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 10

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

10 Testing with Spring DM

This chapter covers

- The concepts of testing, in both standard and OSGi environments
- When and how to use Spring DM's OSGi mocks for unit testing
- When and how to use Spring DM's support for OSGi integration testing

Testing is one of the most important activities in software development, not only because it ensures better quality in the end product, but also because good testing techniques can make a developer more productive. We introduced Spring DM's testing support early on in chapter 3 because manually testing the behavior of OSGi components in a target environment can be cumbersome, mainly because of tasks such as provisioning. You discovered then that Spring DM provides support for bootstrapping an embedded OSGi container and running JUnit test classes in it. This early coverage gave you the basics necessary to test your own OSGi bundles, and proved also that OSGi applications aren't particularly special in this regard: they can also be tested. In this chapter, we'll cover all the techniques and the tools that you'll need to test your OSGi components, and hence make your OSGi applications more reliable.

As software testing has its own vocabulary, this chapter starts by briefly describing the different kinds of tests (unit, integration, and system tests) and how they fit together. We'll then give some guidelines for testing Spring-based applications. This will give us enough of a basis to explain how OSGi applications (Spring-powered or otherwise) should be tested using Spring DM's testing support.

We'll then move on to cover unit tests and integration tests. Unit tests are for testing classes in an isolated manner. Most of the time, classes have dependencies on other classes or APIs (like the OSGi API), and these dependencies must be simulated by using mock objects (objects that mimic the behavior of real objects in a controlled way). That's where Spring DM can help, as it provides ready-to-use OSGi mocks. Automating integration tests for OSGi components is a difficult task, because it consists of testing how different OSGi components behave in an OSGi container. This implies bootstrapping and provisioning an OSGi container, and running the test instance *within* the OSGi container. Fortunately, Spring DM provides powerful and flexible testing support that takes care of all these steps. This testing support also leverages Spring and Spring DM features (like dependency injection and declarative interaction with the OSGi environment) and makes them available for use by the test classes. This is another incarnation of the bridge that Spring DM provides between the OSGi and the Spring worlds.

By the end of this chapter, you'll know everything you need to know about the practical testing of OSGi applications, and you'll have all the techniques necessary to test your OSGi components efficiently, thanks to the testing facilities that Spring DM provides. You'll also be able to take advantage of the hooks that Spring DM testing support provides for Spring-based OSGi applications.

Let's start with an overview of testing in software development.

10.1 Testing OSGi components with Spring DM

Testing OSGi applications isn't much different than testing standard applications, as long as you know the appropriate techniques and tools. So before diving into OSGi-specific testing practices, we'll remind you of the different kinds of tests we encounter in software development. We'll provide guidelines for unit-testing Spring-based applications, and this will give us enough of a basis to understand how to organize the testing of Spring DM applications.

10.1.1 General concepts

There are three different types of tests that are commonly encountered in software development:

- *Unit tests*—These verify and validate an individual programming unit (a class in an OOP language like Java). As a programming unit usually relies on other programming units, we usually resort to mocks and stubs to test it in isolation.
- *Integration tests*—These verify and validate that several programming units work as expected in collaboration.
- *System tests*—These verify a whole system by assembling all its components together (not covered in this book).

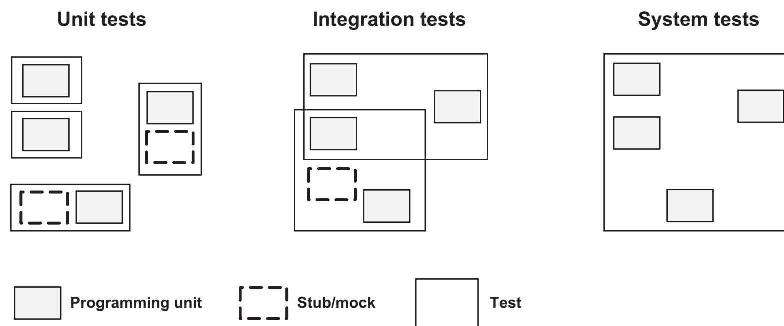

Figure 10.1 Unit tests, integration tests, and system tests are usually run sequentially, because testing programming units in collaboration is easier when they all behave as expected individually.

Testing programming units in collaboration is very difficult if they don't behave as expected individually. That's why it's vital to run unit tests before integration and system tests. Figure 10.1 shows how an application composed of four programming units can be tested. It reads from left to right to illustrate the order of testing.

We mentioned mocks and stubs, but we didn't really explain them. Mocks and stubs help with testing a programming unit in isolation by simulating the collaborators of the tested unit.

- *Stubs* are usually developed for one specific test and provide canned answers to the (limited) set of calls they're supposed to respond to. They can also record information about calls that the test uses for verification.
- *Mocks* are preprogrammed to answer to a sequence of calls and are usually able to tell if the sequence occurred correctly during the test.

Mock object libraries can be generic (like JMock or EasyMock), allowing you to create a mock object from any interface or class and dictate its behavior. They can also be specific to a particular technology, such as OSGi (like the mock object library provided by Spring DM), networking, database, and so on. Generic mock object libraries are very useful for mocking application classes (DAO, business services, and the like) but they can become cumbersome to use for mocking the same set of classes (like those specific to a technology) over and over. In contrast, specific mock libraries are more tailored.

Now that we know more about testing practices, let's look at some guidelines for testing Spring-based applications.

10.1.2 Unit tests with Spring-based applications

Thanks to its POJOprogramming model and its lightweight container, applications based on the Spring Framework are easy to unit test. Strict unit tests (which involve a class that doesn't use any dependencies) can be written with plain test frameworks like JUnit or TestNG. When testing a class that needs some dependencies (such as a Hibernate-based DAO, which needs a `SessionFactory` and so a `DataSource`), the Spring

lightweight container is there to help assemble these different components together with an appropriate configuration (such as a test-dedicated `DataSource`). By bootstrapping a Spring application context for testing the classes of a module, you can benefit from the wiring features of the lightweight container but also from features like transaction management, AOP, and so on. Figure 10.2 illustrates the organization of tests in a Spring-based module.

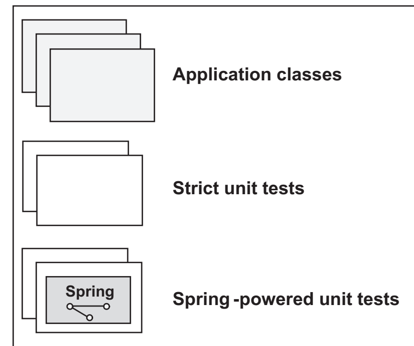

Figure 10.2 In a Spring-based module, strict unit tests involve classes that don't use any dependencies. Tests can also embed a Spring application context when wiring or enterprise features are needed. Because Spring DM promotes a POJO programming model, these tests shouldn't imply the use of OSGi, even if the module is meant to be used in an OSGi environment.

> **NOTE** When a test implies complex operations, like bean wiring with Spring, transaction management, or AOP, we can already speak of integration tests. This terminology is a matter of point of view (the notion of a programming unit changes according to the view we adopt). From the module's point of view, we can call such tests integration tests, because we test the integration of inner components. From the whole system's point of view, such tests are internal to a module, whereas integration tests concern interactions between a set of modules.

