

covers Spring 3.0

Spring IN ACTION

THIRD EDITION

Craig Walls



MANNING

Chapter 1. Springing into action.....	1
Section 1.1. Simplifying Java development.....	2
Section 1.2. Containing your beans.....	15
Section 1.3. Surveying the Spring landscape.....	18
Section 1.4. What's new in Spring.....	25
Section 1.5. Summary.....	27

Springing into action

This chapter covers

- Exploring Spring's core modules
- Decoupling application objects
- Managing cross-cutting concerns with AOP
- Spring's bean container

It all started with a bean.

In 1996, the Java programming language was still a young, exciting, up-and-coming platform. Many developers flocked to the language because they'd seen how to create rich and dynamic web applications using applets. They soon learned that there was more to this strange new language than animated juggling cartoon characters. Unlike any language before it, Java made it possible to write complex applications made up of discrete parts. They came for the applets, but they stayed for the components.

In December of that year, Sun Microsystems published the JavaBeans 1.00-A specification. JavaBeans defined a software component model for Java. This specification defined a set of coding policies that enabled simple Java objects to be reusable and easily composed into more complex applications. Although JavaBeans were intended as a general-purpose means of defining reusable application components,

they were primarily used as a model for building user interface widgets. They seemed too simple to be capable of any “real” work. Enterprise developers wanted more.

Sophisticated applications often require services such as transaction support, security, and distributed computing—services not directly provided by the JavaBeans specification. So in March 1998, Sun published version 1.0 of the Enterprise JavaBeans (EJB) specification. This specification extended the notion of Java components to the server side, providing much-needed enterprise services, but failed to continue the simplicity of the original JavaBeans specification. Except in name, EJB bears little resemblance to the original JavaBeans specification.

Despite the fact that many successful applications have been built based on EJB, EJB never achieved its intended purpose: to simplify enterprise application development. It’s true that EJB’s declarative programming model simplifies many infrastructural aspects of development, such as transactions and security. But in a different way, EJBs complicate development by mandating deployment descriptors and plumbing code (home and remote/local interfaces). Over time, many developers became disenchanted with EJB. As a result, its popularity has waned in recent years, leaving many developers looking for an easier way.

Today, Java component development has returned to its roots. New programming techniques, including aspect-oriented programming (AOP) and dependency injection (DI), are giving JavaBeans much of the power previously reserved for EJBs. These techniques furnish plain-old Java objects (POJOs) with a declarative programming model reminiscent of EJB, but without all of EJB’s complexity. No longer must you resort to writing an unwieldy EJB component when a simple JavaBean will suffice.

In fairness, even EJBs have evolved to promote a POJO-based programming model. Employing ideas such as DI and AOP, the latest EJB specification is significantly simpler than its predecessors. But for many developers, this move is too little, too late. By the time the EJB 3 specification had entered the scene, other POJO-based development frameworks had already established themselves as de facto standards in the Java community.

Leading the charge for lightweight POJO-based development is the Spring Framework, which we’ll explore throughout this book. In this chapter, we’ll explore the Spring Framework at a high level, giving you a taste of what Spring is about. This chapter will give you a good idea of the types of problems Spring solves, and will set the stage for the rest of the book. First things first—let’s find out what Spring is all about.

1.1 ***Simplifying Java development***

Spring is an open source framework, originally created by Rod Johnson and described in his book *Expert One-on-One: J2EE Design and Development*. Spring was created to address the complexity of enterprise application development, and makes it possible to use plain-vanilla JavaBeans to achieve things that were previously only possible with EJBs. But Spring’s usefulness isn’t limited to server-side development. Any Java application can benefit from Spring in terms of simplicity, testability, and loose coupling.

A BEAN BY ANY OTHER NAME... Although Spring uses the words *bean* and *JavaBean* liberally when referring to application components, this doesn't mean that a Spring component must follow the JavaBeans specification to the letter. A Spring component can be any type of POJO. In this book, I assume the loose definition of *JavaBean*, which is synonymous with POJO.

As you'll see throughout this book, Spring does many things. But at the root of almost everything Spring provides are a few foundational ideas, all focused on Spring's fundamental mission: *Spring simplifies Java development*.

That's a bold statement! A lot of frameworks claim to simplify something or other. But Spring aims to simplify the broad subject of Java development. This begs for more explanation. How does Spring simplify Java development?

To back up its attack on Java complexity, Spring employs four key strategies:

- Lightweight and minimally invasive development with plain old Java objects (POJOs)
- Loose coupling through dependency injection and interface orientation
- Declarative programming through aspects and common conventions
- Boilerplate reduction through aspects and templates

Almost everything Spring does can be traced back to one or more of these four strategies. Throughout the rest of this chapter, I'll expand on each of these ideas, showing concrete examples of how Spring makes good on its promise to simplify Java development. Let's start with seeing how Spring remains minimally invasive by encouraging POJO-oriented development.

1.1.1 Unleashing the power of POJOs

If you've been doing Java development for long, you've probably seen (and may have even worked with) frameworks that lock you in by forcing you to extend one of their classes or implement one of their interfaces. The classic example is that of an EJB 2-era stateless session bean. As you can see from this trivial `HelloWorldBean`, the EJB 2 specification made some rather heavy demands:

Listing 1.1 EJB 2.1 forced you to implement methods that weren't needed.

```
package com.habuma.ejb.session;

import javax.ejb.SessionBean;
import javax.ejb.SessionContext;

public class HelloWorldBean implements SessionBean {
    public void ejbActivate() {
    }

    public void ejbPassivate() {
    }

    public void ejbRemove() {
    }
}
```

Why are these methods needed?


```

public void setSessionContext(SessionContext ctx) {
}

public String sayHello() {
    return "Hello World";
}

public void ejbCreate() {
}
}

```

← EJB core business logic

The `SessionBean` interface would let you hook into the EJB's lifecycle by implementing several lifecycle callback methods (those methods that start with *ejb*). Or I should rephrase that to say that the `SessionBean` interface would *force* you to hook into the EJB's lifecycle, even if you didn't need to. The bulk of the code in `HelloWorldBean` is there solely for the sake of the framework. This raises the question: who's working for whom?

EJB 2 wasn't alone when it came to being invasive. Other popular frameworks such as the earlier versions of Struts, WebWork, and Tapestry imposed themselves upon otherwise simple Java classes. These heavyweight frameworks forced developers to write classes that were littered with unnecessary code, locked into their framework, and were often difficult to write tests against.

Spring avoids (as much as possible) littering your application code with its API. Spring almost never forces you to implement a Spring-specific interface or extend a Spring-specific class. Instead, the classes in a Spring-based application often have no indication that they're being used by Spring. At worst, a class may be annotated with one of Spring's annotations, but is otherwise a POJO.

To illustrate, if the `HelloWorldBean` class shown in listing 1.1 were to be rewritten to function as a Spring managed bean, it might look like this.

Listing 1.2 Spring doesn't make any unreasonable demands on `HelloWorldBean`.

```

package com.habuma.spring;

public class HelloWorldBean {
    public String sayHello() {
        return "Hello World";
    }
}

```

← This is all you needed

Isn't that better? Gone are all of those noisy lifecycle methods. `HelloWorldBean` doesn't implement, extend, or even import anything from the Spring API. `HelloWorldBean` is lean, mean, and in every sense of the phrase, a plain-old Java object.

