providers and consumers of classes can additionally specify what version of a class is being used and this makes it possible to use simultaneously two conflicting versions of the same library or framework.

To achieve this, OSGi does not use the classic hierarchical classloader mechanism but prefers a dependent classloader approach. As such, each bundle is linked to a dedicated classloader which interacts with those of other bundles. This approach is called *classloader chaining*.

In order to achieve this OSGi does not introduce any additional Java-based entities, instead the packaging of bundles is based on standard JAR files and their contained MANIFEST file. The format of the manifest file already supports arbitrary meta-data and this is used to specify OSGi-specific attributes for a bundle. At this level, the OSGi extends Java by adding specific meta-data which describe the behavior and properties of the bundle. In the manifest, the first thing that is usually specified is related to the component itself, as shown in the following snippet.

```
Manifest-Version: 1.0 #1
Bundle-Name: Simple Spring DM Bundle #2
Bundle-Version: 1.0 #3
Bundle-ManifestVersion: 2 #4
Bundle-SymbolicName: com.manning.spring.osgi.simple #5
```

#1 Supported version of MANIFEST file (not OSGi specific)

#2 Human readable name of OSGi component

#3 Version of OSGi component

#4 Supported version of MANIFEST file for OSGi headers

#5 Component identifier used by the OSGi container

This layer also handles the linkage between modules in a controlled way. The technology allows you to configure which packages you want to make visible in order to let other components use their constituent classes. Then, on the importing side we have to specify which packages are used by components. At this level, OSGi provides support for versioning modules at runtime. When one bundle uses another, you can specify which version of the bundle to use; particularly convenient if several versions are present in the container. This is configured in the MANIFEST file of bundles using a dedicated directive for dependency configuration. The advantage of this is that the OSGi platform can verify by itself the consistency of all your bundles and advise you when problems occur. It has the knowledge and the responsibility for managing dependencies. The following snippet shows how to configure the provisioning and the use of packages.

```
Manifest-Version: 1.0
Bundle-Name: Spring OSGi Bundle
Bundle-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-SymbolicName: com.manning.spring.osgi.simple
Export-Package: com.manning.spring.osgi.simple.service #1
Import-Package: com.manning.spring.osgi.utils #2
```

#1 Makes specified package visible to other components

#2 Specifies component uses classes from this package

LIFECYCLE LAYER

The lifecycle layer gives OSGi dynamic support for module management. In the component lifecycle bundles transition through different states. Broadly speaking there are two different kinds of states, one kind related to the installation and configuration of bundles in the container and the other related to the runtime behavior of bundles. Figure 1.8 shows the different possible states for bundles.

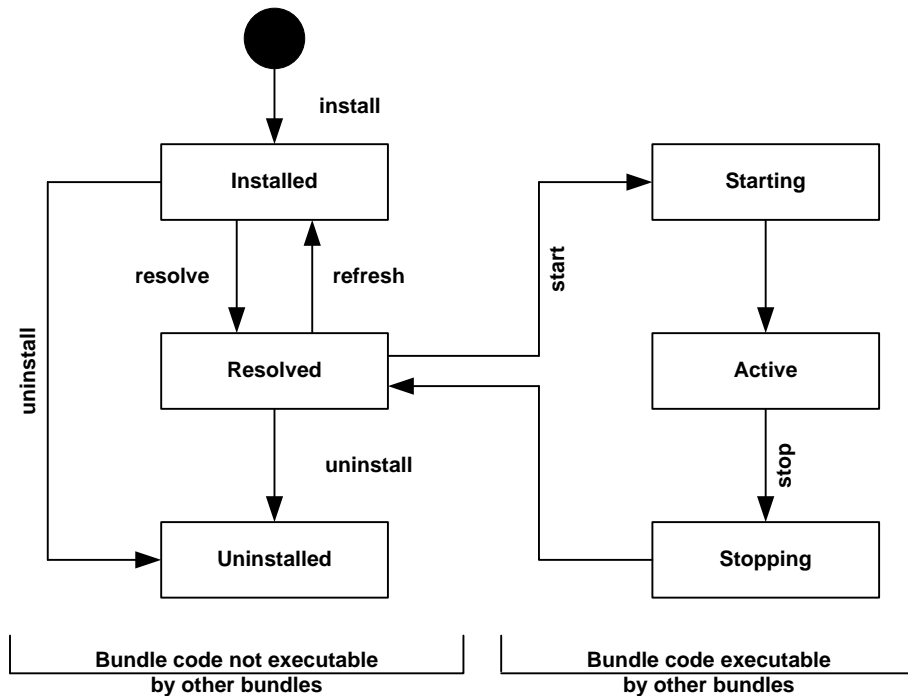


Figure 1.8 Different states of bundles

The lifecycle layer relies on the module layer for classloading and provides a dynamic approach to bundle management making it possible to update parts of an application without stopping it. All these states will be described in the section 2.1.4, “OSGi component lifecycle”, of chapter 2.