What's the best way to write tests that leverage the Spring Framework? You could do it by hand—by bootstrapping a Spring application context at the beginning of the test—but you'd end up writing the same sequence of code for all your tests. The Spring Framework integrates well with test frameworks like JUnit and TestNG, thanks to the Spring `TestContext` Framework, which offers features like these:

- Automatic bootstrapping of a Spring application context for a test
- Caching of contexts if they're reused by several tests (to speed up test execution)
- Dependency injection of Spring beans in the test instance
- The possibility to react to the lifecycle of the test (to inject test data into a database before test execution, for example)

Coverage of the Spring `TestContext` Framework is out of the scope of this book, but let's take a quick peek at how it's used with the following code snippet (from the code samples for this chapter):

```
package com.manning.sdmia.directory.dao.jdbc;

import junit.framework.Assert;

import org.junit.Test;
```

```
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;

import com.manning.sdmia.directory.dao.ContactDao;
import com.manning.sdmia.directory.domain.Contact;

@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration
public class ContactDaoJdbcTest {

  @Autowired
  private ContactDao contactDao;

  @Test public void getContacts() {
    (...)
  }

}
```

Uses TestContext framework to manage Spring application context

Injects bean

Tests contact DAO

Remember that such tests should be written *before* testing the classes in an OSGi container: the core features of a module should be thoroughly tested in the module itself. Having fragile modules can make integration tests more difficult to write and makes diagnosing errors much harder.

Now that you have guidelines about testing the core features of modules, it's time to see how to test them in their target environment—OSGi, in our case. This is the goal of the next section, which covers how software testing applies to OSGi.

10.1.3 Testing OSGi components

Testing an OSGi component involves testing how it behaves when it interacts with the OSGi environment. It can either interact directly with the OSGi API or interact with the OSGi platform (by exporting or consuming services).

OSGi is all about modularity, and hopefully only carefully chosen parts of a component will interact with the OSGi environment. This means that most of the component's parts will be tested the usual way, with plain old unit and integration tests, by using the same techniques and tools we saw in the previous section. This is especially true when using Spring DM, as it enforces a POJO programming model where the framework handles most (if not all) of the interactions with the OSGi platform.

Application classes, like business services or DAOs, should be thoroughly tested as in a normal, non-OSGi environment. Then their basic operations (some of them, but not necessarily all) should be tested in an OSGi environment, to check that they behave as expected when used in their target environment and when they interact with other OSGi components. Automating these tests is challenging as it requires running the tests in an embedded OSGi platform. That's where Spring DM can help, because it provides this support, along with other testing facilities.

We'll introduce these testing tools in the next subsection. As testing OSGi applications can be quite different from testing standard applications in terms of project structuring, we'll also cover how to organize the layout of OSGi projects.

HOW SPRING DM CAN HELP TEST OSGI COMPONENTS

Tests for OSGi components can be roughly divided into two categories:

- *Unit tests*—When a part of the component relies on the OSGi API (`BundleContext`, `ServiceTracker`, and so on)
- *Integration tests*—When the component interacts with the OSGi environment (imports or exports packages, registers or consumes services, and so on)

For example, if an OSGi bundle has a `BundleActivator` that handles critical operations, writing a unit test for it is essential. Such a test can be written by using OSGi mocks: specific mock objects that implement the OSGi API and whose behavior can be modified programmatically. For testing a `BundleActivator`, we'd need at least a mock `BundleContext`. By using such a construct, the unit test doesn't need to be run on an OSGi platform—a plain old test framework is enough. Spring DM provides a set of OSGi mocks, which we'll study in section 10.2.

Integration tests imply interactions with the OSGi environment that can't be easily mocked. In OSGi integration tests we want to check several things, such as that a bundle can be started, which means that the platform resolved all the Java packages it imports or that a bundle published a service with the correct metadata (which can be part of its contract). These two tests can't be done with mock objects; they require a running OSGi platform. This means adapting test frameworks like JUnit to be able to bootstrap an OSGi platform, provision it, and run the test as if it were executing in OSGi. In doing this, OSGi integration tests can benefit from the array of tooling available for test frameworks (IDE and build tool launchers, XML/HTML reports, and so on) and thus don't need any special treatment. Spring DM provides such support, which turns the test into an OSGi bundle on the fly and runs its methods in an embedded OSGi platform. We cover this support in section 10.3.

Now that the purpose of OSGi tests is clearer, let's look at how to organize application projects to make their testing easier and more efficient.

HOW TO ORGANIZE OSGI TESTS

Depending on the nature of the test, test classes can be located in various places, and this has ramifications for the project's structure. We'll see how to organize standard non-OSGi tests, OSGi tests that use OSGi mocks, and OSGi integration tests.

Standard non-OSGi test classes (either unit or integration) usually reside in the same project as the classes they test. Let's take a data access bundle as an example: the tests consist of checking that all the DAOs do their data access job correctly (such as complex SQL queries). The test classes are located in a dedicated directory, separated from the actual application classes. The project also has all the necessary dependencies in its classpath (application classes like the domain layer, and technical libraries like Spring and Hibernate). There's no technical constraint that prevents the tests from being in the same project, which is convenient.

OSGi tests that use OSGi mocks can also be located in the same project as the classes they test (for example, an OSGi test that tests a bundle activator or any application class that relies on the OSGi API). As the scope of these tests is quite limited (there's no need for a full-blown OSGi platform running), the class tests are based on standard test tooling; the only difference is that they use OSGi mocks.

OSGi integration tests usually lie in a different project than the OSGi bundles they test. As these tests provision an OSGi platform with the bundles they test, the bundles being tested must be properly packaged before the tests can be run. The common sequence for OSGi integration tests is to build the to-be-tested bundles (compilation, "standard" tests, and packaging) and then run all the OSGi integration tests using a dedicated test project. Figure 10.3 illustrates the structure of an OSGi application and how to organize the tests.

With the support that Spring DM provides, OSGi integration tests end up being normal tests in that they can be run with common tools like Eclipse or Maven 2. The common way to organize the structure of an OSGi project is to use a Maven 2 module-based project: each bundle is a Maven 2 module, and Maven takes care of building the project in the right order. Once all the application bundles have been packaged and installed in the Maven 2 local repository, Maven 2 builds the last module, which is the integration test project. Note that the default behavior of Spring DM testing support is to provision the embedded OSGi platform from the Maven 2 local repository.

You now have a better understanding of software tests in an OSGi environment; it's time to apply this knowledge! The next two sections cover the techniques and tools that Spring DM provides for creating and running units tests with OSGi mocks (section 10.2) and OSGi integration tests (section 10.3).

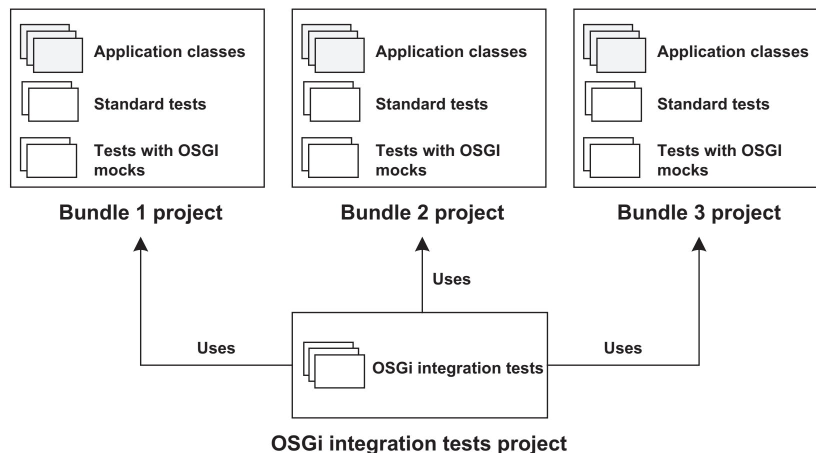

Figure 10.3 Standard tests and tests that use OSGi mocks are in the same project as the classes they test.; OSGi integration tests are in a separate project.