Despite their simple form, POJOs can be powerful. One of the ways Spring empowers POJOs is by assembling them using dependency injection. Let's see how dependency injection can help keep application objects decoupled from each other.

1.1.2 Injecting dependencies

The phrase *dependency injection* may sound intimidating, conjuring up notions of a complex programming technique or design pattern. But as it turns out, DI isn't nearly

as complex as it sounds. By applying DI in your projects, you'll find that your code will become significantly simpler, easier to understand, and easier to test.

Any nontrivial application (pretty much anything more complex than a Hello World example) is made up of two or more classes that collaborate with each other to perform some business logic. Traditionally, each object is responsible for obtaining its own references to the objects it collaborates with (its dependencies). This can lead to highly coupled and hard-to-test code.

For example, consider the Knight class shown next.


Listing 1.3 A DamselRescuingKnight can only embark on RescueDamselQuests.

```
package com.springinaction.knights;

public class DamselRescuingKnight implements Knight {
    private RescueDamselQuest quest;

    public DamselRescuingKnight() {
        quest = new RescueDamselQuest();
    }

    public void embarkOnQuest() throws QuestException {
        quest.embark();
    }
}
```

 **Tightly coupled to
RescueDamselQuest**

As you can see, `DamselRescuingKnight` creates its own `quest`, a `RescueDamselQuest`, within the constructor. This makes a `DamselRescuingKnight` tightly coupled to a `RescueDamselQuest` and severely limits the knight's quest-embarking repertoire. If a damsel needs rescuing, this knight's there. But if a dragon needs slaying or a round table needs... well...rounding, then this knight's going to have to sit it out.

What's more, it'd be terribly difficult to write a unit test for `DamselRescuingKnight`. In such a test, you'd like to be able to assert that the `quest's embark()` method is called when the knight's `embarkOnQuest()` is called. But there's no clear way to accomplish that here. Unfortunately, `DamselRescuingKnight` will remain untested.

Coupling is a two-headed beast. On one hand, tightly coupled code is difficult to test, difficult to reuse, difficult to understand, and typically exhibits "whack-a-mole" bug behavior (fixing one bug results in the creation of one or more new bugs). On the other hand, a certain amount of coupling is necessary—completely uncoupled code doesn't do anything. In order to do anything useful, classes need to know about each other somehow. Coupling is necessary, but should be carefully managed.

With DI, on the other hand, objects are given their dependencies at creation time by some third party that coordinates each object in the system. Objects aren't expected to create or obtain their dependencies—dependencies are injected into the objects that need them.

To illustrate this point, let's look at `BraveKnight` in the following listing, a knight that's not only brave, but is capable of embarking on any kind of quest that comes along.

Listing 1.4 A BraveKnight is flexible enough to take on any Quest he's given

```
package com.springinaction.knights;

public class BraveKnight implements Knight {
    private Quest quest;

    public BraveKnight(Quest quest) {
        this.quest = quest;           ← Quest is injected
    }

    public void embarkOnQuest() throws QuestException {
        quest.embark();
    }
}
```

As you can see, unlike `DamselRescuingKnight`, `BraveKnight` doesn't create his own quest. Instead, he's given a quest at construction time as a constructor argument. This is a type of dependency injection known as *constructor injection*.

What's more, the quest he's given is typed as `Quest`, an interface that all quests implement. So `BraveKnight` could embark on a `RescueDamselQuest`, a `SlayDragonQuest`, a `MakeRoundTableRounderQuest`, or any other `Quest` implementation he's given.

The point here is that `BraveKnight` isn't coupled to any specific implementation of `Quest`. It doesn't matter to him what kind of quest he's asked to embark upon, so long as it implements the `Quest` interface. That's the key benefit of DI—loose coupling. If an object only knows about its dependencies by their interface (not by their implementation or how they're instantiated), then the dependency can be swapped out with a different implementation without the depending object knowing the difference.

One of the most common ways that a dependency will be swapped out is with a mock implementation during testing. You were unable to adequately test `DamselRescuingKnight` due to tight coupling, but you can easily test `BraveKnight` by giving it a mock implementation of `Quest`, as shown next.

Listing 1.5 To test BraveKnight, you'll inject it with a mock Quest .

```
package com.springinaction.knights;

import static org.mockito.Mockito.*;

import org.junit.Test;

public class BraveKnightTest {
    @Test
    public void knightShouldEmbarkOnQuest() throws QuestException {
        Quest mockQuest = mock(Quest.class);           ← Create mock Quest

        BraveKnight knight = new BraveKnight(mockQuest); ← Inject mock Quest
        knight.embarkOnQuest();

        verify(mockQuest, times(1)).embark();
    }
}
```


Here you're using a mock object framework known as *Mockito* to create a mock implementation of the *Quest* interface. With the mock object in hand, you create a new instance of *BraveKnight*, injecting the mock *Quest* via the constructor. After calling the `embarkOnQuest()` method, you ask Mockito to verify that the mock *Quest*'s `embark()` method was called exactly once.

INJECTING A QUEST INTO A KNIGHT

Now that your *BraveKnight* class is written in such a way that you can give him any quest you want, how can you specify which *Quest* to give him?

The act of creating associations between application components is commonly referred to as *wiring*. In Spring, there are many ways to wire components together, but a common approach has always been via XML. The following listing shows a simple Spring configuration file, `knights.xml`, that gives a *BraveKnight* a *SlayDragonQuest*.

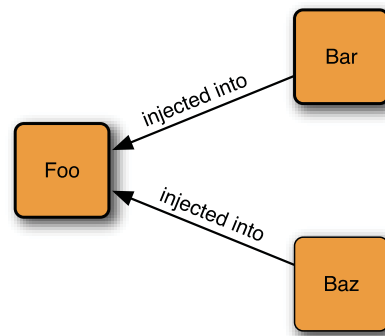


Figure 1.1 Dependency injection involves giving an object its dependencies as opposed to an object having to acquire those dependencies on its own.

Listing 1.6 Injecting a *SlayDragonQuest* into a *BraveKnight* with Spring

```

<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

  <bean id="knight" class="com.springinaction.knights.BraveKnight">
    <constructor-arg ref="quest" />
  </bean>

  <bean id="quest"
        class="com.springinaction.knights.SlayDragonQuest" />
</beans>

```

Inject quest bean

Create SlayDragonQuest

This is a simple approach to wiring beans in Spring. Don't concern yourself too much with the details right now. We'll dig more into Spring configuration and see what's going on when we get to chapter 2. We'll also look at other ways that we can wire beans in Spring.

Now that you've declared the relationship between *BraveKnight* and a *Quest*, you need to load up the XML configuration file and kick off the application.

SEEING IT WORK

In a Spring application, an *application context* loads bean definitions and wires them together. The Spring application context is fully responsible for the creation of and wiring of the objects that make up the application. Spring comes with several implementations of its application context, each primarily differing only in how they load their configuration.

Because the beans in `knights.xml` are declared in an XML file, an appropriate choice for application context might be `ClassPathXmlApplicationContext`. This Spring context implementation loads the Spring context from one or more XML files located in the application's classpath. The `main()` method in the following listing uses `ClassPathXmlApplicationContext` to load `knights.xml` and to get a reference to the `Knight` object.

