Expert Spring MVC and Web Flow

Seth Ladd
with Darren Davison,
Steven Devijver and Colin Yates

Apress®

Expert Spring MVC and Web Flow

ISBN-13 (pbk): 978-1-59059-584-8

ISBN-10 (pbk): 1-59059-584-X

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Lead Editor: Steve Anglin
Technical Reviewers: Rob Harrop, Keith Donald
Editorial Board: Steve Anglin, Dan Appleman, Ewan Buckingham, Gary Cornell, Jason Gilmore, Jonathan Hassell, James Huddleston, Chris Mills, Matthew Moodie, Dominic Shakeshaft, Jim Sumser, Matt Wade
Project Manager: Sofia Marchant
Copy Edit Manager: Nicole LeClerc
Copy Editor: Stephanie Provines
Assistant Production Director: Kari Brooks-Copony
Production Editor: Katie Stence
Compositor and Artist: Van Winkle Design Group
Proofreader: Nancy Sixsmith
Indexer: Broccoli Information Management
Cover Designer: Kurt Krames
Manufacturing Director: Tom Debolski

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail `orders-ny@springer-sbm.com`, or visit `http://www.springeronline.com`.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, e-mail `info@apress.com`, or visit `http://www.apress.com`.

The source code for this book is available to readers at `http://www.apress.com` in the Source Code section.

To my father, who brought home
that old 1200-baud modem from work and
kick-started this crazy journey.
—Seth Ladd

To Mum and Dad, for always encouraging my curiosity.
And to my wife, Lisa, for being my wife.
—Darren Davison

For Beeky and the wriggler for putting up
with the late evenings, and Bruce and Jessie
for missing out on the walks.
—Colin Yates

For Filiz.
—Steven Devijver

To my wonderful wife, Keri,
and our little bundle of joy, Annabelle.
—Keith Donald

Contents at a Glance

Contents

About the Authors

SETH LADD is a software engineer and professional Spring Framework trainer and mentor specializing in object-oriented and testable web applications. He started his own company building websites at age 17, but now enjoys having a real job. Currently working for Camber Corporation, Seth has built and deployed systems for NEC, Rochester Institute of Technology, Brivo Systems, and National Information Consortium. He has architected and developed enterprise applications in Java and C for both the server and remotely connected embedded devices. He enjoys speaking and teaching, and is a frequent presenter at local Java user groups and at corporate developer conferences. Seth is very thankful for living and working in Kailua, Hawaii, with his wife.

DARREN DAVISON is a principal consultant for UPCO, specializing in J2EE and open source Java technologies. He has been involved with Spring since the summer of 2003, well before its 1.0 release, and he used the framework to underpin a global intranet site for an investment bank. Darren has previously worked for multinational manufacturing and engineering companies on e-business, infrastructure, and many web-based projects.

Away from work, Darren enjoys the never-ending journey of discovery that is GNU/Linux. When not in front of a computer screen, he likes reading and any form of live entertainment.

STEVEN DEVIJVER is an experienced Java developer who started developing J2EE applications in 2000. In 2003 he discovered the Spring Framework, and since then he has been one of its most enthusiastic users. Steven is a senior consultant at Interface21, teaching hundreds of students every year about the Spring Framework.

COLIN YATES is a J2EE principal architect who specializes in web-based development. He has been a freelance consultant for the past three years and has worked in a number of environments, both structured and chaotic. Since graduating with a software engineering degree in 1997, he has held a number of positions, including development lead, principal systems engineer, mentor, and professional trainer. His principal skill set includes mentoring others, architecting complex problems into manageable solutions, and optimizing development processes.

Colin was first introduced to the Spring Framework in January 2003 by his mentors, Peter Den Haan and David Hewitt, and he has never looked back. After a couple of years using the Spring and Hibernate technology stack to good effect, in May 2005 he became one of the early adopters of Spring Web Flow, finally finding the missing item in the web development toolbox.

A self-confessed addict of the green bar that comes from following test-driven development and XP, Colin regularly frustrates new team members by introducing a continuous build environment.

When not hanging around the Spring support forums (`http://forum.springframework.org`), Colin can be found out walking with his wife and two dogs, practicing martial arts, attending his local church, or preparing for the arrival of his first child.

About the Technical Reviewers

KEITH DONALD is a software consultant specializing in delivering customer-driven, enterprise-class Java applications. An experienced developer and mentor, Keith has built applications for customers spanning a diverse set of industries, including network management, information assurance, food services, education, and retail. He has extensive experience translating business requirements into technical solutions.

Keith has been involved with the Spring Framework as a user and core contributor since July 2003. He is the founder of the Spring Rich Client Project, an emerging module built on core Spring that substantially reduces the time and effort required to build a well-architected, enterprise-ready Java desktop application. He is also the colead of the Spring Web Flow module, a core Spring web offering that lets developers model business processes that span many screens in a logical manner.