10.2 Strict unit tests for OSGi components

Even though Spring DM promotes a POJO-based programming model and handles most of the interaction with the OSGi environment, some parts of an OSGi bundle may still have to work with the OSGi API. These parts can be bundle activators or Spring beans that leverage the OSGi bridges that Spring DM offers (like the `BundleContextAware` interface introduced in chapter 4). Integration tests can be sufficient for testing these interactions as long as these OSGi-related tasks remain simple, but unit tests generally prove to be necessary when the interactions become more complex. This section introduces Spring DM's OSGi mock classes and provides some examples to illustrate their use.

10.2.1 Spring DM's OSGi mocks

Spring DM comes with a set of OSGi mocks that make OSGi unit testing easier. Generic mock libraries (like EasyMock or JMock) are another possible solution, but using them for mocking an API as complex as the OSGi Framework is very cumbersome. Spring DM itself is tested using a combination of its own OSGi mocks and mocks generated by EasyMock.

NOTE Spring DM's OSGi mocks are located in the `mock` module of the project.

The OSGi mocks that Spring DM provides are not meant to be a full-featured OSGi mock library, but rather internal tools that the Spring DM Framework makes available to others. Nevertheless, they make a good basis for OSGi unit tests—what's good for the goose is good for the gander.

Table 10.1 lists Spring DM's OSGi mocks, which all belong to the `org.springframework.osgi.mock` package. Table 10.1 also shows the corresponding emulated OSGi interfaces, which belong to the `org.osgi.framework` package.

Table 10.1 Spring DM's OSGi mocks

Spring DM class	Emulated OSGi interface	Description
`MockBundle`	`Bundle`	Maintains bundle metadata and delegates loading operations to its own classloader
`MockBundleActivator`	`BundleActivator`	Empty implementation
`MockBundleContext`	`BundleContext`	Maintains a bundle object, a list of service listeners, and a list of bundle listeners
`MockFilter`	`Filter`	Empty implementation
`MockServiceReference`	`ServiceReference`	Maintains service properties and handles OSGi properties like service ID, object class, and service ranking
`MockServiceRegistration`	`ServiceRegistration`	Maintains service properties

Spring DM's OSGi mocks maintain minimal features, but most of their methods are empty implementations. Users are encouraged to subclass the mock classes in their tests (as with anonymous classes) and override only the methods they need.

> **Is there a full-featured OSGi mock library?**
>
> Although generic mock libraries proliferate in the Java world, there's no de facto standard for mocking the OSGi Framework. The SpringSource dm Server team developed the OSGi test stubs library, which is still in its infancy at the time of this writing. Nevertheless, it looks promising and is already used in the dm Server test suite. A SpringSource Team Blog entry (http://blog.springsource.com/2009/06/23/osgi-test-stubs/) introduces the OSGi test stubs library.

Let's see how to use some of Spring DM's OSGi mocks.

10.2.2 Spring DM's OSGi mocks in action

Suppose you want to test a simple `BundleActivator`, like the one defined in listing 10.1.

Listing 10.1 A bundle activator to be unit tested

```
package com.manning.sdmia.ch10.internal;

import org.osgi.framework.BundleActivator;
import org.osgi.framework.BundleContext;
import org.osgi.framework.ServiceRegistration;
import com.manning.sdmia.ch10.dao.ContactDao;
import com.manning.sdmia.ch10.dao.jdbc.ContactDaoJdbc;

public class DirectoryDaoBundleActivatorSimple implements BundleActivator {

  private ServiceRegistration serviceRegistrationDao;

  @Override
  public void start(BundleContext bundleContext) throws Exception {
    serviceRegistrationDao = bundleContext.
                             registerService(
      ContactDao.class.getName(),
      new ContactDaoJdbc(),
      null
    );
  }

  public void stop(BundleContext bundleContext) throws Exception {
    serviceRegistrationDao.unregister();
  }
}
```

Registers DAO as OSGi service (start method); Unregisters service (stop method)

The bundle activator's contract consists of registering a service and also taking care of the unregistration.

> **NOTE** The interactions with the OSGi service registry in the current section are purposefully simple so as to illustrate the use of Spring DM's OSGi mocks. You should always try to use Spring DM's declarative features for such tasks, because they're more reliable and flexible.

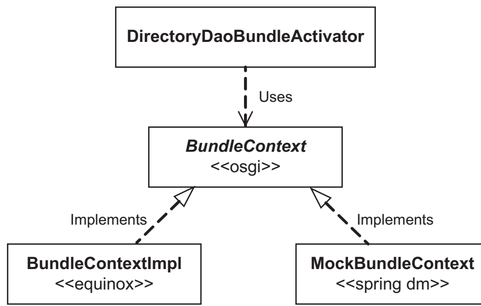

Figure 10.4 Depending on the environment (an OSGi platform like Equinox, or test), the bundle activator uses a different implementation of `BundleContext`. When running a unit test, we use the `MockBundleContext`, which doesn't need any runtime environment.

Figure 10.4 illustrates which implementation of bundle context the bundle activator will use when running on an OSGi platform (such as Equinox) and when running in a test. This is made possible by the use of the `BundleContext` interface.

Listing 10.2 illustrates how to test the bundle activator using Spring DM's mocks.

Listing 10.2 Unit testing the bundle activator with OSGi mocks

```
package com.manning.sdmia.ch10.internal;

import java.util.Dictionary;
import org.junit.Assert;
import org.junit.Before;
import org.junit.Test;
import org.osgi.framework.BundleActivator;
import org.osgi.framework.BundleContext;
import org.osgi.framework.ServiceRegistration;
import org.springframework.osgi.mock.MockBundleContext;
import org.springframework.osgi.mock.MockServiceRegistration;
import com.manning.sdmia.ch10.dao.ContactDao;

public class DirectoryDaoBundleActivatorSimpleTest {

  int daoRegistered;

  @Before public void setUp() {                              ❶ Initializes
    daoRegistered = 0;                                          test instance
  }

  @Test public void startAndStop() throws Exception {        ❷ Declares test method
    MockBundleContext bundleContext =                        ❸ Declares mock
      new MockBundleContext() {                                 bundle context inline

      @Override
      public ServiceRegistration registerService(String clazz,
          Object service, Dictionary properties) {
        if(service instanceof ContactDao) {
          daoRegistered++;
          return new MockServiceRegistration() {
            @Override
            public void unregister() {                       ❹ Tracks
              daoRegistered--;                                  registrations and
            };                                                  unregistrations
          };
```

```
        } else {
          return super.registerService(clazz, service, properties);
        }
      }
    };

    BundleActivator bundleActivator =
      new DirectoryDaoBundleActivatorSimple();
    bundleActivator.start(bundleContext);
    Assert.assertEquals(1,daoRegistered);
    bundleActivator.stop(bundleContext);
    Assert.assertEquals(0,daoRegistered);
  }
}
```

5 Tests start and stop scenario

The unit test consists of checking that the `daoRegistered` variable (reinitialized before each test method at **1** is correctly incremented and decremented. The test method **2** tests both the `start` and `stop` methods of the activator. Because each needs the bundle context as a parameter, we declare a `BundleContext` variable and use Spring DM's `MockBundleContext` class for the implementation **3**. We use an anonymous class to override the `registerService` method and increment the `daoRegistered` variable only if the OSGi service is a `ContactDao`. To track the unregistration of the service, we use a `MockServiceRegistration` instance whose `unregister` method is also overridden to decrement the counter **4**. At **5**, we execute the `start` and `stop` methods of the bundle activator and check that the counter is properly updated.