Listing 1.7 KnightMain.java loads the Spring context containing a knight.

```
package com.springinaction.knights;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class KnightMain {
    public static void main(String[] args) {
        ApplicationContext context =
            new ClassPathXmlApplicationContext("knights.xml");

        Knight knight = (Knight) context.getBean("knight");

        knight.embarkOnQuest();
    }
}
```



Here the `main()` method creates the Spring application context based on the `knights.xml` file. Then it uses the application context as a factory to retrieve the bean whose ID is *knight*. With a reference to the `Knight` object, it calls the `embarkOnQuest()` method to have the knight embark on the quest that it was given. Note that this class knows nothing about which type of `Quest` our hero has. For that matter, it's blissfully unaware of the fact that it's dealing with `BraveKnight`. Only the `knights.xml` file knows for sure what the implementations are.

And with that you have a quick introduction to dependency injection. You'll see a lot more DI throughout this book. But if you want even more dependency injection, I encourage you to have a look at Dhanji R. Prasanna's *Dependency Injection*, which covers DI in fine detail.

But now let's have a look at another of Spring's Java-simplifying strategies: declarative programming through aspects.

1.1.3 Applying aspects

Although DI makes it possible to tie software components together loosely, aspect-oriented programming enables you to capture functionality that's used throughout your application in reusable components.

Aspect-oriented programming is often defined as a technique that promotes separation of concerns within a software system. Systems are composed of several components, each responsible for a specific piece of functionality. Often these components also carry additional responsibility beyond their core functionality. System services such as logging, transaction management, and security often find their way into

components whose core responsibility is something else. These system services are commonly referred to as *cross-cutting concerns* because they tend to cut across multiple components in a system.

By spreading these concerns across multiple components, you introduce two levels of complexity to your code:

- The code that implements the systemwide concerns is duplicated across multiple components. This means that if you need to change how those concerns work, you'll need to visit multiple components. Even if you've abstracted the concern to a separate module so that the impact to your components is a single method call, that method call is duplicated in multiple places.
- Your components are littered with code that isn't aligned with their core functionality. A method to add an entry to an address book should only be concerned with how to add the address and not with whether it's secure or transactional.

Figure 1.2 illustrates this complexity. The business objects on the left are too intimately involved with the system services. Not only does each object know that it's being logged, secured, and involved in a transactional context, but also each object is responsible for performing those services for itself.

AOP makes it possible to modularize these services and then apply them declaratively to the components that they should affect. This results in components that are more cohesive and that focus on their own specific concerns, completely ignorant of any system services that may be involved. In short, aspects ensure that POJOs remain plain.

It may help to think of aspects as blankets that cover many components of an application, as illustrated in figure 1.3. At its core, an application consists of modules that implement business functionality. With AOP, you can then cover your core application with layers of functionality. These layers can be applied declaratively throughout your application in a flexible manner without your core application even knowing they

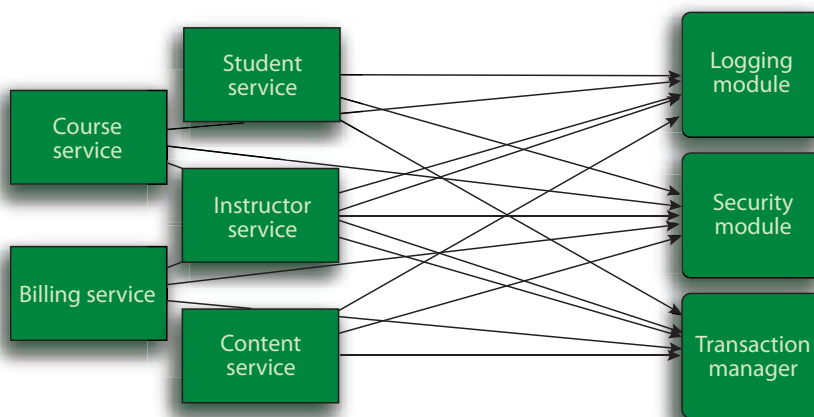


Figure 1.2 Calls to systemwide concerns such as logging and security are often scattered about in modules where those concerns are not their primary concern.

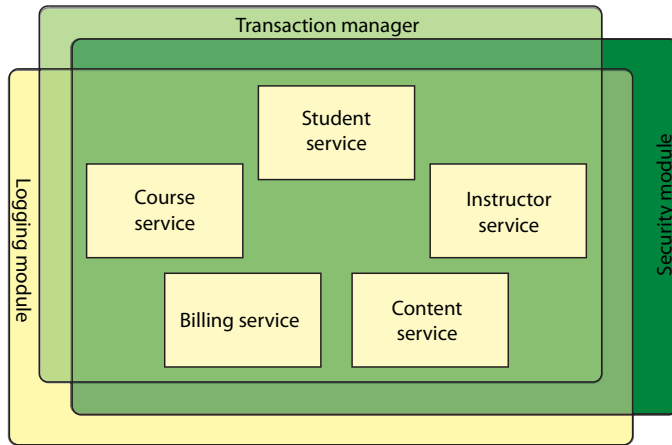


Figure 1.3 Using AOP, systemwide concerns blanket the components that they impact. This leaves the application components to focus on their specific business functionality.

exist. This is a powerful concept, as it keeps the security, transaction, and logging concerns from littering the application's core business logic.

To demonstrate how aspects can be applied in Spring, let's revisit the knight example, adding a basic Spring aspect to the mix.

AOP IN ACTION

Anyone who knows anything about knights only knows about them because their deeds were chronicled in song by the musically inclined storytellers known as minstrels. Let's suppose that you want to record the comings and goings of your Brave-Knight using the services of a minstrel. The following shows the `Minstrel` class you might use.

Listing 1.8 A Minstrel is a musically inclined logging system of medieval times

```
package com.springinaction.knights;

public class Minstrel {
    public void singBeforeQuest() {
        System.out.println("Fa la la; The knight is so brave!");
    }

    public void singAfterQuest() {
        System.out.println(
            "Tee hee he; The brave knight did embark on a quest!");
    }
}
```

← Called before quest

← Called after quest

As you can see, `Minstrel` is a simple class with two methods. The `singBeforeQuest()` method is intended to be invoked before a knight embarks on a quest, and the `singAfterQuest()` method should be invoked after the knight has completed a quest. It should be simple to work this into your code, so let's make the appropriate tweaks to `BraveKnight` to use the `Minstrel`. The following listing shows a first attempt.

Listing 1.9 A BraveKnight that must call Minstrel methods

```
package com.springinaction.knights;

public class BraveKnight implements Knight {
    private Quest quest;
    private Minstrel minstrel;

    public BraveKnight(Quest quest, Minstrel minstrel) {
        this.quest = quest;
        this.minstrel = minstrel;
    }

    public void embarkOnQuest() throws QuestException {
        minstrel.singBeforeQuest();
        quest.embark();
        minstrel.singAfterQuest();
    }
}
```

Should knight
manage its own
Minstrel?

That should do the trick. But something doesn't seem right here. Is it really within the knight's range of concern to manage his minstrel? It seems to me that a minstrel should just do his job without the knight asking him to do so. After all, that's the minstrel's job—to sing about the knight's endeavors. Why should the knight have to keep reminding the minstrel to do his job?

Furthermore, because the knight needs to know about the minstrel, you're forced to inject the Minstrel into the BraveKnight. This not only complicates the BraveKnight's code, but also makes me wonder if you'd ever want a knight who didn't have a minstrel. What if the Minstrel is null? Should we introduce some null-checking logic to cover that case?