Keith enjoys speaking and teaching on technical and business software-related topics, and has a career-oriented weblog where he frequently posts articles. Contact Keith at `keith@interface21.com`.

ROB HARROP is a software consultant specializing in delivering high-performance, highly scalable enterprise applications. He is an experienced architect with a particular flair for understanding and solving complex design issues. With a thorough knowledge of both Java and .NET, Rob has successfully deployed projects across both platforms. He has extensive experience across a variety of sectors, in particular retail and government.

Rob has been a core developer of the Spring Framework since June 2004 and currently leads the JMX and AOP efforts. In addition to his work on the Spring core, Rob leads the Spring Modules project, which is working to provide Spring integration for a variety of popular useful open source tools. He cofounded UK-based software company Cake Solutions Limited in May 2001, having spent the previous two years working as the lead developer for a successful dot-com startup.

Rob is the author of five books, including *Pro Spring*, a widely acclaimed, comprehensive resource on the Spring Framework. He is a member of the JCP and is involved in the JSR-255 Expert Group for JMX 2.0.

Acknowledgments

A book is never written by the authors alone. It is the product of many people's expertise and hard work, time, and superhuman efforts. This book belongs to everyone who had a hand in producing it.

I'd like to first thank my wife, who has the patience of an angel. Her love and support has been monumental through this endeavor.

My coauthors deserve huge thanks, as they have added their unique and invaluable knowledge and insight to make this a stronger book than I could have ever produced alone. Thanks Darren, Steven, Keith, and Colin!

No one would be reading this book if it weren't for the talent and professionalism at Apress. Specifically, I owe my heartfelt appreciation to Sofia Marchant and Beckie Brand for coordinating the many moving parts and making sure the book is the best it can be. A huge shout-out is owed to Stephanie Provines, without whom we would have capitalized Spring MVC 12 different ways. Her attention to detail was impressive and extremely valued. I specifically want to thank Steve Anglin as well, for giving me this opportunity. And to all the people behind the scenes, I am forever indebted to you.

I had the pleasure of having Rob Harrop perform the technical review for the book. His advice was always accurate, helpful, and professional. Thank you, Rob, I was honored to have you as part of the team.

Thanks to Erwin Vervaet, Dan Leuck, and Colin Sampaleanu for their expert opinions while reviewing the book. Thanks to Kathleen Fitzgerald for the photo shoot. And finally, thanks to the Spring Framework developers and community, from whom I have learned an incredible amount about software development.

Seth Ladd

I thank Seth Ladd for the opportunity to coauthor this book and for writing this excellent book in the first place. I also thank Rob Harrop for sharing his insights of Spring Web MVC and for doing the technical review of this book. Many thanks to the core Spring developers for creating and constantly extending this amazing framework. Thanks also go to Erwin Vervaet and Keith Donald for creating Spring Web Flow. I also thank my family for supporting me. I especially thank my girlfriend, Filiz, for her support, for proofreading, and for the warmth and energy she gives to me. Thank you all.

Steven Devijver

CHAPTER 1

Introduction

I can still remember the time I first realized what the Spring Framework was and how it could help me. I was tasked with building a web application that will register new businesses with the local government, and being a Java shop this meant the standard set of frameworks at the time: Struts, JavaServer Pages (JSP), and Hibernate. Having built many applications with these technologies, we dove right into development.

When beginning a new application, I always want to improve a few things from the last product development cycle. This time around, it was time to get serious about two things, unit testing and good object-oriented design. Sure, I had written plenty of unit tests before, but I had never begun a project by writing tests first. And although I've been studying and developing with OOP for many years now, I continue to learn new techniques that help the design of the application retain sustainability in the face of change.

So, off we went developing the application, writing tests for the domain model, creating a service layer (a façade for the web layer to integrate with), and beginning the build-out of the Struts layer. Each layer in the system seemed to progress nicely, but that's exactly when we ran into trouble.

As integration between layers began, we noticed that it became harder and harder to write good tests for the system. The application was using the Service Locator pattern to integrate the service layer and the web layers together. This pattern was implemented using a static lookup, which proved impossible to change for our unit tests. The question soon became, "How do we integrate these components such that both writing tests and running in production is simple and efficient?"

Enter the Spring Framework.

More precisely, enter an introduction article about the Spring Framework, posted to TheServerSide (`http://www.theserverside.com`). The original article has since been updated: `http://www.theserverside.com/articles/article.tss?l=SpringFramework`. I still remember printing it out, stapling it together, and sitting back down to my desk to see what all the fuss was about. Could it really help me create easily testable applications? Could it really bring OOP back to web development? There was only one way to find out.

I passed the article off to the boss, and I still remember his Aha moment after reading it. We decided to go for it and use the framework to integrate the components through the newfangled Dependency Injection. This led to easily testing the components, which led to better code, which led to happier clients. We then replaced our in-house Data Access Object (DAO) framework, one thing led to another, and we had a highly tested, full-blown Spring MVC application.