We can push our bundle activator a little further and make it register the OSGi service only if a `DataSource` is available in the service registry. Listing 10.3 illustrates this slight modification.

Listing 10.3 New version of the bundle activator (with conditional service registration)

```
public class DirectoryDaoBundleActivatorWithCondition
    implements BundleActivator {
  (...)
  @Override
  public void start(BundleContext bundleContext) throws Exception {
    if(bundleContext.getServiceReference(
      DataSource.class.getName()) != null) {
      (...)
    }
  }
  public void stop(BundleContext bundleContext) throws Exception {
    if(serviceRegistrationDao != null) {
      serviceRegistrationDao.unregister();
    }
  }
  (...)
}
```

Tests whether DataSource service is available

Registers service (code omitted)

Unit testing the new version of the bundle activator implies testing two paths (for whether or not the `DataSource` service is available), which means two test cases. It also

implies overriding the `getServiceReference` method of the `MockBundleContext` and making it return either a dummy `ServiceReference` or nothing. Listing 10.4 illustrates this new test.

Listing 10.4 Unit testing the conditional service registration

```
public class DirectoryDaoBundleActivatorWithConditionTest {
  (...)
  @Test public void startAndStopWithDataSource()
      throws Exception {
    BundleContext bundleContext = new MockBundleContext() {
      @Override
      public ServiceReference getServiceReference(String clazz) {
        return new MockServiceReference();
      }
      @Override
      public ServiceRegistration registerService(String clazz,
          Object service, Dictionary properties) {
        // same implementation as before
      }
    };

    BundleActivator bundleActivator =
➥     new DirectoryDaoBundleActivatorWithCondition();
    bundleActivator.start(bundleContext);
    Assert.assertEquals(1,daoRegistered);
    bundleActivator.stop(bundleContext);
    Assert.assertEquals(0,daoRegistered);
  }

  @Test public void startAndStopNoDataSource()
      throws Exception {
    MockBundleContext bundleContext = new MockBundleContext() {
      @Override
      public ServiceReference getServiceReference(String clazz) {
        return null;
      }
      @Override
      public ServiceRegistration registerService(String clazz,
          Object service, Dictionary properties) {
        // same implementation as before
      }
    };

    BundleActivator bundleActivator =
➥     new DirectoryDaoBundleActivatorWithCondition();
    bundleActivator.start(bundleContext);
    Assert.assertEquals(0,daoRegistered);
    bundleActivator.stop(bundleContext);
    Assert.assertEquals(0,daoRegistered);
  }
}
```

Annotations: Tests "DataSource available" path; Simulates DataSource availability; Tests "DataSource unavailable" path; Simulates DataSource unavailability

This ends our tour of the OSGi mocks available in Spring DM. These mock objects should fulfill your needs when the interactions between your classes and the OSGi

environment need comprehensive unit testing rather than more coarse-grained integration tests.

Speaking of integration tests, the next section covers Spring DM's integration test support for OSGi applications.

10.3 Integration tests for OSGi applications

When testing one or more OSGi bundles in their target environment (an implementation of an OSGi container), there are *a lot* of things that can go wrong, so checking them in an efficient way can save a lot of time, money, and sweat. The main challenge with OSGi integration tests is to keep close enough to developers' usual testing habits (such as using test frameworks like JUnit) while executing the test itself within a properly configured and provisioned OSGi container. Fortunately Spring DM takes up this challenge and provides a testing framework that smoothly fills the gap between OSGi and JUnit.

Section 10.3.1 covers the basics of Spring DM's integration test support through an example based on the data access layer of an enterprise application. We'll see in this section how to check common OSGi features, like the correct package exports for tested bundles and the consumption of registered OSGi services.

Section 10.3.2 then dives into more advanced features, like customizing the creation of the test bundle and changing the target OSGi platform.

10.3.1 Developing integration tests with Spring DM support

As shown previously in figure 10.3, OSGi integration tests should be located in a dedicated project. Creating this project is the first step when you start writing integration tests. In a typical Maven 2 modular project, the integration tests project is one of the submodules and should be built *after* the other submodules, so that the Maven 2 local repository is correctly filled with the modules' binaries (we'll see later in this section why the Maven 2 local repository is important when using Spring DM test support).

Once your test project is created, you're ready to benefit from Spring DM's test support. We'll see in this section how to create a simple integration test that only displays the OSGi platform we're running and what it's provisioned with. This first test seems simplistic, but it will illustrate how Spring DM runs the methods of the test within an embedded OSGi platform that it starts on the fly. Through testing this sample application, we'll see how to provision the embedded OSGi platform to test the visibility of Java classes and how to test that OSGi services are properly registered and functional.

CREATING A SIMPLE INTEGRATION TEST

Spring DM's integration test support lies in the `test` module of the project, so you'll have to add the corresponding JAR to your project, along with the other Spring DM binaries. Using Spring DM's test support is then as simple as inheriting from a test base class, as shown in listing 10.5.

WARNING If you use Spring DM 2, you need to override the `getTestingFrameworkBundlesConfiguration` method of the test base class, as explained in section 3.4.4. This is because Spring DM depends on a release of the Spring Framework that isn't available in Maven 2 repositories anymore. The listings of this chapter don't override this method for brevity's sake, but the examples in the book's source code do.

Listing 10.5 A simple OSGi integration based on Spring DM's test support

```
package com.manning.sdmia.directory.test;

import org.osgi.framework.Constants;
import org.springframework.osgi.test.
➥     AbstractConfigurableBundleCreatorTests;

public class SimpleIntegrationTest
       extends AbstractConfigurableBundleCreatorTests {        ❶ Extends test base class

  public void testPlatformInfo() {                              ❷ Declares test method
    System.out.println(
      "Platform is "+
      bundleContext.getProperty(
        Constants.FRAMEWORK_VENDOR)+
      " "+                                                       ❸ Prints OSGi platform information
      bundleContext.getProperty(
        Constants.FRAMEWORK_VERSION)
    );
  }

}
```

The `AbstractConfigurableBundleCreatorTests` class is the entry point for Spring DM's test support. All integration test classes must inherit from this class ❶. As for any test based on JUnit 3, every test method name must start with `test` if it is to be executed ❷. Our test method simply displays information about the running OSGi platform ❸. To do this, it uses the `bundleContext` property available in the `AbstractConfigurableBundleCreatorTests` class. We can use such a property because the test class is turned into an OSGi bundle on the fly by Spring DM test support.

Spring DM test framework and the Spring TestContext Framework

The Spring DM testing framework doesn't build on top of the Spring TestContext Framework (as of version 2.0.0.M1). It uses the JUnit 3.8 support from the Spring Framework, mainly because Spring DM prior to version 2.0 needed to support Java 1.4 (and the Spring TestContext Framework integrates with JUnit 4, which heavily uses Java 5 annotations). This ties you to JUnit 3.8 when using the Spring DM testing framework for OSGi integration test, but it doesn't prevent you from using the Spring TestContext Framework or any test framework for testing the core features of your modules.