Your simple BraveKnight class is starting to get more complicated and would become more so if you were to handle the nullMinstrel scenario. But using AOP, you can declare that the minstrel should sing about a knight's quests and free the knight from having to deal with the Minstrel methods directly.

To turn Minstrel into an aspect, all you need to do is declare it as one in the Spring configuration file. Here's the updated knights.xml file, revised to declare Minstrel as an aspect.

Listing 1.10 Declaring the Minstrel as an aspect

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:aop="http://www.springframework.org/schema/aop"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
        http://www.springframework.org/schema/aop
        http://www.springframework.org/schema/aop/spring-aop-3.0.xsd">

    <bean id="knight" class="com.springinaction.knights.BraveKnight">
        <constructor-arg ref="quest" />
    </bean>
```



```

<bean id="quest"
      class="com.springinaction.knights.SlayDragonQuest" />

<bean id="minstrel"
      class="com.springinaction.knights.Minstrel" />

<aop:config>
  <aop:aspect ref="minstrel">

    <aop:pointcut id="embark"
      expression="execution(* *.embarkOnQuest(..))" />

    <aop:before pointcut-ref="embark"
      method="singBeforeQuest" />

    <aop:after pointcut-ref="embark"
      method="singAfterQuest" />

  </aop:aspect>
</aop:config>
</beans>

```

Here you're using Spring's `aop` configuration namespace to declare that the `Minstrel` bean is an aspect. First, you had to declare the `Minstrel` as a bean. Then you refer to that bean in the `<aop:aspect>` element. Defining the aspect further, you declare (using `<aop:before>`) that before the `embarkOnQuest()` method is executed, the `Minstrel`'s `singBeforeQuest()` should be called. This is called *before advice*. And you (using `<aop:after>`) declare that the `singAfterQuest()` method should be called after `embarkOnQuest()` has executed. This is known as *after advice*.

In both cases, the `pointcut-ref` attribute refers to a pointcut named *embark*. This pointcut is defined in the preceding `<pointcut>` element with an `expression` attribute set to select where the advice should be applied. The expression syntax is AspectJ's pointcut expression language.

Don't worry if you don't know AspectJ or the details of how AspectJ pointcut expressions are written. We'll talk more about Spring AOP later in chapter 4. For now it's enough to know that you've asked Spring to call the `Minstrel`'s `singBeforeQuest()` and `singAfterQuest()` methods before and after the `BraveKnight` embarks on a quest.

That's all there is to it! With a tiny bit of XML, you've just turned `Minstrel` into a Spring aspect. Don't worry if this doesn't make complete sense yet—you'll see plenty more examples of Spring AOP in chapter 4 that should help clear this up. For now, there are two important points to take away from this example.

First, `Minstrel` is still a POJO—nothing about it indicates that it's to be used as an aspect. Instead `Minstrel` became an aspect when we declared it as such in the Spring context.

Second, and most important, `Minstrel` can be applied to the `BraveKnight` without the `BraveKnight` needing to explicitly call on it. In fact, `BraveKnight` remains completely unaware of the `Minstrel`'s existence.

I should also point out that although you used some Spring magic to turn `Minstrel` into an aspect, it was declared as a Spring `<bean>` first. The point here is that

you can do anything with Spring aspects that you can do with other Spring beans, such as injecting them with dependencies.

Using aspects to sing about knights can be fun. But Spring's AOP can be used for even more practical things. As you'll see later, Spring AOP can be employed to provide services such as declarative transactions (chapter 6) and security (chapter 9).

But for now, let's look at one more way that Spring simplifies Java development.

1.1.4 Eliminating boilerplate code with templates

Have you ever written some code and then felt like you'd already written the same code before? That's not *déjà vu*, my friend. That's boilerplate code—the code that you often have to write over and over again to accomplish common and otherwise simple tasks.

Unfortunately, there are a lot of places where Java APIs involve a bunch of boilerplate code. A common example of boilerplate code can be seen when working with JDBC to query data from a database. For example, if you've ever worked with JDBC before, then you've probably written something similar to the following.

Listing 1.11 Many Java APIs, such as JDBC, involve writing a lot of boilerplate code.

```
public Employee getEmployeeById(long id) {
    Connection conn = null;
    PreparedStatement stmt = null;
    ResultSet rs = null;
    try {
        conn = dataSource.getConnection();
        stmt = conn.prepareStatement(
            "select id, firstname, lastname, salary from " +
            "employee where id=?");
        stmt.setLong(1, id);
        rs = stmt.executeQuery();
        Employee employee = null;
        if (rs.next()) {
            employee = new Employee();
            employee.setId(rs.getLong("id"));
            employee.setFirstName(rs.getString("firstname"));
            employee.setLastName(rs.getString("lastname"));
            employee.setSalary(rs.getBigDecimal("salary"));
        }
        return employee;
    } catch (SQLException e) {
    } finally {
        if(rs != null) {
            try {
                rs.close();
            } catch(SQLException e) {}
        }

        if(stmt != null) {
            try {
                stmt.close();
            }
        }
    }
}
```

← Select employee

← Create object from data

← What should be done here?

← Clean up mess

```

        } catch(SQLException e) {}
    }

    if(conn != null) {
        try {
            conn.close();
        } catch(SQLException e) {}
    }
}

return null;
}

```

As you can see, this JDBC code queries the database for an employee's name and salary. But I'll bet you had to look hard to see that. That's because the small bit of code that's specific to querying for an employee is buried in a heap of JDBC ceremony. You first have to create a connection, then a statement, and then finally you can query for the results. And, to appease JDBC's anger, you must catch `SQLException`, a checked exception, even though there's not a lot you can do if it's thrown.

Finally, after all is said and done, you have to clean up the mess, closing down the connection, statement, and result set. This could also stir JDBC's anger. Therefore you must catch `SQLException` here as well.

What's most notable about listing 1.11 is that much of it is the exact same code that you'd write for pretty much any JDBC operation. Little of it has anything to do with querying for an employee, and much of it is JDBC boilerplate.

JDBC's not alone in the boilerplate code business. Many activities often require similar boilerplate code. JMS, JNDI, and the consumption of REST services often involve a lot of commonly repeated code.

Spring seeks to eliminate boilerplate code by encapsulating it in templates. Spring's `JdbcTemplate` makes it possible to perform database operations without all of the ceremony required by traditional JDBC.

For example, using Spring's `SimpleJdbcTemplate` (a specialization of `JdbcTemplate` that takes advantage of Java 5 features), the `getEmployeeById()` method can be rewritten so that its focus is on the task of retrieving employee data and not catering to the demands of the JDBC API. The following shows what such an updated `getEmployeeById()` method might look like.

Listing 1.12 Templates let your code focus on the task at hand.