Of course, ripping out all of the Struts code and in-house cruft took time and energy, but we found we could do it in stages, lowering the risk of the integration. We made some mistakes and wrote lots of code, and in the end we had a better product—with a better design and a clear vision of how we wanted to write web applications from that point onward. In other words, we found what we were looking for in Spring MVC for our Java web applications.

My hope is that you can use this book to peer deeper into Spring MVC and learn new and interesting ways to use the framework to enhance your applications. We found that Spring MVC makes doing the right thing easier, and sometimes simply possible, and we hope you'll find as much joy using it as we do.

—Seth Ladd

Skipping Ahead

If you are the impatient type, you've probably skipped this chapter altogether and headed for the code. If you're still here, we have a recommendation for you. If you want to jump ahead and start with building a Spring MVC application, feel free to check out Chapter 4. There you will find elementary details on how to start building your first Spring MVC application.

We also recommend that you return to the previous chapters to learn about the theory and background of web application creation with Spring MVC. It will help to provide the context for the rest of the book.

How to View This Book

You should look at this book as your in-depth guide to the many features and functions of Spring MVC, including tips and tricks to get the most out of this flexible framework. This book also contains some best practices for developing well-designed and decoupled web applications.

This book is part guidebook, part tutorial, part web development manual. This book works best as a companion to *Pro Spring* by Rob Harrop and Jan Machacek (Apress, 2005), because it does not cover the Spring Framework in a general sense. It is dedicated to and focused on the best ways to write web applications using the Spring Framework and Spring MVC.

Roadmap

This book covers a lot of ground. Use this roadmap and chapter outline for a quick overview of what you will find inside and where.

- Chapter 1 is, well, this chapter you're reading now. It contains an overview of the book and its target audience, as well as where to go for more information and support.
- Chapter 2 is a refresher on the Spring Framework. If you are new to Spring, this can help paint the picture of why the framework exists and what problems it is trying to solve. Entire books are devoted to Spring, but this chapter can kick-start your discovery of the framework. If you are brand-new to the framework, you should purchase a full book on Spring, such as *Pro Spring*.

- Chapter 3 covers the architecture and design of typical Spring MVC applications. Light on code but heavy on design, this chapter presents details on the common layers found in web applications and some simple guidelines to build applications that take full advantage of the Spring Framework.
- Chapter 4 shows you the goods, with a jump start on Spring MVC. The impatient will find this a good starting point to get the feel of a real application. This chapter doesn't go into much detail, but it does take what you've learned from Chapter 3 to build some real functionality.
- Chapter 5 goes into detail about the real workhorse of Spring MVC: the `DispatcherServlet`. In this chapter you'll find all the ancillary services that all web applications require and how they can be configured and extended. Services like multipart file upload support and Locale resolution are covered here.
- Chapter 6 outlines and explains all of the different `Controller` options found in the framework. `Controllers` are written by you to handle incoming web requests, much like servlets or Struts Actions. Spring MVC provides a rich menagerie of `Controllers` to help with many different use cases and requirements.
- Chapter 7 introduces the view layer. Here you will find a tour of how views are managed and how they are integrated into a full Spring MVC application. Darren Davison, committer on Spring's view technologies, contributed both Chapters 7 and 8.
- Chapter 8 builds upon its predecessor and informs you how to integrate the popular view technologies with Spring MVC. JSP, Velocity, FreeMarker, and XSLT are just a few of your options for rendering the view, all covered in this chapter.
- Chapter 9 covers the Validation Framework. It also introduces Valang, a new and exciting validation system to make writing custom validation rules quick and easy. Steven Devijver, the author of Valang and Spring Framework committer, contributed Chapter 9.
- Chapter 10 provides examples of and discussion on testing your Spring MVC applications, including Spring's handy mocks and stubs for the Servlet API. We take the view that testing should be quick and painless, so we use a combination of simple unit tests and mock objects to write tests that run inside your IDE (and outside of your container).
- Chapters 11 and 12 cover the cutting-edge Spring Web Flow, a framework for writing conversational use cases on the web. This project, originally developed by Erwin Vervaet and brought into the Spring Framework fold by Keith Donald, allows you to declaratively build use cases that span multiple requests. Colin Yates provided these chapters.
- Appendix A introduces an excellent tool for documenting your Spring applications. The BeanDoc tool, written and maintained by Darren Davison, is like Javadoc for your bean definition XML files. This handy and easy tool integrates with your build to produce HTML documentation complete with images of the dependencies between beans. This appendix was contributed by Darren Davison, author of BeanDoc.
- Appendix B provides a bit of a sidebar; it introduces one way to integrate AJAX technologies into your Spring-powered web application. Darren Davison explains how to integrate DWR, or Direct Web Remoting (`http://getahead.ltd.uk/dwr`), with your Spring MVC applications.

Target Audience

Even though this book's title contains the word *expert*, you don't need to be an expert in Java or Spring to take advantage of it. However, to get the most out of this book, you should be familiar with Java and have created at least one web application with it.