If you run the test either in your IDE or on the command line with a tool like Maven 2, you should see something like the following on the console:

```
Platform is Eclipse 1.5.0
```

This tells us we are running under Eclipse Equinox, with the 1.5.0 version of the OSGi API (which means OSGi 4.2).

> **NOTE** Each OSGi integration test with Spring DM support must run in its own JVM. You can achieve this by using the "fork" option that tools like Maven 2 or Ant provide.

What happened exactly? What did Spring DM test framework do for us? Here are the steps the `AbstractConfigurableBundleCreatorTests` class accomplished:

- Started the OSGi platform
- Provisioned it with a minimal set of bundles
- Packaged the test into an on-the-fly bundle and installed it in the platform
- Executed the test methods inside the platform
- Shut down the platform

If you're curious and want to learn about the bundles that Spring DM provisioned the platform with, you can write a test method that lists the installed bundles:

```
public void testInstalledBundles() {
  for(Bundle bundle : bundleContext.getBundles()) {
    System.out.println(bundle.getSymbolicName());
  }
}
```

If you execute this method, you should see output like the following on the console:

```
org.eclipse.osgi
com.springsource.org.aopalliance
com.springsource.org.apache.log4j
com.springsource.junit
com.springsource.org.objectweb.asm
com.springsource.slf4j.api
com.springsource.slf4j.log4j
com.springsource.slf4j.org.apache.commons.logging
org.springframework.aop
org.springframework.asm
org.springframework.beans
org.springframework.context
org.springframework.core
org.springframework.expression
org.springframework.test
org.springframework.osgi.extensions.annotations
org.springframework.osgi.core
org.springframework.osgi.extender
org.springframework.osgi.io
org.springframework.osgi.test
TestBundle-testInstalledBundles-com.manning.sdmia.
➥    directory.test.SimpleIntegrationTest
```

1 OSGi platform (Equinox) bundle

2 Spring's dependencies and logging bundles

3 Spring bundles

4 Spring DM bundles

5 Test bundle

The AbstractConfigurableBundleCreatorTests class provisioned the platform with several bundles we can divide into five categories: the system bundle (Equinox, at ❶), Spring's dependencies and logging libraries ❷, Spring ❸, Spring DM ❹, and the test class itself ❺. Note that because Spring DM's extender is installed, any Spring-powered bundle should see its Spring application context bootstrapped.

But where do these bundles come from? By default, Spring DM provisions the embedded OSGi platform with bundles located in the Maven 2 local repository (see figure 10.5). This means that all the bundles listed in the previous console output must be available in the Maven 2 local repository (apart from the test bundle, which is created on the fly, and the system bundle, which must be on the classpath when running the test). This also means that, when provisioning Spring DM test instances from your Maven 2 local repository, the JARs you refer to must be OSGi-compliant bundles.

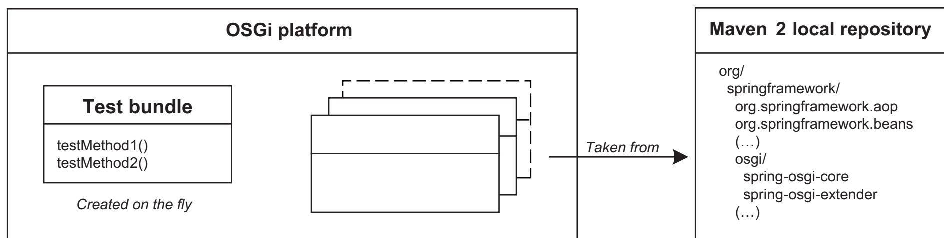

Figure 10.5 When running an integration test, Spring DM bootstraps an OSGi platform for the duration of the test. The test class is turned into an OSGi bundle on the fly and is used to provision the platform. By default, Spring DM provisions the platform with bundles taken from the Maven 2 local repository.

We've now seen the basics of Spring DM's test support. We're about to take advantage of the features provided by the class to test our OSGi bundles, Spring-powered or not, but before doing so, let's introduce the sample application that will be our guinea pig for the tests.

INTRODUCING THE SAMPLE APPLICATION

The sample application we'll use is part of a typical layered enterprise application. It consists of three bundles:

- *Directory domain*—Contains the domain classes (entities)
- *Directory DAO API*—Contains the DAO interfaces
- *Directory DAO JDBC*—A JDBC implementation of the DAO API

Figure 10.6 illustrates the relationships between the bundles and the interactions with the OSGi service registry.

The sample application gives us the opportunity to test the following:

- That the domain and DAO API bundles export their packages correctly
- That the DAO JDBC implementation bundle can see the domain and DAO classes and register the contact DAO

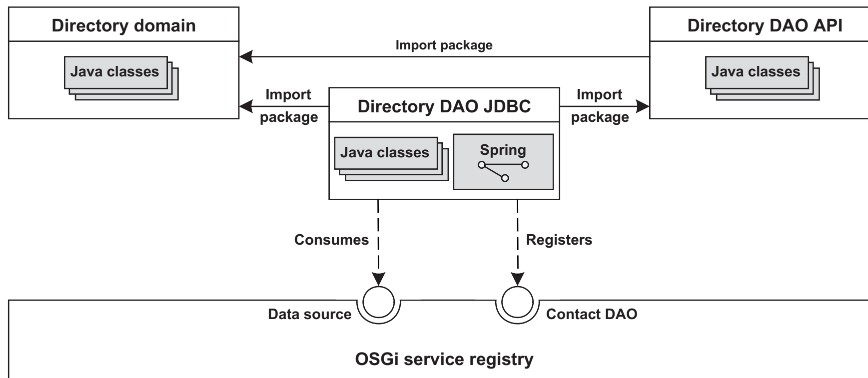

Figure 10.6 The to-be-tested application, composed of two API bundles and a Spring-powered implementation bundle that interacts with the OSGi service registry.

We'll start by testing the exporting of packages.

TESTING THE DOMAIN AND DAO API PACKAGES EXPORTATION

As shown in figure 10.6, our directory domain and directory DAO API bundles export packages that need to be visible to other bundles. An integration test can consist in provisioning an OSGi platform with both bundles and trying to import packages from them to ensure that these packages are correctly exported by our bundles. Note that such a test doesn't involve any Spring-powered bundles: Spring DM integration support works for any kind of bundle.

Listing 10.6 shows the integration test.

Listing 10.6 Provisioning a test and testing the visibility of classes

```
package com.manning.sdmia.directory.test;

import org.springframework.osgi.test.
➥    AbstractConfigurableBundleCreatorTests;

import com.manning.sdmia.directory.dao.ContactDao;
import com.manning.sdmia.directory.domain.Contact;

public class DirectoryClassVisibilityTest
       extends AbstractConfigurableBundleCreatorTests {

  public void testVisibleClasses() {
    Contact.class.getName();              // 1 Forces classloading
    ContactDao.class.getName();
  }

  @Override
  protected String[] getTestBundlesNames() {   // 2 Overrides provisioning method
    return new String [] {
      "com.manning.sdmia.ch10, directory-domain, 1.0.0",   // 3 Provisions with application bundles
      "com.manning.sdmia.ch10, directory-dao, 1.0.0"
    };
  }

}
```

The test method contains instructions that force the loading of a domain class and of a DAO interface ❶. Spring DM needs such explicit references to classes to detect needed Java imports and generate an appropriate manifest on the fly. The test class compiles, but what we want is to ensure that the loading of these classes works correctly in an OSGi platform. At ❷, we use one of `AbstractConfigurableBundleCreatorTests`'s hooks, the `getTestBundlesNames` method, to instruct Spring DM how to provision the OSGi platform. We need to return an array of `Strings`, each element corresponding to a bundle ❸. As stated before, Spring DM's test support heavily relies on Maven 2 concepts for provisioning, so bundles are located with their Maven 2 coordinates, using the following pattern:

```
[groupId], [artifactId], [version]
```

When running the test, you should get a green bar, because the OSGi platform is correctly provisioned and the test bundle can see the classes and interfaces it needs. You can comment out the provisioning part of the test and note that the test fails, as the domain class and the DAO interface are no longer available.