```

public Employee getEmployeeById(long id) {
    return jdbcTemplate.queryForObject(
        "select id, firstname, lastname, salary " +
        "from employee where id=?",
        new RowMapper<Employee>() {
            public Employee mapRow(ResultSet rs,
                int rowNum) throws SQLException {
                Employee employee = new Employee();
                employee.setId(rs.getLong("id"));
                employee.setFirstName(rs.getString("firstname"));
            }
        }
    );
}

```

← SQL query

← Map results to object

```

        employee.setLastName(rs.getString("lastname"));
        employee.setSalary(rs.getBigDecimal("salary"));
        return employee;
    }
},
id);
}

```

Specify query
parameter

As you can see, this new version of `getEmployeeById()` is much simpler and acutely focused on selecting an employee from the database. The template's `queryForObject()` method is given the SQL query, a `RowMapper` (for mapping result set data to a domain object), and zero or more query parameters. What you don't see in `getEmployeeById()` is any of the JDBC boilerplate from before. It's all handled internal to the template.

I've shown you how Spring attacks complexity in Java development using POJO-oriented development, dependency injection, AOP, and templates. Along the way I showed you how to configure beans and aspects in XML-based configuration files. But how do those files get loaded? And what are they loaded into? Let's look at the Spring container, the place where your application's beans will reside.

1.2 Containing your beans

In a Spring-based application, your application objects will live within the Spring container. As illustrated in figure 1.4, the container will create the objects, wire them together, configure them, and manage their complete lifecycle from cradle to grave (or new to `finalize()`, as the case may be).

In the next chapter, you'll see how to configure Spring to know what objects it should create, configure, and wire together. First, it's important to get to know the container where your objects will be hanging out. Understanding the container helps you grasp how your objects will be managed.

The container is at the core of the Spring Framework. Spring's container uses dependency injection (DI) to manage the components that make up an application. This includes creating associations between collaborating components. As such, these objects are cleaner and easier to understand, support reuse, and are easy to unit test.

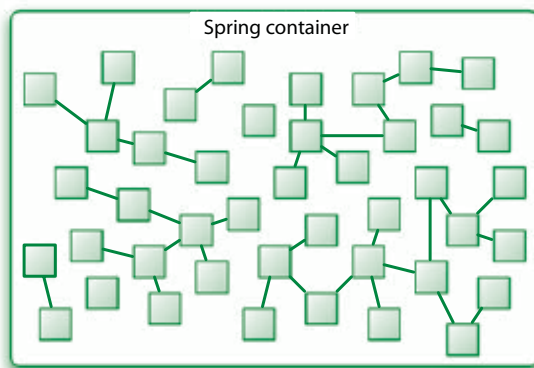


Figure 1.4 In a Spring application, objects are created, wired together, and live within the Spring container.

There's no single Spring container. Spring comes with several container implementations that can be categorized into two distinct types. *Bean factories* (defined by the `org.springframework.beans.factory.BeanFactory` interface) are the simplest of containers, providing basic support for DI. *Application contexts* (defined by the `org.springframework.context.ApplicationContext` interface) build on the notion of a bean factory by providing application framework services, such as the ability to resolve textual messages from a properties file and the ability to publish application events to interested event listeners.

Although it's possible to work with Spring using either bean factories or application contexts, bean factories are often too low-level for most applications. Therefore, application contexts are preferred over bean factories. We'll focus on working with application contexts and not spend any more time talking about bean factories.

1.2.1 Working with an application context

Spring comes with several flavors of application context. The three that you'll most likely encounter are

- `ClassPathXmlApplicationContext`—Loads a context definition from an XML file located in the classpath, treating context definition files as classpath resources.
- `FileSystemXmlApplicationContext`—Loads a context definition from an XML file in the file system.
- `XmlWebApplicationContext`—Loads context definitions from an XML file contained within a web application.

We'll talk more about `XmlWebApplicationContext` in chapter 7 when we discuss web-based Spring applications. For now, let's simply load the application context from the file system using `FileSystemXmlApplicationContext` or from the classpath using `ClassPathXmlApplicationContext`.

Loading an application context from the file system or from the classpath is similar to how you load beans into a bean factory. For example, here's how you'd load a `FileSystemXmlApplicationContext`:

```
ApplicationContext context = new
    FileSystemXmlApplicationContext("c:/foo.xml");
```

Similarly, you can load an application context from within the application's classpath using `ClassPathXmlApplicationContext`:

```
ApplicationContext context = new
    ClassPathXmlApplicationContext("foo.xml");
```

The difference between using `FileSystemXmlApplicationContext` and `ClassPathXmlApplicationContext` is that `FileSystemXmlApplicationContext` will look for `foo.xml` in a specific location within the file system, whereas `ClassPathXmlApplicationContext` will look for `foo.xml` anywhere in the classpath (including JAR files).

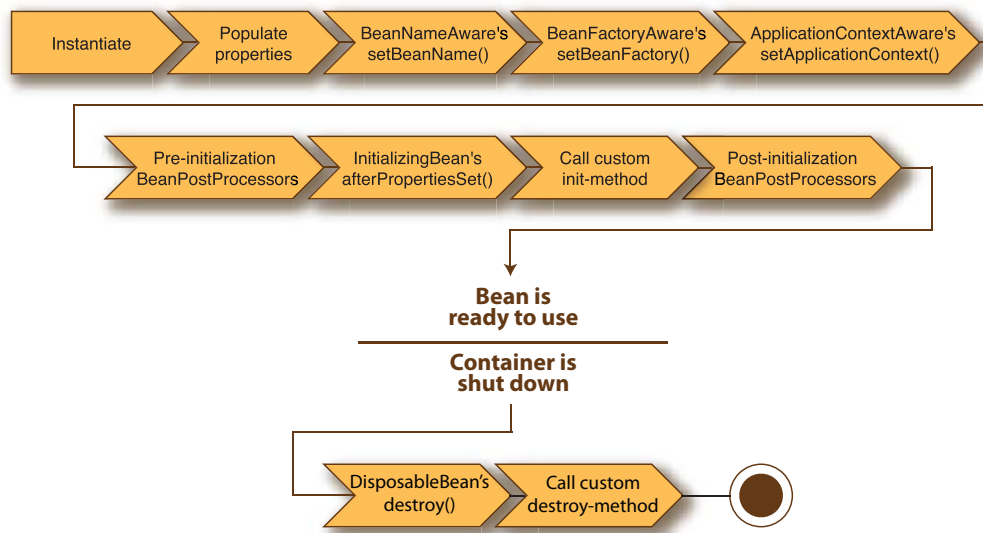


Figure 1.5 A bean goes through several steps between creation and destruction in the Spring container. Each step is an opportunity to customize how the bean is managed in Spring.

With an application context in hand, you can retrieve beans from the Spring container by calling the context's `getBean()` method.

Now that you know the basics of how to create a Spring container, let's take a closer look at the lifecycle of a bean in the bean container.

1.2.2 A bean's life

In a traditional Java application, the lifecycle of a bean is simple. Java's `new` keyword is used to instantiate the bean (or perhaps it's deserialized) and it's ready to use. Once the bean is no longer in use, it's eligible for garbage collection and eventually goes to the big bit bucket in the sky.

In contrast, the lifecycle of a bean within a Spring container is more elaborate. It's important to understand the lifecycle of a Spring bean, because you may want to take advantage of some of the opportunities that Spring offers to customize how a bean is created. Figure 1.5 shows the startup lifecycle of a typical bean as it's loaded into a Spring application context.

As you can see, a bean factory performs several setup steps before a bean is ready to use. Breaking down figure 1.5 in more detail:

- 1 Spring instantiates the bean.
- 2 Spring injects values and bean references into the bean's properties.
- 3 If the bean implements `BeanNameAware`, Spring passes the bean's ID to the `setBeanName()` method.
- 4 If the bean implements `BeanFactoryAware`, Spring calls the `setBeanFactory()` method, passing in the bean factory itself.

- 5 If the bean implements `ApplicationContextAware`, Spring will call the `setApplicationContext()` method, passing in a reference to the enclosing application context.
- 6 If any of the beans implement the `BeanPostProcessor` interface, Spring calls their `postProcessBeforeInitialization()` method.
- 7 If any beans implement the `InitializingBean` interface, Spring calls their `afterPropertiesSet()` method. Similarly, if the bean was declared with an `init`-method, then the specified initialization method will be called.
- 8 If there are any beans that implement `BeanPostProcessor`, Spring will call their `postProcessAfterInitialization()` method.
- 9 At this point, the bean is ready to be used by the application and will remain in the application context until the application context is destroyed.
- 10 If any beans implement the `DisposableBean` interface, then Spring will call their `destroy()` methods. Likewise, if any bean was declared with a `destroy`-method, then the specified method will be called.

Now you know how to create and load a Spring container. But an empty container isn't much good by itself; it doesn't contain anything unless you put something in it. To achieve the benefits of Spring DI, we must wire our application objects into the Spring container. We'll go into bean wiring in more detail in chapter 2.

But first, let's survey the modern Spring landscape to see what the Spring Framework is made up of and to see what the latest versions of Spring have to offer.

1.3 *Surveying the Spring landscape*

As you've seen, the Spring Framework is focused on simplifying enterprise Java development through dependency injection, aspect-oriented programming, and boilerplate reduction. Even if that were all that Spring did, it'd be worth using. But there's more to Spring than meets the eye.

Within the Spring Framework proper, you'll find several ways that Spring can ease Java development. But beyond the Spring Framework itself is a greater ecosystem of projects that build upon the core framework, extending Spring into areas such as web services, OSGi, Flash, and even .NET.

Let's first break down the core Spring Framework to see what it brings to the table. Then we'll expand our sights to review the other members of the greater Spring portfolio.

1.3.1 *Spring modules*

The Spring Framework is composed of several distinct modules. When you download and unzip the Spring Framework distribution, you'll find 20 different JAR files in the `dist` directory, as shown in figure 1.6.

The 20 JAR files that make up Spring can be arranged in one of six different categories of functionality, as illustrated in figure 1.7.

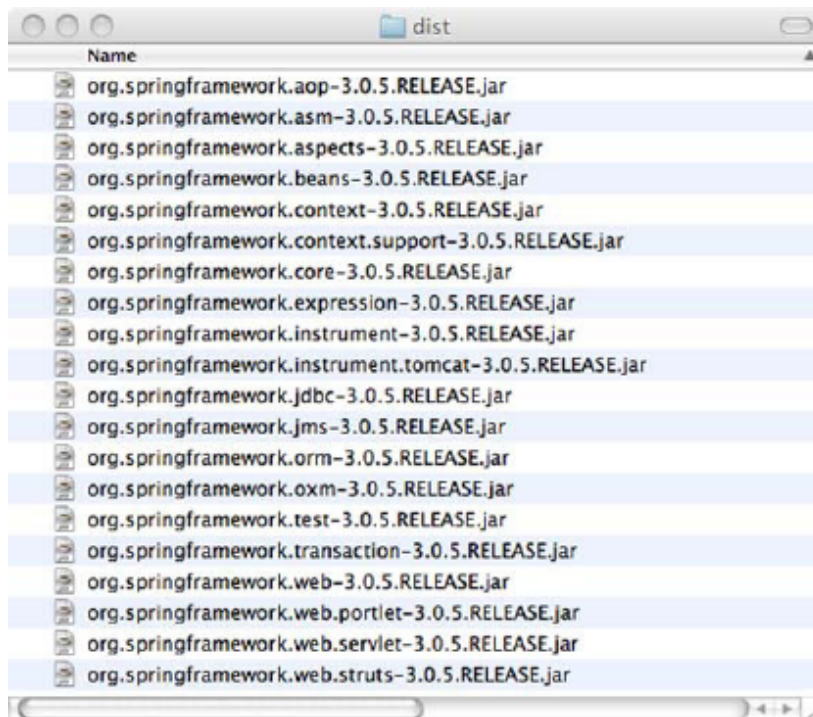


Figure 1.6 The JAR files that come with the Spring Framework distribution

When taken as a whole, these modules give you everything you need to develop enterprise-ready applications. But you don't have to base your application fully on the Spring Framework. You're free to choose the modules that suit your application and

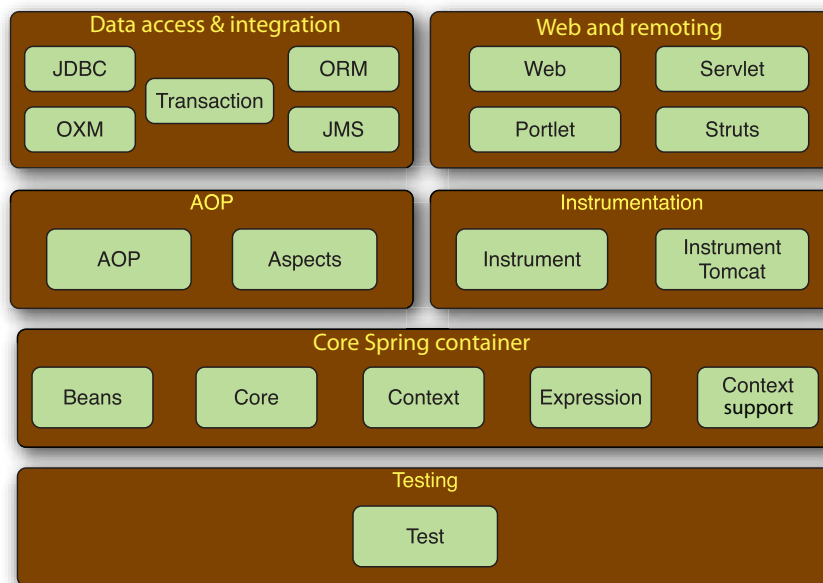


Figure 1.7 The Spring Framework is made up of six well-defined modules.

look to other options when Spring doesn't fit the bill. Spring even offers integration points with several other frameworks and libraries so that you won't have to write them yourself.

Let's take a look at each of Spring's modules, one at a time, to see how each fits in the overall Spring picture.

CORE SPRING CONTAINER

The centerpiece of the Spring Framework is a container that manages how the beans in a Spring-enabled application are created, configured, and managed. Within this module you'll find the Spring bean factory, which is the portion of Spring that provides dependency injection. Building upon the bean factory, you'll find several implementations of Spring's application context, each of which provides a different way to configure Spring.

In addition to the bean factory and application context, this module also supplies many enterprise services such as email, JNDI access, EJB integration, and scheduling.

As you can see, all of Spring's modules are built on top of the core container. You'll implicitly use these classes when you configure your application. We'll discuss the core module throughout this book, starting in chapter 2 where we dig deep into Spring dependency injection.

SPRING'S AOP MODULE

Spring provides rich support for aspect-oriented programming in its AOP module. This module serves as the basis for developing your own aspects for your Spring-enabled application. Like DI, AOP supports loose coupling of application objects. But with AOP, application-wide concerns (such as transactions and security) are decoupled from the objects to which they're applied.

We'll dig into Spring's AOP support in chapter 4.

DATA ACCESS AND INTEGRATION

Working with JDBC often results in a lot of boilerplate code that gets a connection, creates a statement, processes a result set, and then closes the connection. Spring's JDBC and *data access objects* (DAO) module abstracts away the boilerplate code so that you can keep your database code clean and simple, and prevents problems that result from a failure to close database resources. This module also builds a layer of meaningful exceptions on top of the error messages given by several database servers. No more trying to decipher cryptic and proprietary SQL error messages!

For those who prefer using an *object-relational mapping* (ORM) tool over straight JDBC, Spring provides the ORM module. Spring's ORM support builds on the DAO support, providing a convenient way to build DAOs for several ORM solutions. Spring doesn't attempt to implement its own ORM solution, but does provide hooks into several popular ORM frameworks, including Hibernate, Java Persistence API, Java Data Objects, and iBATIS SQL Maps. Spring's transaction management supports each of these ORM frameworks as well as JDBC.

We'll see how Spring's template-based JDBC abstraction can greatly simplify JDBC code when we look at Spring data access in chapter 5.

This module also includes a Spring abstraction over the Java Message Service (JMS) for asynchronous integration with other applications through messaging. And, as of Spring 3.0, this module includes the object-to-XML mapping features that were originally part of the Spring Web Services project.

In addition, this module uses Spring's AOP module to provide transaction management services for objects in a Spring application. We'll look at Spring's transaction support in detail in chapter 6.