You won't find discussions on basic Servlet API constructs or how to set up and configure your favorite servlet container. Many great books and resources—including countless web resources—already exist for this. We assume that you have at least a passing knowledge of what the Servlet API provides and how to deploy a Java web application. We also assume you are a competent Java developer, familiar with the language and its APIs.

Although you need not be a Spring Framework expert, it helps if you have investigated it to get a feel for what it is and what it brings to the table. We merely provide an introduction to the framework in this book. We recommend that you have a reference resource handy to turn to when we mention a Spring concept that you might not be familiar with.

If you are familiar with Java web programming and curious how Spring MVC stacks up against other request/response web frameworks, then this book will certainly help you determine that.

If you have built a few web applications with Spring MVC, we believe this book can still offer you great value. We provide many little tips and tricks, including some best practices for making the most from the web architecture in general. This book also covers some of the motivations for the designs of the components of Spring MVC, providing valuable insight into why the elements were built that way and how they connect.

For More Information

When you run into a situation that this book can't cover, you'll find that the Spring Framework has a vibrant and supportive community ready to help you out. The Spring community is made up of Java developers who take OOP, testability, and good design seriously, so you'll be in good company.

- The Spring Framework's home page, `http://www.springframework.org`, is the place to get news about the framework and links to many resources found on the web. Use this as a jumping-off point to downloads, forums, CVS, and issue tracker services.

- The Spring Framework Support Forums, `http://forum.springframework.org`, are your first choice when you want to ask a question or have a problem. Here you can choose from many forums, including those dedicated to Spring MVC and Spring Web Flow, and even one on architectural issues. These forums are active and helpful.

- The user mailing list is largely deprecated in favor of the support forums. However, you can access the archives via Spring's SourceForge page, `http://sourceforge.net/projects/springframework`. There you will also find the developers' mailing list, useful if you want to track development issues.

- You will find that the excellent Reference Manual, available from `http://www.springframework.org/documentation`, is up-to-date and quite full of content. Spring is one open-source project that does not skimp on its bundled documentation.

- Spring uses JIRA for its issue and bug tracking, found at `http://opensource2.atlassian.com/projects/spring/secure/Dashboard.jspa`. You can use this site to register new bugs you have found or to check whether someone else has discovered the issue first. This site also has the roadmap for future versions of the framework.
- For more on Spring Web Flow, that project has a very active Wiki page found at `http://opensource2.atlassian.com/confluence/spring/display/WEBFLOW/Home`. There you will find more tutorials, documentation, and links to articles on this up-and-coming project.

With the Spring Framework, there is no shortage of support options available, including many other books and professional consulting organizations and individuals.

Sample Applications

Sometimes looking at raw code is the only way to make the light bulb go off. If you're stuck and want to see how others might do it, Spring comes with many sample applications with full source code. These are excellent opportunities to investigate real working apps to see examples of Spring MVC and its integration with the rest of the application.

The sample applications can be found in the `samples` directory of the Spring Framework distribution or CVS repository.

Table 1-1. *Sample Web Applications*

Name	Description
countries	Demonstrates paged list navigation, locale and theme switching, localized view definitions, page composition through view definitions, and generation of PDF and Excel views.
imagedb	Demonstrates BLOB/CLOB handling, native JDBC connection handling, multipart file uploads, and Velocity integration.
jasperdemo	Demonstrates using JasperReports as the view technology.
JPetStore	Full application with all layers, using either Spring MVC or Struts for the web layers. Also demonstrates different remoting options.
PetClinic	Demonstrates integration with JDBC, Hibernate, Apache OJB, and Oracle TopLink. Also demonstrates JMX integration.
webapp-minimal	Minimal web application structure, including build scripts.

Spring 2.0

This book was written while Spring 2.0 was under development, so everything mentioned here will work with 2.0 or earlier. Nothing is 2.0 specific, so don't worry if you are using an earlier version of the framework.

The biggest addition to Spring's web capabilities with Spring 2.0 is the formal bundling of Spring Web Flow and Spring Portlet support. Spring MVC stays largely the same as previous versions, but does gain a few helpful simplifications and shortcuts. The changelog for the latest version is currently found at `http://static.springframework.org/spring/docs/current/changelog.txt`.

Summary

With so many options available for web frameworks, many of them perfectly fine solutions, it might come down to which framework is simply more enjoyable to work with. We believe that using Spring MVC will not only lead you to better designs and code, but also inspire *fun* developing with it. It really is a joy to apply good OOP design techniques and to write applications that are easily tested.

We have found that using Spring MVC has enhanced our ability to develop and deliver quality applications, and we want you to have the same level of success that we have enjoyed. So go forth, use Spring MVC, and bring OOP back to web programming!

CHAPTER 2

Spring Fundamentals

The Spring Framework has pumped new life into Java development. In the period immediately following the dot com bubble burst, Java applications were facing an uncertain future. The initial promises of J2EE had been thoroughly debunked, .NET was poised to offer a strong alternative, and the industry was generally sobering. Companies began to expect more application for less money and effort, and it wasn't certain that the J2EE platform would be able to deliver.