> **WARNING** Always place your test classes in different packages than the ones you want to import. When Spring DM generates the manifest of the test bundle, it never imports packages that are in the test bundle. If the test class of our sample had been in the `com.manning.sdmia.directory.domain` package, Spring DM would not have imported it and the test would have failed.

What did we learn from this test? Spring DM scans the test class to detect dependencies and generate an appropriate manifest for the test bundle. Because the test methods are executed in the OSGi platform, these dependencies must be available. They're usually provided by bundles that the platform is provisioned with, so we need to override the `getTestBundlesNames` method to instruct Spring DM with bundles we want it to provision the platform with. We use a Maven-like method for this, by locating bundles with their coordinates.

We've just seen how to test a typical export/import package scenario. Let's see now how to test whether bundles correctly interact with the OSGi service registry.

TESTING THE DAO IMPLEMENTATION BUNDLE

There are several conditions to be fulfilled for our DAO implementation bundle to work correctly: some packages need to be exported (the domain package and the package of the DAO API), the bundle must import them correctly, and it must register a DAO as an OSGi service. Our integration tests will check that all these things work fine. (We wrote "integration tests", plural, because we'll explore different testing strategies. This will also give us the opportunity to explore features of Spring DM test support.)

Listing 10.7 shows the first version of our integration test.

Listing 10.7 Using a lookup on the OSGi registry to get the DAO

```
package com.manning.sdmia.directory.test.dao;

import junit.framework.Assert;

import org.osgi.framework.ServiceReference;
```

```
import org.springframework.osgi.test.
➥     AbstractConfigurableBundleCreatorTests;

import com.manning.sdmia.directory.dao.ContactDao;

public class ContactDaoLookupIntegrationTest
       extends AbstractConfigurableBundleCreatorTests {

  public void testGetContacts() throws Exception {
    ServiceReference ref =
        bundleContext.getServiceReference(
      ContactDao.class.getName()
    );
    Assert.assertNotNull(
      "a DAO should be registered",
      ref);
    ContactDao contactDao =
        (ContactDao) bundleContext.getService(
      ref
    );
    Assert.assertEquals(
      3,
     contactDao.getContacts().size());
  }

  @Override
  protected String[] getTestBundlesNames() {
    return new String [] {
      "com.manning.sdmia.ch10, " +
        "directory-domain, 1.0.0",
      "com.manning.sdmia.ch10, directory-dao, 1.0.0",
      "com.manning.sdmia.ch10, " +
        "directory-dao-jdbc, 1.0.0",
      "org.springframework, " +
        "org.springframework.jdbc, "+
        getSpringVersion(),
      "org.springframework, " +
        "org.springframework.transaction, "+
        getSpringVersion(),
      "com.h2database, h2, 1.1.118",
      "com.manning.sdmia.ch10, " +
        "ch10-datasource-tests, 1.0.0"
    };
  }
}
```

1. Checks DAO OSGi service is present
2. Tests DAO method
3. Adds application bundles
4. Adds Spring data access bundles
5. Adds database bundles

At (1), we create a lookup to get the DAO from the OSGi service registry (we test that there's at least a service registered under the interface). The core of our test is rather simple (2): we call a method to retrieve `Contact` objects and check the size of the list (this implies that we know how many `Contact` rows there are in the `DataSource` we're using—we'll see more about that later). Should we test more than that? If we suppose the DAO is correctly tested in its owning module, we don't need to do more in an integration test.

Then comes the provisioning. We start with the application bundles (3): domain, DAO API, and JDBC-based DAO implementation. The latter uses the `JdbcTemplate`, so we also need to add some modules of the Spring Framework (4). Note the use of the `getSpringVersion` method, which is convenient for provisioning the test OSGi

container with the same version of the Spring Framework that Spring DM uses in the test support (there's also a getSpringDMVersion method). As the JDBC-based DAO implementation needs a DataSource service (as shown in figure 10.6), we need to provision the OSGi platform with a bundle that registers such a service. That's what we do at ❺, where we add a bundle meant for the test and another bundle for the corresponding database driver.

> **NOTE** The DataSource bundle is a Spring-powered bundle that bootstraps an in-memory database and registers the DataSource as an OSGi service—all of this with Spring and Spring DM. We don't show its full code here for the sake of brevity.

This is just fine: the OSGi platform is provisioned with everything we need to test that the DAO implementation bundle works correctly. We just have to run the test and wait for the green bar. But we want more! Spring DM promotes a POJO programming model, and the lookup on the service registry denies this. Wouldn't it be nice to have the DAO directly available (injected!) in the test? This is possible, because Spring DM allows tying an OSGi Spring application context to the test and injecting beans in it. Listing 10.8 shows what could be the configuration of this application context: it looks up the DAO on the OSGi registry with the osgi:reference XML element.

Listing 10.8 Spring application context for the test

```
<?xml version="1.0" encoding="UTF-8"?>
<beans (...)>

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

</beans>
```

The application context configuration leverages Spring DM's osgi namespace to look up the DAO and create a contactDao bean. This bean will be injected in the test instance at runtime, just by adding the corresponding property. This configuration may look like overkill for a single OSGi service, but it can save you a lot of tedious code when you have more services.

Listing 10.9 shows a new version of the test that benefits from the injection.

Listing 10.9 Injecting the DAO into the test via its Spring application context

```
public class ContactDaoInjectionIntegrationTest
       extends AbstractConfigurableBundleCreatorTests {

  private ContactDao contactDao;

  public void setContactDao(ContactDao contactDao) {
    this.contactDao = contactDao;
  }

  public void testGetContacts() throws Exception {
    Assert.assertEquals(1,contactDao.getContacts().size());   <-- ❶ Tests DAO method
```

```
  }

  @Override
  protected String[] getTestBundlesNames() {
    (...)
  }

  @Override
  protected String[] getConfigLocations() {
    return new String[] {
      "/com/manning/sdmia/directory/test/dao/
➥    ContactDaoServiceDataSourceIntegrationTest
➥    -context.xml"
    };
  }
}
```

❷ **Declares Spring file for test**

The test now declares a `contactDao` property and the corresponding setter. Spring DM uses the latter to automatically inject the `contactDao` bean. This injection is made possible by the matching between the bean name in the application context and the name of the property in the test (the corresponding setter is mandatory). The test method is then simpler ❶, as it doesn't need the lookup anymore. We need to override the `getConfigLocations` method ❷ to tell Spring DM where to find the Spring configuration files.