In addition, it also provides an API which makes it possible to interact with the bundles’ lifecycle programmatically. This API is usable by every active bundle in the OSGi container.

SERVICE REGISTRY LAYER

The last layer, the service registry, is an important part of OSGi since it provides a service-oriented feature. Through the service registry OSGi offers the ability to register one or several access points to components through services. This feature is based on the service registry of OSGi which allows bundles to be used and interact with each other in a way that takes the dynamic nature of the system into account. In fact the module layer providing class importing and exporting is not directly compatible with dynamically adding and removing bundles. The service based approach is the only way to make secure, dynamic interactions between bundles possible. The service registry provides a number of events that can be used to handle the coming and going of services.

In this model, services are POJO and interface driven programming is recommended to describe their contracts. You don’t have to be tied to the OSGi API. The following class can easily be used to implement an OSGi service, as shown in the following snippet.

```

public class SimpleServiceImpl implements SimpleService {
    public void executeSomething(String parameter) {
        (...)
    }
}

```

At this level, OSGi provides an interesting feature that enables you to hide the implementations of services based on the strict classloader of bundles. The only thing to do is to define interface and implementations in different packages and only make the interface package visible. This means that interfaces can be loosely coupled with their corresponding implementations. Thus, the consumer of a service does not need to have the knowledge of both the service provider and the way it implements the service.

Figure 1.9 shows the service feature of OSGi and their possible interactions with the service registry.

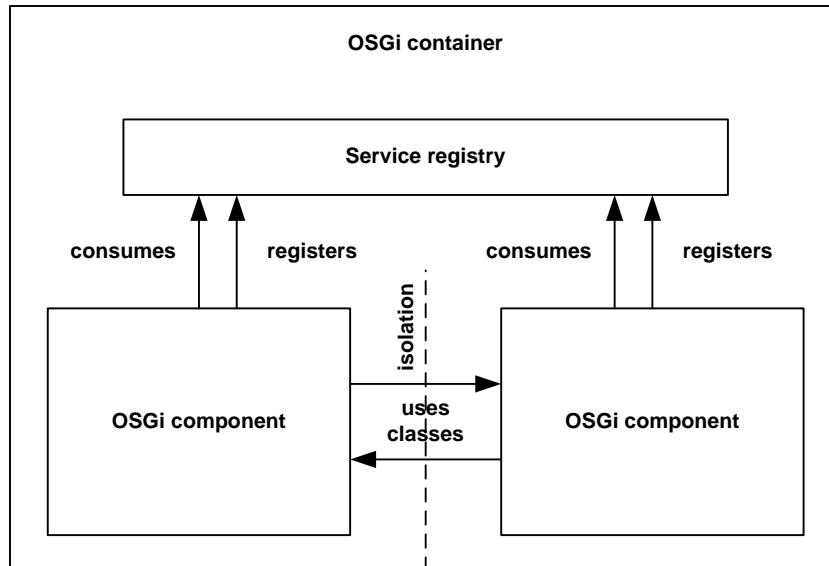


Figure 1.9 related to services and their possible interactions with the service registry

To summarize, OSGi provides a foundation for developing component and service based solutions. It implements components through a strict class loader and dependency versioning scheme. It also provides a platform following all the principles of service oriented architectures in a particularly lightweight way and directly within a Java virtual machine.

The strength of the solution is that OSGi enables the implementation of all these mechanisms in a dynamic way. Indeed, the platform provides the mechanisms necessary to manage components at runtime and to dynamically interact with the service registry to both publish and consume services without stopping and restarting it. OSGi offers a lightweight alternative to monolithic Java EE applications and provides better flexibility for these kinds of applications. These dynamic aspects create new issues since application features must now take into account the varying presence of entities or services.

We discuss in detail the principles of OSGi in chapter 2. It's also worth noting that the new version 4.2 of the OSGi specification includes additional features of relevance to us. In particular the Blueprint Service is introduced which aims to standardize the concepts and mechanisms of Spring DM. We describe Blueprint in more details in chapter 12.

The OSGi technology fills the lack of modularity in Java applications by providing a complete and lightweight platform to implement component based and service oriented applications within a Java virtual machine. It provides a real add-on to improve modularity and structuring of this kind of applications while maintaining the freedom to implement internal processing of components.

1.4 Using Spring in an OSGi environment with Spring DM

We described in the previous section the characteristics of Spring and OSGi, and saw that they address a complementary set of features. These steps are mandatory to understand where Spring DM fits in to the mix and now we are ready to see how it's possible to use Spring technologies in an OSGi environment.

1.4.1 What is Spring DM?

Despite the benefits of the Spring framework described in section 1.2, it suffers from several drawbacks in particular use cases. The first one occurs when trying to scale up to large and complex applications. In this case, a lot of beans need to be configured within the Spring container making its configuration more difficult to maintain. In addition, Spring suffers from a lack of modularity and provides no real support to improve this. Although you can divide your configuration into several XML files, they all serve to configure a single application context and the alternative – integrating several Spring containers - can be difficult to achieve.