After the release of Rod Johnson's *Expert One-on-One J2EE Design and Development (Programmer to Programmer)* (Wrox, 2002) and its eventual evolution into the Spring Framework, the Java landscape had a new beacon of hope. The Spring Framework encapsulates a refreshing new beginning to Java development. First and foremost, it has enabled the return of the plain old Java object (POJO) to enterprise development. The framework combines best practices learned from actual deployments, with best-of-breed third-party utilities, to deliver a complete package.

Before we dive into Spring MVC and Web Flow, we feel it important to touch on a few very important concepts from the Spring Framework that we will rely on for the rest of the book. The Spring Framework has a unique, lightweight/full-featured duality, and we won't attempt to glance over the framework in this chapter. That job has been performed quite successfully by other works such as *Pro Spring*, or the Spring documentation. We wish to reintroduce only the core principles we believe to be important. If you are new to Spring, or need a refresher, there are many great resources available. Refer to *Pro Spring*, by Harrop and Machacek (Apress, 2005), or the online Spring Framework documentation (`http://www.springframework.org`).

Inversion of Control

You might hear the terms *Inversion of Control* and *Dependency Injection* used interchangeably, but in fact they are not the same thing. Inversion of Control is a much more general concept, and it can be expressed in many different ways. Dependency Injection is merely one concrete example of Inversion of Control.

Inversion of Control (or IoC) covers a broad range of techniques that allow an object to become a passive participant in the system. When the IoC technique is applied, an object will relinquish control over some feature or aspect to the framework or environment. Some examples of control include the creation of objects or the delegation to dependent objects. IoC can remove these concerns from objects with Dependency Injection and aspect-oriented programming, respectively.

IoC Example

Many systems of medium to large scale require some sort of a security system. Performing authorization, authentication, and accounting is a concern of the application that typically cuts across the entire object model.

A first attempt at implementing security might place the authorization calls directly inside the domain object, effectively forcing the object to control security itself. This can lead to a bloated object model implementation, because now the security code has become interlaced across the system, obscuring the business logic (Listing 2-1).

Listing 2-1. *Simple POJO with Control of Security*

```
public class BankAccount {
    public void transfer(BigDecimal amount, BankAccount recipient) {
        SecurityManager.hasPermission(this, Permission.TRANSFER,
            SecurityContext.getCurrentUser());
        recipient.deposit(this.withdraw(amount));
    }

    public void closeOut() {
        SecurityManager.hasPermission(this, Permission.CLOSE_OUT,
            SecurityContext.getCurrentUser());
        this.open = false;
    }

    public void changeRates(BigDecimal newRate) {
        SecurityManager.hasPermission(this, Permission.CHANGE_RATES,
            SecurityContext.getCurrentUser());
        this.rate = newRate;
    }
}
```

Listing 2-1 shows a simple `BankAccount` class with typical business logic methods (`transfer`, `closeout`, `changeRates`). These method implementations are cluttered with nearly duplicate security-related checks, obscuring the original intent of the business logic. In addition, the `SecurityManager` calls add a dependency that will be difficult to work with when we unit test this class.

To remove the clutter and simplify the implementation, the `BankAccount` should let go of this security responsibility altogether (Listing 2-2). In effect, the control over security should be turned inside out from the object to the surrounding framework.

Listing 2-2. *Simple POJO with Security Concerns Relinquished*

```
public class BankAccount {
    public void transfer(BigDecimal amount, BankAccount recipient) {
        recipient.deposit(this.withdraw(amount));
    }

    public void closeOut() {
        this.open = false;
    }

    public void changeRates(BigDecimal newRate) {
        this.rate = newRate;
    }
}
```

This *Inversion of Control* has freed the object from the cross-cutting constraint of security authorization. The end result is a removal of duplicate code and a simplified class that is focused on its core business logic.

So how do we get the security checks back into the system? You can add the authorization mechanism into the execution path with a type of IoC implementation called *aspect-oriented programming (AOP)*. Aspects are concerns of the application that apply themselves across the entire system. The `SecurityManager` is one example of a system-wide aspect, as its `hasPermission` methods are used by many methods. Other typical aspects include logging, auditing, and transaction management. These types of concerns are best left to the framework hosting the application, allowing developers to focus more on business logic.

An AOP framework, such as Spring AOP, will interject (also called *weaving)* aspect code transparently into your domain model at runtime or compile time. This means that while we may have removed calls to the `SecurityManager` from the `BankAccount`, the deleted code will still be executed in the AOP framework. The beauty of this technique is that both the domain model (the `BankAccount`) and any client of the code are unaware of this enhancement to the code.