We just saw how to look up an OSGi service via Spring DM and inject it in the test instance. This is made possible by the Spring application context that Spring DM ties to the test instance. The scenario we chose is a little complex, because it involves an OSGi lookup and then dependency injection in the test, but the application context of a test instance can be used for any operation. You could, for instance, retrieve the `DataSource` service and inject it in a bean that inserts data into the database. That's exactly what's done in the application context of the final version of our test:

```
<osgi:reference id="dataSource"
                interface="javax.sql.DataSource" />

<bean class="com.manning.sdmia.directory.test.dao.DatabaseInitializer">
  <constructor-arg ref="dataSource" />
</bean>
```

The `DatabaseInitializer` uses a `JdbcTemplate` to initialize the `DataSource`: this means that the package of the `JdbcTemplate` needs to be imported by the test bundle. This is the case for any package involved in the creation of the application context of the test. Spring DM has no way to know about such imports, so we must explicitly inform it about them. In our case, this can be done by declaring a dummy `JdbcTemplate` property in the test class:

```
public class ContactDaoServiceDataSourceIntegrationTest
       extends AbstractConfigurableBundleCreatorTests {

  private JdbcTemplate jdbcTemplate;

  (...)
}
```

Why not embed the DataSource in the test application context?

The DAO implementation bundle needs a `DataSource` available in the service registry to work correctly. We developed a dedicated bundle that creates and registers such a `DataSource` for our integration test, but wouldn't it be simpler to do this directly in the test application context, to avoid developing a dedicated bundle?

Yes it would be, but there's a reason we did what we did. By default, `AbstractConfigurableBundleCreatorTests` waits for the application context of each Spring-powered bundle to be created before running. This means the creation of the test application context is triggered *after* the creation of the Spring-powered bundles. Why should we care? Because of dependencies. If the `DataSource` service is a mandatory dependency of the DAO implementation bundle, Spring DM won't complete the creation of the corresponding application context. As the `DataSource` service is provided by the test application, which will start only after the creation of all application contexts, we're in front of a deadlock: the test doesn't run, it hangs.

What should we do then? We can make the `DataSource` service an optional dependency of the DAO implementation bundle, but if the `DataSource` must remain mandatory, this isn't an option. Another solution is to change the test's default behavior (waiting for the creation of the context of Spring-powered bundles) by overriding the `shouldWaitForSpringBundlesContextCreation` method and make it return `false`. If you choose this path, remember that the test methods can be run *before* all the Spring-powered bundles have completed their initialization. Creating the dedicated `DataSource` bundle doesn't seem like such a bad idea after all!

You're now aware of the basics of Spring DM's test support. We've covered how to test classic scenarios like package export/import and interaction with the OSGi service registry. This introduction should be enough for most of your OSGi integration tests.

In the next section, we'll learn more about the hooks that the `AbstractConfigurableBundleCreatorTests` class offers for running tests under different OSGi platforms or customizing the creation of the bundle test manifest (among other scenarios).

10.3.2 Advanced features of Spring DM test support

The `AbstractConfigurableBundleCreatorTests` offers reasonable defaults for the on-the-fly generation of the test bundle, the provisioning mechanics, and the underlying OSGi platform. But sometimes these defaults aren't appropriate, so you'll be happy to learn that the `AbstractConfigurableBundleCreatorTests` class offers hooks to override them. We're about to look at some of these hooks, which will allow us to dive further into the mechanics of test-bundle generation. We'll then apply this knowledge to customizing the creation of the test manifest and to changing the OSGi platform the test is run under.

HOOKS OF THE ABSTRACTCONFIGURABLEBUNDLECREATORTESTS CLASS

Before running the test methods, Spring DM turns the test instance into an on-the-fly bundle before provisioning the OSGi platform with it. Spring DM makes decisions regarding the content of the test bundle and its manifest (such as for import and export

entries) that you can customize by overriding methods in your test class. Table 10.2 lists these methods as well as methods that change the default behavior of test execution.

Table 10.2 Test methods to override to change the behavior of the test execution

Method	Default value	Description
`getRootPath`	`file:./target/test-classes`	The root path used for located resources.
`getBundleContentPattern`	`**/*`	Comma-separated patterns to identify resources included in the bundle.
`getManifestLocation`	`null`	Location of the manifest file to use.
`getSettingsLocation`	`[TestName]-bundle.properties`	Location of the setting properties to use.
`shouldWaitForSpringBundlesContextCreation`	`true`	Whether or not to wait for the context creation of Spring-powered bundles before running the test.
`getBootDelegationPackages`	`javax.*, org.w3c.*, org.xml.*, sun.*, org.apache.xerces.jaxp.*`	The list of packages whose loading is delegated to the boot classloader.
`getPlatformName`	`Platforms.EQUINOX`	The OSGi platform to use for the execution of the test.

We can order the methods of table 10.2 into different categories. We'll start by covering those meant to customize the content of the test bundle, but first we need to study the principles of this customization.

> **Why customize the content of the test bundle?**
>
> Spring DM has reasonable defaults when it generates the test bundle—unfortunately they can't always be appropriate. The default root path is adapted to a standard Maven 2 layout, and not every Spring DM project uses Maven 2. By default, all the content of the root path is included in the test bundle, and narrowing the included files by using appropriate patterns can speed up bundle generation, which matters when the number of integration tests grows. Very specific integration tests may also need a very specific bundle manifest that Spring DM can't automatically generate.

Spring DM offers two ways to configure the content of the test bundle:

- Programmatically, by overriding methods from `AbstractConfigurableBundleCreatorTests`
- Declaratively, by using a properties file

Table 10.3 lists the customizable content of the test bundle and the corresponding elements (method or property keys) for both configuration approaches.

Table 10.3 Customizable items of the test bundle content

Item	Property	Method	Default value
Root path for located resources	`root.dir`	`getRootPath`	`file:./target/test-classes`
Comma-separated patterns to identify resources included in the bundle	`include.patterns`	`getBundleContentPattern`	`**/*`
Location of the manifest file to use	`Manifest`	`getManifestLocation`	`null`

Listing 10.10 shows how to customize the test bundle content *programmatically*.

Listing 10.10 Programmatically customizing the test bundle content

```
public class ProgrammaticContentTest
        extends AbstractConfigurableBundleCreatorTests {

  @Override
  protected String getRootPath() {
    return "file:./bin/test";
  }

  @Override
  protected String[] getBundleContentPattern() {
    return new String [] {
      "/**/*.class",
      "/**/*.xml"
    };
  }

  @Override
  protected String getManifestLocation() {
    return "file:./src/test/resources/com/manning/
➥     sdmia/directory/test/content/manifest.mf";
  }

  (...)

}
```

- Customizes bundle root directory
- Customizes resources included in bundle
- Indicates specific manifest location

Note in listing 10.10 that the `getManifestLocation` method returns the manifest location using the Spring resource syntax. Let's move on now to the declarative customization.

By default, Spring DM's test infrastructure looks for a property file that has a similar name to the test case: for the test `com.manning.sdmia.MyTest`, the properties file must be named `com/manning/sdmia/MyTest-bundle.properties` and it must be located in the classpath of the test (the standard test, not the on-the-fly test bundle).