Another limitation of Spring is the static nature of its dependency graph configured using dependency injection. The problem arises because the links between beans are created once, mainly at the start-up of the container, and there is no built-in support to update them at runtime. When updates occur, the whole application needs to be restarted in order to completely reload the dependency graph and take the updates into account. This problem with static coupling is made worse by issues with the instantiation order of beans. Spring computes a complex dependency graph to determine the order in which to instantiate beans and then inject them. Developers don't hand over this processing and debugging it can be particularly complex with a lot of dependencies. Furthermore there is very limited support for circular dependencies putting an even heavier burden on developers to configure their beans correctly.

On the other side of the table the OSGi core specification does not provide any support for modern in patterns and tools in the design and implementation of bundles. This is actually by design leaving the developer the freedom to choose a suitable architecture and the framework. Thus, using a framework like Spring can really enhance the power of the platform since it enables the use of dependency injection, AOP and other modern paradigms. However, "here be dragons" since there are obvious challenges in managing the disparate lifecycle's of both Spring and OSGi bundles.

Another challenge in using OSGi relates to the explicit configuration and dynamic behaviour of services. Services are essential in creating any OSGi application since they are the only safe way to access functionality across bundle boundaries. However, using the service management API is tedious and managing the dynamic nature of services in user code, extremely error prone.

Finally, OSGi technology doesn't really provide a convenient way to simply implement, deploy and undeploy Web bundles within an embedded Web container.

We can see that the two technologies appear to be mutually complementary with the whole being potentially greater than the sum of the parts. Spring DM provides a powerful modular solution to develop Spring and Web-based applications that can be deployed in an OSGi execution environment. Thus, the aim of the technology is both to make Spring and OSGi work together in a simple way as well as addressing the limitations of the two technologies that we have just described. The figure 1.10 shows where Spring DM fits and how it is the bridge between components, service registry and embedded Web containers. This figure gives a high level overall picture of how Spring DM both allows the embedding of the Spring container within OSGi components and the use of Web technology inside an OSGi container.

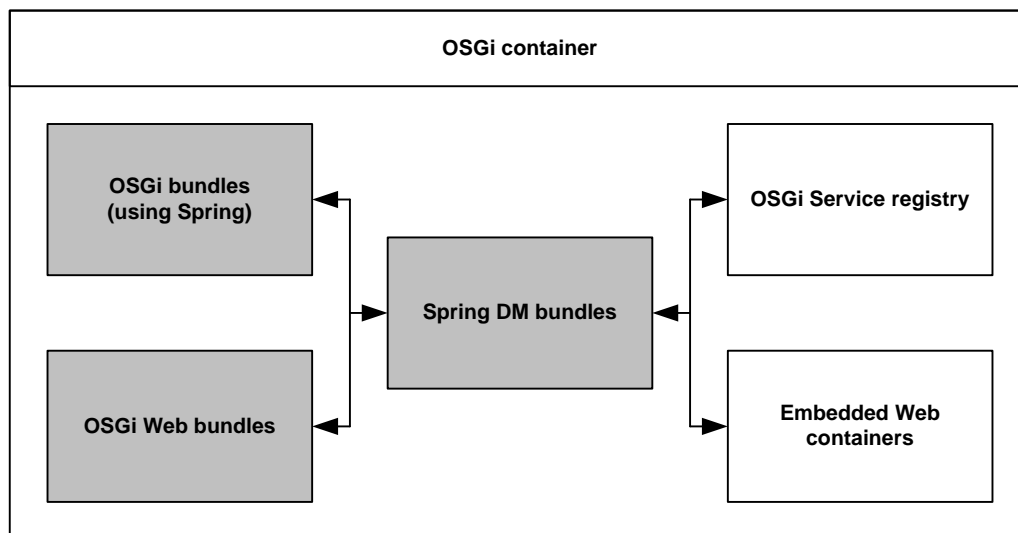


Figure 1.10 Interactions between Spring DM bundles and other elements present in the OSGi container

SPRING DM AND SPRINGSOURCE DM SERVER

We should mention in passing that Spring DM is *not* SpringSource dm Server, although the names sound similar. dm Server is a full-stack application server that leverages Spring DM and OSGi as its modular kernel, but provides many more features on top of these including tooling, deployment and management.

1.4.2 Embedding Spring within an OSGi container

The first aim of Spring DM is to allow Spring based applications to be deployed as OSGi bundles. This allows for true application modularity whilst still harnessing the power of the Spring framework. Spring DM also provides a semantically rich and user-friendly way of linking components together by declaratively providing and consuming OSGi services. All these aspects allow applications to take advantage of the dynamic nature of OSGi.

Another important thing to understand is that you are not tied to a specific tool when developing your OSGi components, not even Spring DM! You can interact with existing bundles using a variety of different technologies. Since links between bundles are based on the OSGi-standard service registry, the service consumer doesn't know – and doesn't care - what technology was used in the creation of the service.