To explain a little more, it helps to talk about a concrete implementation of AOP as applied by Spring. The Spring Framework uses what is called *proxy-based AOP*. These proxies essentially wrap a target object (the `BankAccount` instance) in order to apply aspects (`SecurityManager` calls) before and after delegation to the target object. The proxies appear as the class of the target object to any client, making the proxies simple drop-in replacements anywhere the original target is used.

■ **Note** Spring also supports AspectJ, which is implemented not with proxies but with compile-time weaving. Weaving is a more capable AOP implementation and a nice alternative to the more simple proxy solution.

Figure 2-1 illustrates the sequence of calls when a `BankAccount` is closed out, using proxy-based AOP to perform the security checks.

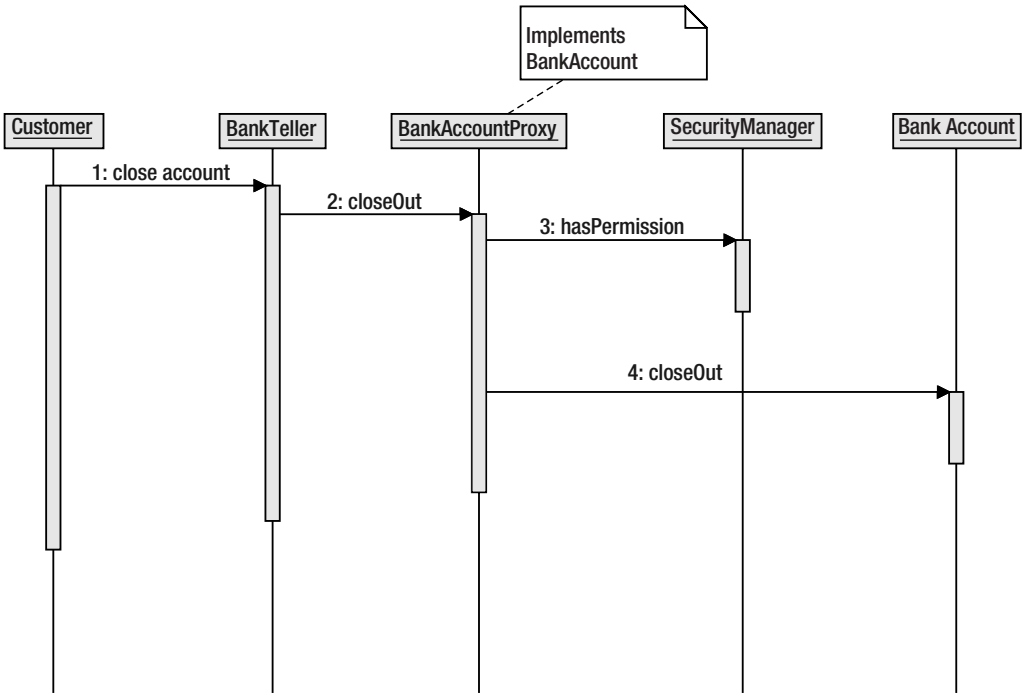

Figure 2-1. *Call sequence with AOP SecurityManager*

As you can see from the preceding diagram, the `SecurityManager` calls are handled by the proxy before it delegates to the real `BankAccount` class.

For more information on aspect-oriented programming, we recommend Ramnivas Laddad's *AspectJ in Action* (Manning Publications, 2003) or Adrian Colyer's *Eclipse AspectJ: Aspect-Oriented Programming with AspectJ and the Eclipse AspectJ Development Tools* (Eclipse/Addison-Wesley, 2004). For more information on how Spring supports and implements AOP, consult *Pro Spring*.

Summary

When you apply an AOP solution to your system, you are actually applying a form of IoC. For instance, introducing security aspects to your object model is merely inverting the responsibility from the object layer to the framework layer.

The Spring Framework provides a robust implementation of AOP. Every time you use the declarative transaction management, you are using AOP. Thus, you are using another form of IoC. In this form, the object model becomes a passive participant in transaction management, relinquishing control over when to commit or roll back to the framework.

To summarize, Inversion of Control is the broad concept of giving control back to the framework. This control can be control over creating new objects, control over transactions, or control over the security implementation. Aspect-oriented programming is one technique to

implement IoC. Dependency Injection is another technique to implement, which we will discuss in the following section.

Dependency Injection

The concept of Dependency Injection is core to the Spring Framework. A specialization of Inversion of Control, Dependency Injection is a technique that frameworks use to wire together an application. The framework performs the work of connecting an application's dependencies together, removing the wiring logic and object creation from the application code completely.

We will contrast Dependency Injection with an older technique named the Service Locator pattern. We will show how the Service Locator pattern harms the testability and flexibility of an application. We then show how Dependency Injection, and Spring's implementation, fix this issue.

In nearly all applications there are at least two participants, and these two participants are required to somehow collaborate. The trick is to connect these two objects without locking in the connection or requiring a certain environment to even exist for the connection to be made.

For our example, we will consider the following use case. A cash register must obtain up-to-date prices for items being purchased. The prices are stored and calculated inside a large legacy system, but the cash register is physically located at the point of sale. The `CashRegister` object must have a reference to the price database to perform its work.