WEB AND REMOTING

The *Model-View-Controller* (MVC) paradigm is a commonly accepted approach to building web applications such that the user interface is separate from the application logic. Java has no shortage of MVC frameworks, with Apache Struts, JSF, WebWork, and Tapestry among the most popular MVC choices.

Even though Spring integrates with several popular MVC frameworks, its web and remoting module comes with a capable MVC framework that promotes Spring's loosely coupled techniques in the web layer of an application. This framework comes in two forms: a servlet-based framework for conventional web applications and a portlet-based application for developing against the Java portlet API.

In addition to user-facing web applications, this module also provides several remoting options for building applications that interact with other applications. Spring's remoting capabilities include *Remote Method Invocation* (RMI), Hessian, Burlap, JAX-WS, and Spring's own HTTP invoker.

We'll look at Spring's MVC framework in chapter 7. Then, in chapter 10, we'll check out Spring remoting.

TESTING

Recognizing the importance of developer-written tests, Spring provides a module dedicated to testing Spring applications.

Within this module you'll find a collection of mock object implementations for writing unit tests against code that works with JNDI, servlets, and portlets. For integration-level testing, this module provides support for loading a collection of beans in a Spring application context and working with the beans in that context.

We'll get our first taste of Spring's testing module in chapter 4. Then in chapters 5 and 6, we'll expand on what we've learned by seeing how to test Spring data access and transactions.

1.3.2 The Spring portfolio

When it comes to Spring, there's more than meets the eye. In fact, there's more than what comes in the Spring Framework download. If we stopped at just the core Spring Framework, we'd miss out on a wealth of potential afforded by the larger Spring portfolio. The whole Spring portfolio includes several frameworks and libraries that build upon the core Spring Framework and upon each other. All together, the entire Spring portfolio brings the Spring programming model to almost every facet of Java development.

It would take several volumes to cover everything that the Spring portfolio has to offer, and much of it's outside the scope of this book. But we'll look at some of the elements of the Spring portfolio. Meanwhile, here's a taste of what lies beyond the core Spring Framework.

SPRING WEB FLOW

Spring Web Flow builds upon Spring's core MVC framework to provide support for building conversational, flow-based web applications that guide users toward a goal (think wizards or shopping carts). We'll talk more about Spring Web Flow in chapter 8, and you can learn more about Spring Web Flow from its home page at <http://www.springsource.org/webflow>.

SPRING WEB SERVICES

Although the core Spring Framework provides for declaratively publishing Spring beans as web services, those services are based on an arguably architecturally inferior contract-last model. The contract for the service is determined from the bean's interface. Spring Web Services offers a contract-first web services model where service implementations are written to satisfy the service contract.

I won't be talking about Spring-WS in this book, but you can read more about it from its home page at <http://static.springsource.org/spring-ws/sites/2.0>.

SPRING SECURITY

Security is a critical aspect of many applications. Implemented using Spring AOP, Spring Security offers a declarative security mechanism for Spring-based applications. We'll see how add Spring Security to applications in chapter 9. For further exploration, Spring Security's home page is at <http://static.springsource.org/spring-security/site>.

SPRING INTEGRATION

Many enterprise applications must interact with other enterprise applications. Spring Integration offers implementations of several common integration patterns in Spring's declarative style.

We won't cover Spring Integration in this book. But if you want more information on Spring Integration, have a look at *Spring Integration in Action* by Mark Fisher, Jonas Partner, Marius Bogoevici, and Iwein Fuld. Or you can visit the Spring Integration home page at <http://www.springsource.org/spring-integration>.

SPRING BATCH

When it's necessary to perform bulk operations on data, nothing beats batch processing. If you're going to be developing a batch application, you can leverage Spring's robust, POJO-oriented development model to do it using Spring Batch.

Spring Batch is outside of the scope of this book. But Thierry Templier and Arnaud Cogoluègnes will enlighten you in their book, *Spring Batch in Action*. You can also learn about Spring Batch from its home page at <http://static.springsource.org/spring-batch>.

SPRING SOCIAL

Social networking is a rising trend on the internet, and more and more applications are being outfitted with integration into social networking sites such as Facebook and Twitter. If this is the kind of thing that interests you, then you'll want to look at Spring Social, a social networking extension to Spring.

Spring Social is relatively new and didn't make it into this book. But you can find out more about it at <http://www.springsource.org/spring-social>.

SPRING MOBILE

Mobile applications are another significant area of software development. Smartphones and tablet devices are taking over as the preferred client for many users. Spring Mobile is a new extension to Spring to support development of mobile web applications.

Related to Spring Mobile is the Spring Android project. This new project, less than a month old as I write this, aims to bring some of the simplicity afforded by the Spring Framework to development of native applications for Android-based devices. Initially, this project is offering a version of Spring's `RestTemplate` (see chapter 11 to learn about `RestTemplate`) that can be used within an Android application.

Again, these projects are new and are outside of the scope of *Spring in Action*. But you can learn more about them at <http://www.springsource.org/spring-mobile> and <http://www.springsource.org/spring-android>.

SPRING DYNAMIC MODULES

Spring Dynamic Modules (Spring-DM) blends Spring's declarative dependency injection with OSGi's dynamic modularity. Using Spring-DM, you can build applications that are composed of several distinct, highly cohesive, loosely coupled modules that publish and consume services declaratively within the OSGi framework.

It should be noted that due to its tremendous impact in the world of OSGi, the Spring-DM model for declarative OSGi services has been formalized into the OSGi specification itself as the *OSGi Blueprint Container*. In addition, SpringSource has transitioned Spring-DM to the Eclipse project as a part of the Gemini family of OSGi projects and is now known as Gemini Blueprint.

SPRING LDAP

In addition to dependency injection and AOP, another common technique applied throughout the Spring Framework is to create template-based abstractions around unnecessarily complex operations such as JDBC queries or JMS messaging. Spring LDAP brings Spring-style template-based access to LDAP, eliminating the boilerplate code that's commonly involved in LDAP operations.

More information on Spring LDAP can be found at <http://www.springsource.org/ldap>.

SPRING RICH CLIENT

Web-based applications seem to have stolen the spotlight from traditional desktop applications. But if you're one of the few still developing Swing applications, you'll

want to check out Spring Rich Client, a rich application toolkit that brings the power of Spring to Swing.

SPRING.NET

You don't have to abandon dependency injection and AOP if you're put on a .NET project. Spring.NET offers the same loose-coupling and aspect-oriented features of Spring, but for the .NET platform.

In addition to the core DI and AOP functionality, Spring.NET comes with several modules for simplifying .NET development, including modules for working with ADO.NET, NHibernate, ASP.NET, and MSMQ.

To learn more about Spring .NET, visit <http://www.springframework.net>.

SPRING-FLEX