In fact, Spring DM addresses several different issues in order to make it easier to use Spring within an OSGi container in order to improve the modularity of Spring-based applications:

- Enforce module boundaries and add dynamics at runtime;
- Detection and management of Spring-powered and Web-powered bundles;
- Transparent configuration, starting and stopping Spring containers inside Spring powered bundles;
- Facilities related to service management and use;
- Transparent handling of the dynamic aspects of OSGi.

In the same spirit as the design of the Spring framework, Spring DM provides a programming model based on best practices and patterns that enable you to use OSGi technology in an optimal and efficient way. It also provides a more convenient, easy and transparent way of implementing the technology, hiding the complexity of handling dynamic services.

SPRING MANAGEMENT WITH SPRING DM

Let's now concentrate on how Spring DM makes it easy to work with Spring within OSGi. In a classical Spring application, a dedicated Spring container is used and the framework is configured from its enclosing contexts (e.g. from the classpath or Web environment). The framework also provides a hierarchical relationship between application contexts whilst tying them together. With Spring DM, things work slightly differently. A dedicated Spring container is associated with each Spring DM powered bundle. Each container is unconnected with the rest and is really internal to its enclosing bundle. Logically speaking having separate containers is mandatory since each bundle has its own lifecycle and can appear or disappear at any time.

Spring DM manages all of these containers in a convenient and non-intrusive way. No code or configuration is required inside a bundle to configure, start or destroy a Spring container. All that is required is that Spring DM is installed in the OSGi container and that the bundle includes a Spring application context configuration file. Spring DM then monitors the bundle lifecycle in order to determine when to trigger the appropriate action. Once Spring DM is deployed inside an OSGi container, it can automatically detect Spring-powered components based on the presence of the Spring application context configuration file as well as through certain dedicated headers in the MANIFEST file of the bundle. When a bundle with the appropriate configuration is started, a Spring container is also configured and embedded inside the component. Figure 1.11 shows a summary of all these mechanisms.

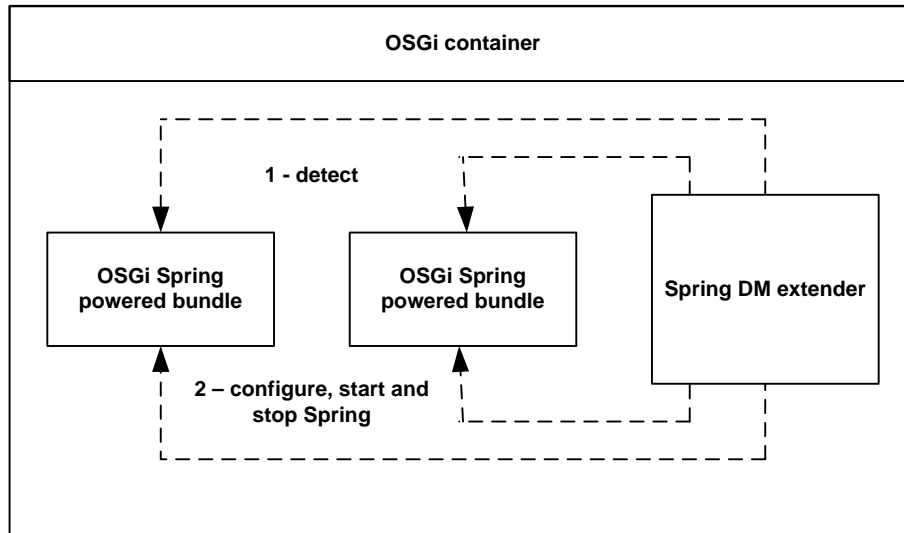


Figure 1.11 Management of classic bundles by Spring DM

OSGi SERVICE SUPPORT WITHIN SPRING DM

As well as managing the creation of Spring containers Spring DM also provides a convenient way to link bundles together using the OSGi service registry. In any OSGi application simply using package imports and exports is not sufficient since these simply specify the behaviour of class *linkages* within the application, they do nothing to manage the lifecycle of object *instances* created from these classes. This is what makes the OSGi service registry so essential, it provides a well-defined boundary through which object instances can be accessed while still maintaining appropriate class and lifecycle boundaries. Think of it a bit like the old English class system – servants were never allowed to speak directly with the master of the house, instead they had to interact through the butler or other go-between! It did not matter what the servants looked like, or even whether they were the same servants from one day to the next; the facilities provided to the master of the house were always the same, and hidden behind the austere demeanour of Jeeves or his counterpart. Where Spring DM improves upon the services of the butler – or the service registry in our case – is that it provides a very easy and declarative way to configure and reference services from within the Spring configuration itself, even allowing dependency injection of artefacts from the service registry.