We begin by defining the interface to represent the cash register. It has one method, `calculateTotalPrice`, which takes a shopping cart and returns the total price for all items in the cart.

```
public interface CashRegister {
    public BigDecimal calculateTotalPrice(ShoppingCart cart);
}
```

Next, we define the interface for the service that will provide the real time price lookup. This interface has one method, `lookupPrice`, to return the price for an item.

```
public interface PriceMatrix {
    public BigDecimal lookupPrice(Item item);
}
```

Finally, we will create the implementation of the `CashRegister` interface. It simply creates its own dependency, an instance of `PriceMatrix`.

```
public class CashRegisterImpl implements CashRegister {
    private PriceMatrix priceMatrix = new PriceMatrixImpl();

    public BigDecimal calculateTotalPrice(ShoppingCart cart) {
        BigDecimal total = new BigDecimal("0.0");
        for (Item item : cart.getItems()) {
            total.add(priceMatrix.lookupPrice(item));
        }
        return total;
    }
}
```

There are three major issues with the preceding implementation. The first is that every instance of `CashRegisterImpl` has a separate instance of `PriceMatrixImpl`. If it is costly to create or maintain that object, then this is a waste of system resources. With heavy services (those that are remote or those that require connections to external resources such as databases) it is preferable to share a single instance across multiple clients.

The second and most important issue is that the `CashRegisterImpl` now has concrete knowledge of the implementation of `PriceMatrix`. The `CashRegisterImpl` class should not know the details of the implementation of its dependency interfaces. By explicitly creating the instance of `PriceMatrixImpl`, the `CashRegisterImpl` has tightly coupled itself to the concrete implementation class.

The third issue with the preceding implementation is a direct result of the tight coupling to the implementation class. By explicitly creating its own dependent objects, the `CashRegisterImpl` creates a difficult test situation. One of the most important tenets of writing unit tests is to divorce them from any environment requirements. The unit test itself should run without connecting to outside resources. If we were to test the `calculateTotalPrice` method as is, we would have no choice but to require a fully functioning `PriceMatrixImpl`. Not only would this slow down our unit test runs, it would now couple our tests to resources we can't control. What if the price returned by `lookupPrice` changes over time? Our unit tests would have to stay in sync with the physical resource, increasing the burden of maintaining the tests.

If we can't interact with a real `PriceMatrix`, how do we test our `calculateTotalPrice` method? We will create a stub instance of `PriceMatrix`, one where we can control the conditions and outcomes of the method calls. This technique is called a *mock*, and it is very useful in unit tests. For a full explanation of mock objects and their uses, refer to *JUnit in Action* by Husted and Massol (Manning Publications, 2003). In the meantime, it is sufficient to think about a mock object as a fake object that looks like a particular class, but whose behavior is controlled by the test author.

As you can see, we have noticed three deficiencies of the above implementation. The responsibility of creating the object is left to the method, prohibiting the use of a shared `PriceMatrixImpl`. The method is also difficult to test, as we need to somehow insert a mock object in place of the real thing. Most importantly, the client code is now aware of implementation details of its dependency, creating two tightly coupled classes.

To address the first issue, we will remove the explicit instantiation of the `PriceMatrix` dependency. This frees the `CashRegisterImpl` from the burden of object creation and from the knowledge of any physical implementation details. More importantly, the `PriceMatrixImpl` is no longer located inside the `CashRegisterImpl` instance. By moving the dependency out of the client object, it is no longer solely owned by `CashRegisterImpl`, and can now easily be shared among all classes. The question then becomes, how do we now locate the dependency?

Service Locator

Enter the *Service Locator pattern*, our first attempt at fixing the method. The Service Locator pattern encapsulates the actions taken to obtain a reference to the object required. This shields the client from knowing how, or even where, to obtain a reference to the object.

This pattern emerged as a workaround from using Java Naming and Directory Interface (JNDI) to obtain references to other Enterprise JavaBeans (EJBs) in a J2EE application. Using JNDI to obtain a simple object reference can be cumbersome and can require a lot of defen-

sive programming. To protect the client, and to reduce code duplication, the Service Locator pattern was born. It usually manifests itself as a static method, returning a single instance of the requested object.

We can now change our initial code to the following:

```
public class CashRegisterImpl implements CashRegister {

    private PriceMatrix priceMatrix;

    public CashRegisterImpl() {
        priceMatrix = ServiceLocator.getPriceMatrix();
    }

    public BigDecimal calculateTotalPrice(ShoppingCart cart) {
        BigDecimal total = new BigDecimal("0.0");
        for (Item item : cart.getItems()) {
            total.add(priceMatrix.lookupPrice(item));
        }
        return total;
    }

}
```

Using this Service Locator, the class no longer has to manage object creation. The location of the actual instance of `PriceMatrix` is now independent of the client class. In a managed environment, such as J2EE servers, this point is critical. The act of obtaining the resource is now hidden from the client so that it may get on with the work at hand. The first problem we had with the original implementation has been solved.