Adobe's Flex and AIR offer one of the most powerful options for rich internet application development. When those rich user interfaces need to interact with Java code on the server side, they can use a remoting and messaging technology known as *BlazeDS*. The Spring-Flex integration package enables Flex and AIR applications to communicate with server-side Spring beans using BlazeDS. It also includes an addon for Spring Roo to enable rapid application development of Flex applications.

You may begin your exploration of Spring Flex at <http://www.springsource.org/spring-flex>. You may also want to check out Spring ActionScript at <http://www.springactionscript.org>, which offers many benefits of the Spring Framework in ActionScript.

SPRING ROO

As more and more developers are basing their work on Spring, a set of common idioms and best practices has emerged around Spring and its related frameworks. At the same time, frameworks such as Ruby on Rails and Grails have arisen with a script-driven development model that makes simple work of building applications.

Spring Roo provides an interactive tooling environment that enables rapid development of Spring applications, pulling together the best practices that have been identified over the past few years.

What differentiates Roo from these other rapid application development frameworks is that it produces Java code using the Spring Framework. The outcome is an honest-to-goodness Spring application, not a separate framework coded in a language that's foreign to many corporate development organizations.

More information about Spring Roo can be found at <http://www.springsource.org/roo>.

SPRING EXTENSIONS

In addition to all of the projects described up to this point, there's also a community-driven collection of Spring extensions at <http://www.springsource.org/extensions>. A few of the goodies you'll find there include

- An implementation of Spring for the Python language
- Blob storage

- db4o and CouchDB persistence
- A Spring-based workflow management library
- Kerberos and SAML extensions for Spring Security

1.4 What's new in Spring

It's been almost three years since I wrote the second edition of this book, and a lot has happened in the intervening time. The Spring Framework has seen two significant releases, each bringing new features and improvements to ease application development. And several of the other members of the Spring portfolio have undergone major changes.

We'll cover many of these changes throughout this book. But for now, let's briefly size up what's new in Spring.

1.4.1 What's new in Spring 2.5?

In November 2007, the Spring team released version 2.5 of the Spring Framework. The significance of Spring 2.5 was that it marked Spring's embrace of annotation-driven development. Prior to Spring 2.5, XML-based configuration was the norm. But Spring 2.5 introduced several ways of using annotations to greatly reduce the amount of XML needed to configure Spring:

- Annotation-driven dependency injection through the `@Autowired` annotation and fine-grained auto-wiring control with `@Qualifier`.
- Support for JSR-250 annotations, including `@Resource` for dependency injection of a named resource, as well as `@PostConstruct` and `@PreDestroy` for life-cycle methods.
- Auto-detection of Spring components that are annotated with `@Component` (or one of several stereotype annotations).
- An all-new annotation-driven Spring MVC programming model that greatly simplifies Spring web development.
- A new integration test framework that's based on JUnit 4 and annotations.

Even though annotations were the big story of Spring 2.5, there's more:

- Full Java 6 and Java EE 5 support, including JDBC 4.0, JTA 1.1, JavaMail 1.4, and JAX-WS 2.0.
- A new bean-name pointcut expression for weaving aspects into Spring beans by their name.
- Built-in support for AspectJ load-time weaving.
- New XML configuration namespaces, including the context namespace for configuring application context details and a `jms` namespace for configuring message-driven beans.
- Support for named parameters in `SqlJdbcTemplate`.

We'll explore many of these new Spring features as we progress through this book.

1.4.2 What's new in Spring 3.0?

With all of the good stuff in Spring 2.5, it's hard to imagine what could possibly follow in Spring 3.0. But with the 3.0 release, Spring one-upped itself with the continuation of the annotation-driven theme and several new features:

- Full-scale REST support in Spring MVC, including Spring MVC controllers that respond to REST-style URLs with XML, JSON, RSS, or any other appropriate response. We'll look into Spring 3's new REST support in chapter 11.
- A new expression language that brings Spring dependency injection to a new level by enabling injection of values from a variety of sources, including other beans and system properties. We'll dig into Spring's expression language in the next chapter.
- New annotations for Spring MVC, including `@CookieValue` and `@RequestHeader`, to pull values from cookies and request headers, respectively. We'll see how to use these annotations as we look at Spring MVC in chapter 7.
- A new XML namespace for easing configuration of Spring MVC.
- Support for declarative validation with JSR-303 (Bean Validation) annotations.
- Support for the new JSR-330 dependency injection specification.
- Annotation-oriented declaration of asynchronous and scheduled methods.
- A new annotation-based configuration model that allows for nearly XML-free Spring configuration. We'll see this new configuration style in the next chapter.
- The Object-to-XML (OXM) mapping functionality from the Spring Web Services project has been moved into the core Spring Framework.

Just as important as what's new in Spring 3.0 is what's not in Spring 3.0. Specifically, starting with Spring 3.0, Java 5 is now required, as Java 1.4 has reached end-of-life and will no longer be supported in Spring.

1.4.3 What's new in the Spring portfolio?

Aside from the core Spring Framework, there's also been exciting new activity in the projects that are based on Spring. I don't have enough space to cover every detail of what's changed, but there are a few items that I think are significant enough to mention:

- *Spring Web Flow 2.0* was released with a simplified flow definition schema, making it even easier to create conversational web applications.
- With Spring Web Flow 2.0 came *Spring JavaScript* and *Spring Faces*. Spring JavaScript is a JavaScript library that enables progressive enhancement of web pages with dynamic behavior. Spring Faces allows use of JSF as a view technology within Spring MVC and Spring Web Flow.
- The old Acegi Security framework was completely overhauled and released as *Spring Security 2.0*. In this new incarnation, Spring Security offers a new configuration schema that dramatically reduces the amount of XML required to configure application security.

Even as I was writing this book, Spring Security continued to evolve. Spring Security 3.0 was recently released, further simplifying declarative security by taking advantage of Spring's new expression language to declare security constraints.

As you can see, Spring is an active, continuously evolving project. There's always something new that aims to make developing enterprise Java applications easier.

1.5 Summary

You should now have a good idea of what Spring brings to the table. Spring aims to make enterprise Java development easier and to promote loosely coupled code. Vital to this is dependency injection and AOP.

In this chapter, we got a taste of dependency injection in Spring. DI is a way of associating application objects such that the objects don't need to know where their dependencies come from or how they're implemented. Rather than acquiring dependencies on their own, dependent objects are given the objects that they depend on. Because dependent objects often only know about their injected objects through interfaces, coupling is kept low.

In addition to dependency injection, we also saw a glimpse of Spring's AOP support. AOP enables you to centralize logic that would normally be scattered throughout an application in one place—an aspect. When Spring wires your beans together, these aspects can be woven in at runtime, effectively giving the beans new behavior.

Dependency injection and AOP are central to everything in Spring. Thus you must understand how to use these principal functions of Spring to be able to use the rest of the framework. In this chapter, we've just scratched the surface of Spring's DI and AOP features. Over the next few chapters, we'll dig deeper into DI and AOP. Without further ado, let's move on to chapter 2 to learn how to wire objects together in Spring using dependency injection.