The heart of this configuration is a dedicated Spring 2 XML namespace. Thanks to this mechanism, simple POJOs can be exposed as OSGi services and OSGi services can be injected into regular Spring beans simply by using the appropriate elements. Listing 1.3 shows how to expose a bean named `myBean` as an OSGi service using the `<osgi:service>` element. At the start-up of the corresponding Spring container, the service is automatically registered - and unregistered when it is destroyed.

Listing 1.3 Registering a Spring bean in the service registry with dependency injection

```

<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:osgi="http://www.springframework.org/schema/osgi"
       (...)>

  <osgi:service ref="myService"
               interface="com.manning.spring.osgi.simple.MyService"/> #1
  <bean id="myService"
        class="com.manning.spring.osgi.simple.impl.MyServiceImpl"/> #2
</beans>
  
```

- #1 Exposes `myService` bean as an OSGi service
- #2 Defines a classic POJO within the Spring XML configuration

Listing 1.4 shows the other side of the service coin - referencing and using an OSGi service in a classic Spring bean through the use of the `<osgi:reference>` element. The example shows a service named

com.manning.spring.osgi.simple.MyService being made available as a bean called myService and then being injected into myBean.

Listing 1.4 Referencing an OSGi service within Spring configuration

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:osgi="http://www.springframework.org/schema/osgi"
       (...)>