The other benefit of using this Service Locator is that our client has no knowledge of the concrete implementation of `PriceMatrix`. This shields the client, allowing the implementation of `PriceMatrix` to evolve independently of `CashRegisterImpl`.

The third problem, the lack of testability, is unfortunately still with us, even after the change to the Service Locator. Creating a unit test for the preceding method is extremely difficult, because the implementation relies on a functioning `PriceMatrix` object. The test should be written with a mock `PriceMatrix`, but there is no way to insert the mock object during testing.

The Service Locator pattern, implemented here as a static method, falls down in a test scenario. The static `ServiceLocator.getPriceMatrix` is difficult to change during a test run. The locator method has to be told to return a mock `PriceMatrix` during the test, and a real `PriceMatrix` during deployment.

This situation has illustrated the need to swap different implementations of `PriceMatrix` without affecting the client. To effectively do this, the client (in this case, the `calculateTotalPrice` method) must not actively participate in the construction or retrieval of the resource. The resource must be *given* to the client.

Dependency Injection

Instead of the lookup call to the Service Locator, the framework can provide a reference of type `PriceMatrix` to the `CashRegisterImpl` class. This reduces the active work the client has to do to obtain a reference to zero, making it a passive client of the framework.

The responsibility for object creation and object location has been inverted, from the class to the framework. This wiring of dependencies is Dependency Injection in action.

Spring supports Dependency Injection in two main ways, and both are extremely simple. In fact, both use plain old Java idioms.

■ **Tip** There is a third way, called *method-based injection*, which takes advantage of Spring's AOP support. It's more complicated, however no less useful, so it is not mentioned here. Consult the documentation or *Pro Spring* for more information on method-based injection.

The first type of Dependency Injection we will cover is *constructor-based injection*. This concept merely means the dependency is provided via the constructor at object creation time. For instance, to use constructor-based injection on our `CashRegisterImpl` object, the class would look as shown in Listing 2-3.

Listing 2-3. *Constructor-Based Dependency Injection*

```
public class CashRegisterImpl implements CashRegister {
    private PriceMatrix priceMatrix;

    public CashRegisterImpl(PriceMatrix priceMatrix) {
       this.priceMatrix = priceMatrix;
    }

    public BigDecimal calculateTotalPrice(ShoppingCart cart) {
        BigDecimal total = new BigDecimal("0.0");
        for (Item item : cart.getItems()) {
            total.add(priceMatrix.lookupPrice(item));
        }
        return total;
    }
}
```

That's it! The framework is responsible for obtaining the reference to a `PriceMatrix` object and then calling the constructor of the `CashRegisterImpl` and providing the `PriceMatrix` object.

The class is obeying what is commonly called the Hollywood Principle. In other words, the framework's contract is "Don't call me; I'll call you." Even more technically, the contract is "Don't ask for the resource; I'll give it to you."

Another type of Dependency Injection that's more popular with Spring is setter-based injection; just what it sounds like, it uses setter methods to inject the dependency. To use setter-based injection, the constructor will be removed and replaced with a simple JavaBean-compliant setter, as shown in Listing 2-4.

Listing 2-4. *Setter-Based Dependency Injection*

```
public class CashRegisterImpl implements CashRegister {
    private PriceMatrix priceMatrix;

    public setPriceMatrix(PriceMatrix priceMatrix) {
       this.priceMatrix = priceMatrix;
    }

    public BigDecimal calculateTotalPrice(ShoppingCart cart) {
        BigDecimal total = new BigDecimal("0.0");
        for (Item item : cart.getItems()) {
            total.add(priceMatrix.lookupPrice(item));
        }
        return total;
    }
}
```

The framework simply calls the setter with the `PriceMatrix` instance, and now `CashRegister` has everything it needs. By not using the constructor, the `CashRegister` object can now be created without the immediate availability of a `PriceMatrix`, making its life cycle a bit more flexible.

Which type should you use, constructor-based injection or setter-based injection? This is purely a matter of taste. The Spring Framework does not mandate one method or the other. In fact, you may even use both methods on the same bean. Those who prefer constructor-based injection claim that it enforces a correctly initialized object due to the intrinsically self-validating nature of constructors. A potential downside to constructor-based injection is the risk of a proliferation of constructors to accommodate different use cases. As use cases grow, each requiring different sets of dependencies, so shall grow the number of constructors.

Those who prefer setter-based injection argue that it is more flexible (able to mix and match for different situations) or that it is self-documenting. For instance, compare the following two bean definition examples in Listings 2-5 and 2-6, and consider which tells you more about the relationship between the object and its dependencies.

Listing 2-5. *Example A*

```
<bean id="addressService" class="org.example.addr.AddressServiceImpl">
  <constructor-arg ref="zipCodeService" />
  <constructor-arg ref="uspsValidator" />
  <constructor-arg ref="googleMapService" />
</bean>
```