The following snippet shows a properties file that performs the equivalent of listing 10.10:

```
root.dir=file:./bin/test
include.patterns=/**/*.class,/**/*.xml
manifest=file:./src/test/resources/com/manning/
➥    sdmia/directory/test/content/manifest.mf
```

If the default name and location of the properties file doesn't suit you, you can override the `getSettingsLocation` method to give it a new location:

```
public class DeclarativeSettingLocationContentTest
       extends AbstractConfigurableBundleCreatorTests {

  @Override
  protected String getSettingsLocation() {
    return "/my/path/to/DeclarativeTest.properties";
  }

}
```

We've now covered the ways Spring DM offers to customize the content of the test bundle it generates. The manifest is one element of this configuration; let's see how we can create it instead of relying on its automatic generation.

CUSTOMIZING THE GENERATED MANIFEST

By analyzing the test class, Spring DM is able to generate an appropriate manifest for the on-the-fly bundle. This is particularly convenient when the test depends on classes that are exported by other bundles; Spring DM automatically adds the correct `Import-Package` entries. But sometimes the automatic manifest generation isn't enough, and we need to explicitly provide the manifest of the test bundle.

> **WARNING** As the writing of OSGi manifests is a cumbersome and error-prone process, you should only write them yourself when you face a dead-end with Spring DM's automatic generation.

We saw previously how to indicate the location of a specific manifest file (either programmatically or declaratively), so let's see now what it should contain. Listing 10.11 illustrates this.

Listing 10.11 Customizing the test manifest

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: com.manning.sdmia.directory.test.content
 .ProgrammaticContentTest
Bundle-SymbolicName: com.manning.sdmia.directory.test.content
 .ProgrammaticContentTest
Bundle-Description: on-the-fly test bundle
Bundle-Activator: org.springframework.osgi.test                 ❶ Internal test
➥    .JUnitTestActivator                                         bundle activator
Import-Package: org.springframework.osgi.test,             ❷ Import necessary
 junit.framework,                                            packages
 com.manning.sdmia.directory.domain
```

The manifest needs specific entries for the test to run correctly as an OSGi bundle in the Spring DM test framework. The first entry is the bundle activator ❶, and the second is the importation of packages related to the test infrastructure and the execution of the test methods ❷. The packages of the test infrastructure are mandatory for every Spring DM test. Packages that are used inside the test methods must also be specified. The manifest must also contain all the compulsory headers of an OSGi bundle, like `Bundle-SymbolicName`. The manifest must also meet the usual conditions of its folks (such as having no more than 72 characters on a line). With all these conditions, you can certainly understand that Spring DM's automatic manifest generation is worth using!

We thoroughly covered the customization of test bundles, so let's finish our tour of Spring DM's test support with the choice of the OSGi platform the tests run under.

CHOOSING THE OSGI PLATFORM

The Spring DM test framework supports three OSGi platforms: Equinox (the default), Felix, and Knopflerfish. This means that you can easily change the platform that tests are run under to check the portability of your applications.

Spring DM offers two ways to choose the OSGi platform:

- Programmatically, by overriding the `getPlatformName` method in the test class
- Declaratively, by specifying a system property

Listing 10.12 shows how to switch to Felix, using the programmatic approach.

Listing 10.12 Changing the platform programmatically in a test

```
package com.manning.sdmia.directory.test;

import org.osgi.framework.Bundle;
import org.osgi.framework.Constants;
import org.springframework.osgi.test.
➥     AbstractConfigurableBundleCreatorTests;
import org.springframework.osgi.test.platform.Platforms;

public class SimpleIntegrationTest
       extends AbstractConfigurableBundleCreatorTests {

  public void testPlatformInfo() {
    System.out.println(
      "Platform is "+
      bundleContext.getProperty(
        Constants.FRAMEWORK_VENDOR)+
      " "+
      bundleContext.getProperty(
        Constants.FRAMEWORK_VERSION)
    );
  }

  @Override
  protected String getPlatformName() {
    return Platforms.FELIX;
  }
}
```

Displays info on platform

Switches to Felix

The `getPlatformName` method should return public properties of `org.springframework.osgi.test.platform.Platforms`: `EQUINOX`, `FELIX`, and `KNOPFLERFISH`. This programmatic approach is simple, but it ties the test to the platform. Let's look at the declarative approach, which doesn't interfere with test classes.

The declarative approach consists in specifying a system property, `org.springframework.osgi.test.framework`. The property must take as a value the name of the `OsgiPlatform` class you want to use. Spring DM provides an implementation for each platform it supports.

How you specify the system property depends on the way you launch your tests. The declarative approach is meant to be used with build tools: running the whole test suite on different platforms becomes just a matter of configuration. Each build tool has its own way to set system properties. Let's see how to do it with Maven 2.

Maven 2 uses the `surefire` plug-in to run tests, and system properties must be set in the configuration of this plug-in. This happens to be in the `build` section of the POM, as shown in listing 10.13.

Listing 10.13 Setting the OSGi platform for tests with Maven 2

```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-surefire-plugin</artifactId>
      <configuration>
        <systemProperties>
          <property>
            <name>
            org.springframework.osgi.test.framework
            </name>
            <value>
            org.springframework.osgi.test.platform.FelixPlatform
            </value>
          </property>
        </systemProperties>
      </configuration>
    </plugin>
  </plugins>
</build>
```

Uses Felix for tests

Note that the platform binaries need to be in the classpath of the project; otherwise Spring DM won't be able to create the embedded instance. When using Maven 2, profiles are a good way to switch from one platform to another: you declare a profile for each platform with the corresponding dependencies and the `surefire` plug-in configuration.

You now know how to change the platform in Spring DM tests. Table 10.4 summarizes the constants (for the `getPlatformName` method) and the `OsgiPlatform` implementation (for the system property) for each platform Spring DM supports.

This ends our tour of the advanced features of Spring DM's testing framework. We looked at the basics of this testing framework and how to test classic classloading and

Table 10.4 Settings for the OSGi platforms Spring DM test framework supports

Platform	Constants *	Implementation class *
Equinox	`Platforms.EQUINOX`	`EquinoxPlatform`
Felix	`Platforms.FELIX`	`FelixPlatform`
Knopflerfish	`Platforms.KNOPFLERFISH`	`KnopflerfishPlatform`

* From the `org.springframework.osgi.test.platform` package

service import scenarios in section 10.3.1, and in this section we covered the different hooks of the `AbstractConfigurableBundleCreatorTests` class, customizing the content of the on-the-fly bundle and of its manifest and changing the OSGi platform the tests are run under. Thanks to all the features the testing framework offers, writing integration tests for OSGi and Spring DM applications is the same as for any application.

10.4 Summary

Testing OSGi-based applications isn't much different than testing traditional applications. Thanks to the POJO programming model that Spring DM promotes, most parts of OSGi components can be tested with common testing techniques and tools, without knowing they're meant to be run in an OSGi container. So adopting OSGi won't require that you lose your testing habits (build tools, continuous integration, and so on).

Nevertheless, OSGi brings with it a set of features like package visibility and services that need to be tested within an OSGi container. That's why Spring DM provides powerful testing support, which makes OSGi integration tests easier to write and run. More importantly, they're very similar to regular Java tests, because Spring DM builds on top of JUnit. This support takes care of bootstrapping an OSGi container, provisioning it with bundles that the test class specifies, and running the test methods *in* the container. The test is then run in an environment that mimics as much as possible the target environment.

Spring DM's testing support not only makes available all of OSGi's features to test classes, it also adds some Spring goodies like dependency injection to the test. Spring DM still fulfills its role as a bridge between the Spring and OSGi worlds.

We're now done with Spring DM's testing support; the next chapter is dedicated to another part of OSGi for which Spring DM provides support: compendium services.

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]