    <osgi:reference id="myService"
                  interface="com.manning.spring.osgi.simple.MyService"/>      #1
    <bean id="myBean" class="com.manning.spring.osgi.simple.MyBean">
        <property name="service" ref="myService"/>                             #2
    </bean>
</beans>
```

#1 Configures a reference to an OSGi service with myService identifier

#2 Injects the OSGi service reference into the service property

We should notice that the Spring DM support for OSGi services is heavily dependent on OSGi since it provides the underlying infrastructure for implementing services. Also of note is the fact that despite services being *dynamic*, the references provided to beans are *concrete*. However, because the underlying services are indeed dynamic the static dependency tree of a classical Spring application has been removed. We will describe later how this works. Finally note the deliberate mistake – the service interface and its implementation class are both in the same package – and we will also describe later why this might be a bad idea.

Another issue in OSGi development is how to efficiently implement Web applications. Spring DM provides a mechanism to handle components implementing this kind of application.

1.4.3 Web support

Spring DM also supports another special kind of bundle which contains a Web application. For the remainder of this book we shall call them *Web bundles*. In this case, Spring DM is responsible for deploying or undeploying the Web application by interacting with the Web container embedded in an OSGi container. An application of this kind adheres to the structure of a classic Java EE Web application but can use now all the mechanisms provided by OSGi; such as classloader isolation, dependency management and dynamic services. In this instance, the Web bundles do not have to use Spring since this feature is primarily a way to manage Web applications in an OSGi environment. Spring DM delegates the task of handling the application to the Web container. The latter then manages them as regular Java EE Web applications. Although Spring is not required we can choose to use the Spring framework as one might commonly do in a non-OSGi environment. Figure 1.12 shows the interactions of the different elements in the context of the Web support of Spring DM.

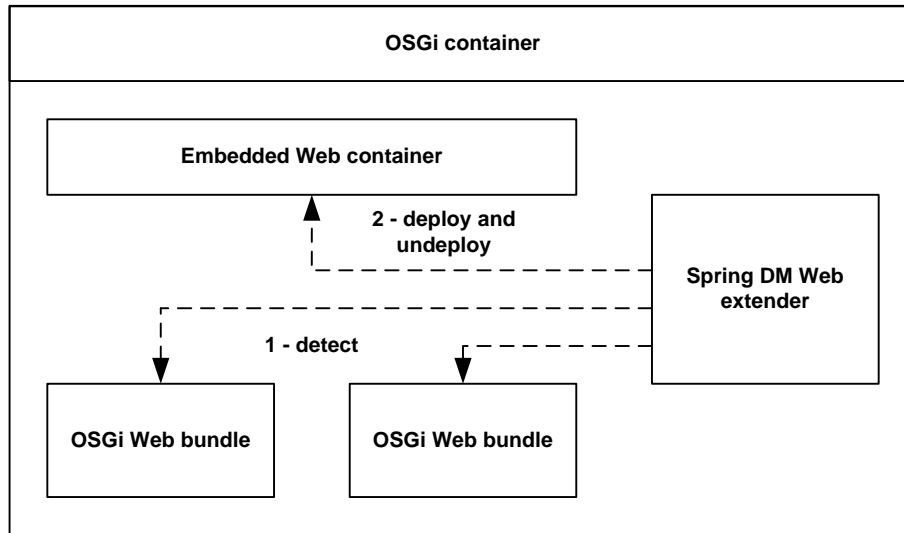


Figure 1.12 Mechanisms of the Web feature of Spring DM

1.4.4 Benefits of Spring DM for real-life OSGi applications

As we shall see, developing proof of concept applications with OSGi is not really that difficult; however, things are different with real Java applications. As applications get more complicated, and especially as they start to use third-party libraries, you'll quickly encounter unexpected errors due to the strict classloader and runtime dependency management of OSGi. However, never fear! Spring DM is at hand to rescue us from some of the more painful issues.

STRUCTURING INTERNAL BUNDLE PROCESSING

Spring DM aims to keep the simple and familiar Spring programming model for enterprise developers but still make it possible to exploit all the features of the OSGi platform. Industry-proven techniques such as dependency injection and aspect-oriented programming can be easily used to improve and simplify an application's structure. So while OSGi addresses modularity at the component level, Spring DM makes it possible to reuse all the mechanisms of the Spring framework *inside* bundles. In addition, enterprise developers can use all of the enterprise features that Spring provides that they are familiar with such as Web, data access and transactions.

USING LIBRARIES AND FRAMEWORKS INSIDE BUNDLES

On the other hand, if you want to use your favourite libraries and frameworks within an OSGi environment, you are required to use artefacts which contain MANIFEST files with the correct configuration for OSGi. The problem is that not many libraries are not provide "OSGi-ready" so to speak. To make it easier to use libraries and frameworks of this nature, SpringSource provides a dedicated repository which offers a large set of OSG-ified libraries and frameworks. In it you can find all your favorite libraries ready to be deployed to your OSGi container. SpringSource has already done the, sometimes painful, job of converting them for the OSGi world. The repository also has a web interface which allows you to browse the repository and is compatible with modular build tools such as Maven2 or Ivy.

UNIT TESTS

Finally Spring DM also provides the pieces necessary to easily develop both unit and integration tests based on standard frameworks such as JUnit and TestNG. Spring DM automates the creation of a dedicated bundle for the tests, provides a very convenient way to resolve dependent bundles from a local repository or tools like Maven and Ivy and automatically installs and runs all of the pieces inside an OSGi container.

Now that we have introduced all the background of Spring DM and the technologies it leverages, it's time to see how to implement your first application.

1.5 Spring DM Hello World

We will stick to old programming traditions by providing a Hello World sample using Spring Dynamic Modules. The example consists of declaring a simple Java bean in a Spring application context and letting Spring DM handle the context creation when the bundle is deployed. The bean will emit the 'Hello World' message at its creation. You will see that Spring DM handles most of the dirty work - application context discovery and bean creation - and that we'll only have to write the bean class and create XML configuration files.

Before starting the sample, you should take a look at table 1.3, which gives an overview of the various versions of Spring DM and their corresponding requirements and main features.

Table 1.3 Overview of Spring DM's versions

Version	Requirements	Supported OSGi platforms	Description
2.0	Java 1.5+ Spring 3.0	OSGi R4 versions 4.0, 4.1, 4.2 Equinox 3.5.x Felix 2.x Knopflerfish 3.x	Blueprint (RFC 121) Reference Implementation.
1.2	Java 1.4+ Spring 2.5.6	OSGi R4 versions 4.0, 4.1 Equinox 3.3.x Felix 1.4.x Knopflerfish 2.2.x	Support for Compendium Services.
1.1	Java 1.4+ Spring 2.5.5	OSGi R4 versions 4.0, 4.1 Equinox 3.2.x Felix 1.0.x Knopflerfish 2.1.x	Web support.
1.0	Java 1.4+ Spring 2.5.1	OSGi R4 versions 4.0, 4.1 Equinox 3.2.x Felix 1.0.x Knopflerfish 2.0.x	Standard extender for embedding a Spring application container in an OSGi bundle.

You can choose the version that best suits your target environment (version of Java or Spring or supported OSGi platform). If you do not have any constraint, you should choose Spring DM 2.0 (the latest version at the time of this writing) as it provides most features and benefits from the latest bug fixes. This is the version we'll be using all along this book. Nevertheless, the Hello Sample should work with any version.

1.5.1 Provisioning the OSGi container

First we have to provision OSGi with all the bundles required by Spring DM. Most of the bundles are in the Spring DM distribution, which you can download from <http://www.springsource.org/osgi>. Once the download is complete, create a `hello-world-springdm` directory and copy `org.eclipse.osgi-3.5.1.R35x_v20090827.jar` from the `lib` directory of Spring DM distribution.

What does Eclipse, the famous Java Integration Development Environment, have to do with an OSGi Hello World example? Well, a lot, actually. Eclipse is *based* on OSGi and is built upon Equinox, the Eclipse Foundation OSGi container. The `org.eclipse.osgi-3.5.1.R35x_v20090827.jar` file contains Equinox and not the whole Eclipse IDE.

To launch Equinox, open a command line, go to the `hello-world-springdm` directory and type the following command:

```
java -jar org.eclipse.osgi-3.5.1.R35x_v20090827.jar -console
```

An `osgi` command prompt appears. You can type the `ss` command (for short status) to learn about the container state and the installed bundles:

```
osgi> ss

Framework is launched.

id      State      Bundle
0       ACTIVE    org.eclipse.osgi_3.5.1.R35x_v20090827

osgi>
```

The only bundle is the container itself; it is now time to provision it with the Spring DM bundles. Type `exit` to stop Equinox.

Create a `bundles` directory at the root of your project and copy all the necessary bundles listed in table 1.3.

Table 1.3 OSGi bundles for the Hello World project

JAR file	Origin
<code>org.springframework.aop-3.0.0.RC1.jar</code>	Spring DM distribution lib directory.
<code>org.springframework.asm-3.0.0.RC1.jar</code>	Spring DM distribution lib directory.
<code>org.springframework.beans-3.0.0.RC1.jar</code>	Spring DM distribution lib directory.
<code>org.springframework.context-3.0.0.RC1.jar</code>	Spring DM distribution lib directory.
<code>org.springframework.context.support-3.0.0.RC1.jar</code>	Spring DM distribution lib directory.
<code>org.springframework.core-3.0.0.RC1.jar</code>	Spring DM distribution lib directory.
<code>org.springframework.expression-3.0.0.RC1.jar</code>	Spring DM distribution lib directory.
<code>com.springsource.slf4j.api-1.5.6.jar</code>	Spring DM distribution lib directory.
<code>com.springsource.slf4j.org.apache.commons.logging-1.5.6.jar</code>	Spring DM distribution lib directory.
<code>com.springsource.slf4j.nop-1.5.6.jar</code>	SpringSource Enterprise Bundle Repository
<code>com.springsource.org.aopalliance-1.0.0.jar</code>	Spring DM distribution dist directory.
<code>spring-osgi-core-2.0.0.M1.jar</code>	Spring DM distribution dist directory.
<code>spring-osgi-extender-2.0.0.M1.jar</code>	Spring DM distribution dist directory.
<code>spring-osgi-io-2.0.0.M1.jar</code>	Spring DM distribution dist directory.

All bundles can be found in the Spring DM distribution except for one, the NOP SLF4J implementation. You can download it from the SpringSource Enterprise Bundle Repository at the following URL:

```
http://www.springsource.com/repository/app/bundle/version/download?name=com.springsource.slf4j.nop&version=1.5.6&type=binary
```

When you first launched Equinox, it created a configuration directory where it keeps runtime information between executions. We will now tell Equinox which bundles it has to install each time it starts. In the configuration directory, create a `config.ini` file with the following content:

```
osgi.bundles=bundles/org.springframework.aop-3.0.0.RC1.jar@start, \
bundles/org.springframework.asm-3.0.0.RC1.jar@start, \
bundles/org.springframework.beans-3.0.0.RC1.jar@start, \
bundles/org.springframework.context-3.0.0.RC1.jar@start, \
bundles/org.springframework.core-3.0.0.RC1.jar@start, \
bundles/org.springframework.expression-3.0.0.RC1.jar@start, \
```

```

bundles/spring-osgi-core-2.0.0.M1.jar@start, \
bundles/spring-osgi-extender-2.0.0.M1.jar@start, \
bundles/spring-osgi-io-2.0.0.M1.jar@start, \
bundles/com.springsource.org.aopalliance-1.0.0.jar@start, \
bundles/com.springsource.slf4j.api-1.5.6.jar@start, \
bundles/com.springsource.slf4j.nop-1.5.6.jar, \
bundles/com.springsource.slf4j.org.apache.commons.logging-1.5.6.jar@start

```

Be careful to enter all the trailing backslashes (\) at the end of each line and add @start at the end of each filename. Equinox is now properly provisioned, you can start it and check if all the bundles have been resolved and some of them started:

```
osgi> ss
```

Framework is launched.

id	State	Bundle
0	ACTIVE	org.eclipse.osgi_3.5.1.R35x_v20090827
1	ACTIVE	org.springframework.aop_3.0.0.RC1
2	ACTIVE	org.springframework.asm_3.0.0.RC1
3	ACTIVE	org.springframework.beans_3.0.0.RC1
4	ACTIVE	org.springframework.context_3.0.0.RC1
5	ACTIVE	org.springframework.core_3.0.0.RC1
6	ACTIVE	org.springframework.expression_3.0.0.RC1
7	ACTIVE	org.springframework.osgi.core_2.0.0.M1
8	ACTIVE	org.springframework.osgi.extender_2.0.0.M1
9	ACTIVE	org.springframework.osgi.io_2.0.0.M1
10	ACTIVE	com.springsource.org.aopalliance_1.0.0
11	ACTIVE	com.springsource.slf4j.api_1.5.6 Fragments=12
12	RESOLVED	com.springsource.slf4j.nop_1.5.6 Master=11
13	ACTIVE	com.springsource.slf4j.org.apache.commons.logging_1.5.6

The Spring DM extender (bundle 8) now keeps an eye on each bundle installation to gently create Spring application contexts for those powered by Spring DM. Exit from Equinox and get ready to write your first Spring DM OSGi bundle.

1.5.2 Writing the Spring DM powered bundle

Our first bundle will be minimalistic as it will contain only one Java class and the necessary configuration files (JAR manifest and Spring DM XML configuration). Create a `src` directory at the root of the project and create the following Java class (be careful to create it in the corresponding directory, following Java package conventions):

```

package com.manning.sdmi;

public class HelloWorld {

    public HelloWorld() {
        System.out.println("Hello world, I'm being created!");
    }

}

```

This class is obviously not useful but it will emit our Hello World in the console when Spring DM instantiates it, which is the point of this exercise! Create a `bin` directory and compile the class:

```
javac -d bin src/com/manning/sdmi/HelloWorld.java
```

OSGi bundles are standard JAR files, with additional headers in the manifest. The OSGi container uses these headers to identify the bundle: identity, description, version... In the `src` directory, create a `META-INF` directory with a `MANIFEST.MF` file in it. The manifest has the following content:

```

Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: Spring DM Hello World
Bundle-SymbolicName: com.manning.sdmi.helloworld
Bundle-Version: 1.0.0
Export-Package: com.manning.sdmi

```

Note the empty last line of the manifest! On top of the description headers (Bundle-Name, Bundle-SymbolicName and Bundle-Version), the manifest contains an `Export-Package` header, which tells

the OSGi container to make the corresponding package available to other bundles. By this means, bundles will be able to import classes from our bundle package.

So far, so modular; our bundle is a usual OSGi bundle and we will never see our Hello World message if we deploy it as-is in Equinox. We want Spring DM to create an instance of our HelloWorld class and this implies creating a configuration file in the location expected by the Spring DM extender. Create a `spring` directory in `META-INF` and create a `hello-world-context.xml` in it. Listing 1.5 shows the content of the file.

Listing 1.5 Spring DM Hello World bundle configuration file

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"           #A
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"       #A
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans.xsd"> #A

  <bean id="helloWorld" class="com.manning.sdmi HelloWorld" />      #B

</beans>
#A Declare Spring Core namespace
#B Declare Hello World bean
```

Those familiar with Spring will see no difference between declaring the `helloWorld` bean in an OSGi environment and declaring it in a Spring-based application, mainly because there is no difference at all. Every feature of the Spring framework, dependency injection, AOP, utilities classes, are available and can be leveraged to properly initialize our bundle.

We have now all the pieces of our bundle, we only need to package it before installing in Equinox. Copy the whole `META-INF` directory to the `bin` directory and use the `jar` program to create the archive:

```
jar cvfm helloworld.jar src/META-INF/MANIFEST.MF -C bin .
```

This creates a `helloworld.jar` file, ready to be deployed in Equinox.

1.5.3 Deploying the bundle

We are about to install our bundle into Equinox and see how Spring DM handles it. Launch Equinox and type `install file:helloworld.jar` to install the bundle. Equinox answers by giving the bundle id in the runtime environment, so the console should output something like:

```
osgi> install file:helloworld.jar
Bundle id is 14
```

The bundle id in your environment can be different from what you see in this text. Remember this id because it will be used in subsequent commands. Check the bundle status by launching the `ss` command, our bundle appears at the end and is in the `INSTALLED` state:

```
14      INSTALLED  com.manning.sdmi.helloworld_1.0.0
```

Bundle installed but no hello world on the console? We did not miss anything, our bean has not been created yet, as the bundle needs to be started, which will trigger Spring DM extender and our bundle Spring context creation. Start the bundle and you should see the long expected message (use the correct bundle id after the `start` command):

```
osgi> start 14
```

```
osgi> Hello world, I'm being created!
```

Congratulations! You've just finished the Spring DM hello world sample! This rather simplistic example shows one of the strengths of Spring DM: we are able to interact with the OSGi container without manipulating any OSGi API, everything is done declaratively. This follows the Spring philosophy: the framework handles technical concerns to let the developer concentrate on the application code.

1.6 Summary

All Java applications eventually have to deal with issues of modularity both at build time and at runtime. Making regular Java SE or Java EE applications modular, however, is not an easy task and really needs something else to do the heavy lifting. That "something else" is OSGi. OSGi provides an open and lightweight framework for implementing truly modular Java applications.

Despite the power of this technology, it remains difficult to use thanks to both the strictness of its classloader isolation model and its lack of support for modern application frameworks such as Spring. All is not lost however, since Spring DM is available to address these issues, making it easier to develop Java applications based on the Spring framework and Web applications in an OSGi environment. The power of both the Spring framework and OSGi are harnessed by Spring DM in order to provide a rich framework for developing component-based and service oriented applications. Spring DM simply allows you to embed a Spring container inside OSGi components and manage these containers using the standard OSGi lifecycle. Each component, or bundle, can now use dependency injection, aspect oriented programming and other enterprise support provided by the framework like in any regular Java application outside an OSGi container.

In the next chapter, we will describe the key concepts of OSGi in a Spring DM perspective. Since the tool runs inside an OSGi container, you need to understand how this platform works and become comfortable with OSGi mechanisms. The next chapter will help you understand the features and mechanisms of Spring DM.